Estimation de performances et de consommation énergétique de systèmes de stockage à base de mémoire flash dans les systèmes embarqués

Pierre Olivier

THESE / UNIVERSITE DE BRETAGNE-SUD
sous le sceau de l'Université européenne de Bretagne
pour obtenir le titre de
DOCTEUR DE L'UNIVERSITE DE BRETAGNE-SUD
Mention : Sciences et Technologies de l'Information et de la Communication
Ecole doctorale SICMA

présentée par

Pierre Olivier

Laboratoire des Sciences Techniques de l'Information, de la Communication et de la Connaissance
UMR3192

Estimation de performances et de consommation énergétique de systèmes de stockage à base de mémoire flash dans les systèmes embarqués

Thèse soutenue le 1er décembre 2014
devant le jury composé de :

Mme. Cécile Belleudy, *Rapportrice*
Maître de conférences HDR - Université de Nice Sophia Antipolis

M. Luc Bouganim, *Rapporteur*
Directeur de recherches - INRIA Paris-Rocquencourt

M. William Jalby, *Examinateur*
Professeur - Université de Versailles St Quentin en Yvelines

M. Frank Singhoff, *Examinateur*
Professeur - Université de Bretagne Occidentale

M. Gaël Thomas, *Examinateur*
Professeur - Télécom SudParis

M. Pierre Ficheux, *Examinateur*
Ingénieur - OpenWide

M. Eric Senn, *Directeur de thèse*
Maître de conférences HDR - Université de Bretagne Sud

M. Jalil Boukhobza, *Co-encadrant*
Maître de conférences - Université de Bretagne Occidentale

Université de Bretagne Sud

Rapport de thèse

Pour l'obtention du diplôme de

Docteur de l'Université de Bretagne Sud

Discipline scientifique : Informatique

Laboratoire : Lab-STICC CNRS UMR 6285

Présentée et soutenue publiquement par

Pierre OLIVIER

Le : 1er décembre 2014

Estimation de performances et de consommation énergétique de systèmes de stockage à base de mémoire flash dans les systèmes embarqués

Composition du jury :

Rapporteur	Mme Cécile Belleudy	Maître de conférences HDR à l'Université Nice Sophia Antipolis
Rapporteur	M. Luc Bouganim	Directeur de recherche à l'INRIA Paris-Rocquencourt
Examinateur	M. William Jalby	Professeur à l'Université de Versailles St Quentin en Yvelines
Examinateur	M. Frank Singhoff	Professeur à l'Université de Bretagne Occidentale
Examinateur	M. Gaël Thomas	Professeur à Télécom SudParis
Examinateur	M. Pierre Ficheux	Ingénieur dans l'entreprise OpenWide
Co-encadrant	M. Jalil Boukhobza	Maître de conférences à l'Université de Bretagne Occidentale
Directeur de thèse	M. Eric Senn	Maître de conférences HDR à l'Université de Bretagne Sud

Résumé

Maîtriser et optimiser les performances et la consommation énergétique dans les systèmes embarqués, omniprésents dans notre quotidien, est aujourd'hui crucial. Pour ce faire, des techniques d'estimation de ces métriques sont utilisées dans des environnements où la réalisation de mesures peut être difficile voir impossible. Les systèmes embarqués considérés dans le cadre de cette thèse sont complexes, et utilisent un système d'exploitation. Cette complexité fait que l'obtention d'estimations concernant les performances et la consommation passe généralement par la réalisation de modèles. Ces modèles sont par la suite exécutés dans des simulateurs. Pour construire les modèles il est nécessaire de comprendre le fonctionnement des systèmes en question, et leur comportement concernant les performances et la consommation.

Dans ce travail de thèse on se concentre sur le service du stockage secondaire dans un système d'exploitation embarqué. La mémoire flash NAND est le principal média de stockage dans l'embarqué. Ce type de mémoire possède des contraintes d'utilisation fortes, ce qui mène à l'introduction de couches de gestion complexes dans les systèmes de stockage à base de mémoire flash. Ces couches impactent les performances et la consommation du système embarqué en général. Parmi ces couches, on distingue la FTL (*Flash Translation Layer*) présente dans les périphériques à base de mémoire flash, et les systèmes de fichiers dédiés (*Flash File Systems*, *FFS*). Dans ce travail on cible particulièrement l'utilisation des FFS. On peut constater un manque de travaux dans la littérature en ce qui concerne les techniques de modélisation des performances et de la consommation des systèmes à base de FFS. On considère le système d'exploitation Linux car il est utilisé dans de nombreux systèmes embarqués, et inclut le support des FFS les plus populaires.

Les contributions apportées dans cette thèse s'articulent autour d'une méthodologie de modélisation pour l'estimation des performances et de la consommation des systèmes de stockage embarqués de type FFS. Cette méthodologie est divisée en trois phases : l'*exploration*, la *modélisation* et la *simulation*.

En phase d'exploration on identifie les différents éléments d'un système de stockage de type FFS qui impactent les performances et la consommation. Il s'agit de composants matériels : la mémoire flash, le CPU et la RAM ; et de composants logiciels, correspondant au différents niveaux de la pile logicielle de gestion du stockage flash sous Linux : le système de fichiers virtuel (*Virtual File System*, *VFS*), le système de fichiers, et le pilote NAND.

La phase de modélisation correspond à la représentation au sein de modèles de l'impact des différents éléments identifiés en phase précédente. Des modèles de différents types sont proposés : les modèles *fonctionnels* représentent les algorithmes de la couche de gestion. Les modèles de *performances* et de *consommation* caractérisent un profil concernant ces métriques pour une plate-forme donnée. Les modèles de *charge d'entrées / sorties* (*E/S*) sont utilisés pour représenter la charge appliquée par une application sur le système de stockage. Enfin, des modèles spécifiques aux composant matériel flash sont présentés : il s'agit de modèles *structurels*, représentant son architecture, et *opérationnels*, relatifs aux différentes opérations supportées par la mémoire flash. A chaque modèle de performances et de consommation est associé une méthodologie d'extraction de paramètres permettant de construire le profil d'une plate-forme matérielle et logicielle donnée. On prend pour cas d'étude pour le modèle fonctionnel le FFS JFFS2. On donne un exemple d'application de la méthodologie d'extraction de paramètres sur la carte *Mistral Omap3evm*. On montre également comment les modèles peuvent être généralisés pour représenter un système à base de FTL.

Les modèles sont implémentés dans un simulateur nommé *OpenFlash*. Cet outil permet d'une part d'obtenir des estimations concernant les performances et la consommation des systèmes de

stockage à base de mémoire flash. D'autre part, l'outil fournit une infrastructure de développement pour y intégrer de nouvelles couches de gestion (de type FFS et FTL). Il peut ainsi être utilisé pour le prototypage de couches de gestion, la validation d'optimisations, et les études comparatives concernant les performances et la consommation énergétique. Le simulateur est également utilisé pour valider les modèles précédemment présentés. L'erreur d'estimation reste dans la majeure partie des cas sous la barre des 10 %.

Mots clefs : mémoire flash NAND, systèmes embarqués, stockage secondaire, modélisation, performances, consommation, simulation

Abstract

Today, controlling and optimizing embedded systems performance and power consumption is critical. Due to time or financial constraints, performing measurements campaigns can be difficult in some environements. In that case estimation techniques are used. The embedded systems considered in this work are complex, and use an operating system. Because of that complexity, estimations concerning performance and power consumption are obtained by model building. Those models are then ran on a computer program, a simulator. Building models implies a deep understanding on how the considered embedded system works, and what influences its performance and power consumption profiles.

In this thesis we focus on NAND flash memory based secodary storage in an embedded system running an operating system. NAND flash is the main storage media in embedded systems. There are strong constraints to deal with when operating this type of memory. In order to do so, flash management layers are used. They can be of two types : the *Flash Translation Layer* (FTL) is used in flash-based peripherals. Dedicated Flash File Systems are used in embedded systems with raw flash chips. Those management layers impact the system performance and power consumption. In this work we mainly target FFS. As one can observe, there is a lack of work in the litterature concerning FFS performance and power consumption modeling and estimation techniques. In this work we consider the Linux operating system, because it is widely used in embedded systems, and because Linux supports the most popular FFS.

The contributions of this thesis relies on to a three steps modeling methodology : the *exploration*, *modeling* and *simulation* phases.

In the exploration phase, we identify the primary elements of a FFS storage system impacting performance and power consumption in an embedded system. These are hardware elements : the flash memory, the CPU and RAM. We also identify software elements, corresponding to the multiple levels of the software stack responsible for flash storage management with Linux : the virtual file system, the flash file system and the NAND driver.

In the modeling phase, we characterize the impact of these elements on the storage system performance and power consumption. Models of various types are proposed. *Functional* models represent the flash management layer algorithms. *Performance* and *power consumption* models are used to depict a profile for a given hardware / software platform, related to those metrics. The *I/O workload* models represent the workload submitted by an application to the storage system. Finally, we propose models related to the flash memory component : the *structural* model represents its architecture, and the *operational* model depicts the various operations supported by the flash memory chip. For each performance and power consumption model, we propose a parameter exctraction methodology used to build a power and performance profile for a given hardware and software platform. We present as a case study a set of models concerning the JFFS2 FFS executed with Linux running on the *Mistral Omap3evm* embedded board. We also show how those models can be generalized to represent a FTL based storage system.

The whole set of presented models are implemented in a simulator, named *OpenFlash*. OpenFlash can be used to obtain estimation for the performance and power consumption of NAND flash based storage systems. Moreover, the simulator offers a development infrastructure allowing the user to integrate new flash management layers (FFS and FTL) in OpenFlash. Thus, The tool can be used to prototype new flash management layers, validate optimizations and perform comparative studies in termes of performance and power consumption. The simulator is also used to validate the previously presented models, the error percentage stays under 10 %.

Keywords : NAND flash memory, embedded systems, secondary storage, modeling, performance, power consumption, simulation

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Liste des publications

On présente ici les différentes publications réalisées au cours de ce travail de thèse.

Publications en lien direct avec le travail de thèse

- Revues & chapitre de livre :
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **Revisiting Read-ahead Efficiency for Raw NAND Flash Storage in Embedded Linux**, in *proceedings of the 4th Embed With Linux (EWiLi) Workshop*, 2014 (to be published in *ACM SIGBED Review* ;
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **Flashmon v2 : Monitoring Raw Flash memory Accesses on Embedded Linux**, in **ACM SIGBED Review** vol. 11, issue 1, Special issue of the Embed With Linux (EWiLi) International workshop, Toulouse, France, 2013 ;
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **Flash Based Storage in Embedded Systems**, in *Encyclopedia of Embedded Computing Systems*, IGI Global Editor, DOI : 10.4018/978-1-4666-3922-5, ISBN13 : 9781466639225, April 2013 ;
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **On Benchmarking Embedded Linux Flash File Systems**, in *ACM SIGBED Review* vol. 9, issue 2, Special issue of the Embed With Linux (EWiLi) International workshop, Lorient, France, 2012.
- Conférences internationales :
  - Pierre Olivier, Jalil Boukhobza, Mathieu Soula, Michelle Le Grand, Ismat Chaib Draa and Eric Senn, **A Tracing Toolset for Embedded Linux Flash File Systems**, in *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, Bratislava, Slovakia, December 2014 ;
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **Modeling Driver Level NAND Flash Memory I/O Performance and Power Consumption for Embedded Linux**, in *proceedings of the IEEE 11th International Symposium on Programming and Systems (ISPS)*, Algiers, Algeria, April 2013 ;
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **Micro-benchmarking Flash Memory File-System Wear Leveling and Garbage Collection : a Focus on Initial State Impact**, in *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, Paphos, Cyprus, December 2012.
- Conférences nationales :
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **Toward a Unified Performance and Power Consumption NAND Flash Memory Model of Embedded and Solid State Secondary Storage Systems**, in *proceedings of the SoC-SiP GDR Workshop*, Lyon, June 2013 ;
  - Pierre Olivier, Jalil Boukhobza, Eric Senn, **Performance Evaluation of Flash File-systems**, in *proceedings of the GDR SoC-SiP Workshop*, Paris, June 2012 ;

Publications annexes :

- Revues :
  - Hamza Ouarnoughi, Jalil Boukhobza, Pierre Olivier, Loic Plassart, Ladjel Bellatreche, **Performance analysis and modeling of SQLite embedded databases on flash file systems**, in *Design Automation for Embedded Systems*, Springer, October 2014 ;
  - Jalil Boukhobza, Pierre Olivier, Stéphane Rubini, **A Scalable and Highly Configurable Cache-Aware Hybrid Flash Translation Layer**, in *Computers*, 3(1),36-57 ; doi :10.3390/computers301003, March 2014.
  - Pierre Olivier, Jalil Boukhobza, **Un Système de Cache Hiérarchique pour les E/S Présentant des Motifs d'Accès Séquentiels pour les Mémoires Flash NAND**, in *Techniques et Sciences Informatique (TSI)* vol. 32, issue 2, pp. 203-228, April 2013 ;

— Pierre Olivier, Jalil Boukhobza, **A Hardware Time Manager Implementation for the Xenomai Real-Time Kernel of Embedded Linux**, in *ACM SIGBED Review* vol. 9, issue 2, Special issue of the Embed With Linux (EWiLi) International workshop, Lorient, France, 2012.

• Conférences internationales :

— Jalil Boukhobza, Pierre Olivier, Loic Plassart, Hamza Ouarnoughi, Ladjel Bellatreche, **Embedded Databases on Flash Memories : Performance and Lifetime Issues, the case of SQLite**, in *proceedings of Embedded Real-time Software and Systems (ERTS)* Toulouse, France, 2014 ;

— Jalil Boukhobza, Pierre Olivier, Stéphane Rubini, **CACH-FTL : A Cache Aware Configurable Hybrid Flash Translation Layer**, in *EUROMICRO International conference on Parallel, Distributed, and Network based processing (PDP)*, Belfast, February 2013.

Introduction

1 Contexte et problématique

Aujourd'hui, les systèmes informatiques embarqués sont plus que jamais présents dans notre environnement. On constate une présence croissante de systèmes complexes, fortement orientés vers les applications multimédias, comme les lecteurs audio / vidéo, smart-phones, tablettes, etc. Ces systèmes présentent un besoin crucial de performances pour offrir une qualité de service acceptable à l'utilisateur. On peut le voir en constatant l'évolution de la puissance des processeurs embarqués ces dernières années. En effet, le nombre d'opérations à virgule flottante par seconde pouvant être réalisé par ces processeurs embarqué évolue de manière exponentielle (Rajovic et coll., 2013). La consommation énergétique est également une métrique critique dans ce contexte. En effet, la majeure partie de ces systèmes embarqués fonctionnent sur batteries. La consommation du système impacte donc son autonomie, ainsi que sa durée de vie. Or, on constate que la densité de puissance (puissance consommée pour une surface donnée dans une puce) des processeurs double tous les trois ans (Skadron et coll., 2003). En revanche, l'évolution de la quantité d'énergie pouvant être stockée dans les batteries ne suit pas cette progression (Lahiri et coll., 2002). Par conséquent, les concepteurs de ces systèmes tentent aujourd'hui de maximiser les performances de leur produits, tout en en minimisant la consommation.

La maîtrise et l'optimisation des métriques de performances et de consommation sont donc cruciaux. Ces objectifs requièrent une compréhension fine des principes de fonctionnement des multiples éléments logiciels et matériels composants les systèmes embarqués, et impactant leur performances et leur consommation. La capacité à estimer ces métriques au plus tôt lors de la conception permet de maîtriser de manière fine les performances et la consommation du produit final. De plus, les techniques d'estimation sont également utiles dans des environnements où la réalisation de mesures est parfois difficile ou fastidieux. Ces techniques peuvent ainsi être utilisées pour valider de nouvelles propositions de systèmes, ou des optimisations de systèmes existants. Pour obtenir des estimations on se base sur la construction de modèles, qui sont des représentation de la structure et du fonctionnement des systèmes réels étudiés. Les techniques d'estimation des performances et de la consommation peuvent être de type analytique, ou par simulation via des modèles fonctionnels (Kumar Rethinagiri, 2014). Dans les systèmes complexes (munis d'un système d'exploitation) tels que ceux considérés dans ce travail de thèse, les estimations sont généralement obtenues via des méthodes de simulation. La simulation est nécessaire car au niveau du système, les éléments impactant les performances et la consommation sont multiples et variés. Il y a donc aujourd'hui un fort besoin de méthodologies de modélisation et d'outils d'estimation par la simulation des métriques de performances et de consommation concernant les systèmes informatiques embarqués.

On constate aujourd'hui une croissance importante du marché de la mémoire flash NAND, du fait de son utilisation comme principal média de stockage dans le domaine de l'embarqué. Cette croissance est également due à son apparition progressive dans les systèmes informatiques en général, où les disques SSD (*Solid State Drives*) commencent à remplacer les disques durs. IC Insights (2012) rapporte qu'en 2012, la valeur totale du marché de la mémoire flash (plus de 30 milliards de dollars) a surpassé celui de la mémoire DRAM. Dans ce travail de thèse, on se concentre sur la gestion du stockage secondaire à base de mémoire flash, mettant en œuvre la hiérarchie mémoire de ces systèmes embarqués. La mémoire flash NAND est fortement présente dans ces systèmes grâce aux

nombreux avantages qu'elle offre, entre autres une forte densité de stockage, un faible prix au bit, et une bonne résistance aux chocs. Néanmoins, de part ses principes de fonctionnement fondamentaux, des contraintes fortes s'appliquent lors son utilisation. Des mécanismes de gestion complexes sont donc implémentés au sein des systèmes de stockage à base de mémoire flash. Ces mécanismes impactent fortement les performances et la consommation du système de stockage. Ils peuvent être divisés en deux grandes classes, les couches de traduction (*FTL* pour *Flash Translation Layer*) et les systèmes de fichiers dédiés (*FFS* pour *Flash File Systems*) spécifiques au domaine de l'embarqué.

Aujourd'hui les périphériques de stockage à base de FTL (clés USB, carte SD/MMC, puces eMMC, disques SSD) sont très répandus. L'utilisation d'une FTL se traduit par l'introduction d'une couche émulant un périphérique de type bloc dans les systèmes de stockage entre l'applicatif et la mémoire flash. L'abstraction apportée par la FTL pose un certain nombre de difficultés, comme le prouve l'introduction de nouvelles commandes (*TRIM*) spécifiques aux FTL dans les protocoles ATA, SCSI ou eMMC (Serial ATA International Organization, 2011) ; ou encore l'apparition de nouveaux systèmes de fichiers dédiés aux FTL comme *F2FS* Hwang (2012). Dans le domaine de l'embarqué, l'utilisation des FFS est très développée, et ce pour les raisons suivantes : par rapport aux FTL, ils sont moins coûteux, moins complexes, et constituent un lien direct entre l'applicatif et le média de stockage flash (UBIFS Contributors, 2009; Sowa, 2011). De plus, concernant la phase d'exploration des performances et de la consommation nécessaire dans tout processus de modélisation, il faut savoir que les FTL sont des systèmes propriétaires et fermés (*boîtes noires*). Les FFS supportés par Linux sont ouverts, concernant leur sources et donc leur fonctionnement : cela facilite grandement la phase d'exploration.

Bien que l'étude des FTL soit un domaine de recherche très actif, il n'existe actuellement que très peu de travaux concernant les performances et la consommation énergétique des FFS. Il n'existe notamment aucune étude approfondie concernant des techniques d'estimation des métriques de performances et de consommation dans le cadre de l'utilisation de FFS. La complexité des systèmes de gestion flash fait que l'obtention d'estimations sur les performances et la consommation énergétique passe fréquemment par la simulation. Cela est particulièrement vrai lorsque le système à propos duquel on souhaite obtenir des estimations n'est pas physiquement disponible. Alors qu'il existe plusieurs simulateurs permettant de représenter des systèmes à base de FTL (entre autres Kim et coll. (2009b); Hu et coll. (2011); Jung et coll. (2012); Dayan et coll. (2013)), il n'existe pas d'outil permettant de simuler des systèmes de type FFS. Les simulateurs existant permettent (A) l'obtention d'estimations concernant les performances et la consommation et, pour certains outils, (B) la possibilité de prototyper de nouveaux mécanismes de gestion pour mémoire flash. On constate donc qu'il est nécessaire de définir des méthodologies pour la caractérisation et la modélisation des métriques de performances et de consommation dans le cadre des systèmes de stockage de type FFS. Il y a également un besoin d'outils de simulation concernant ce type de systèmes.

Les systèmes embarqués complexes considérés dans ce travail de thèse impliquent l'usage d'un système d'exploitation (*OS* pour *Operating System*). On s'intéresse aux systèmes exécutant Linux, et ce pour plusieurs raisons. Premièrement, Linux est aujourd'hui l'un des principaux OS utilisés dans le monde de l'embarqué. Une étude récente réalisée par l'entreprise *UBM* (UBM Tech, 2013) montre que 50% des industriels sondés utilisent Linux dans l'un de leurs projets embarqués. Deuxièmement, Linux supporte l'intégralité des FFS les plus populaires. Enfin, Linux est *open-source*, ce qui signifie qu'il est possible d'étudier et de comprendre de manière fine comment fonctionnent les mécanismes internes à ce système d'exploitation.

On constate donc un besoin général d'augmentation des performances, et de réduction de la consommation énergétique dans les systèmes embarqués. Obtenir des estimations sur les performances et la consommation des systèmes embarqués au plus tôt lors de la conception de ces systèmes est critique pour maîtriser ces métriques au niveau du produit final. De plus, les techniques d'estimation

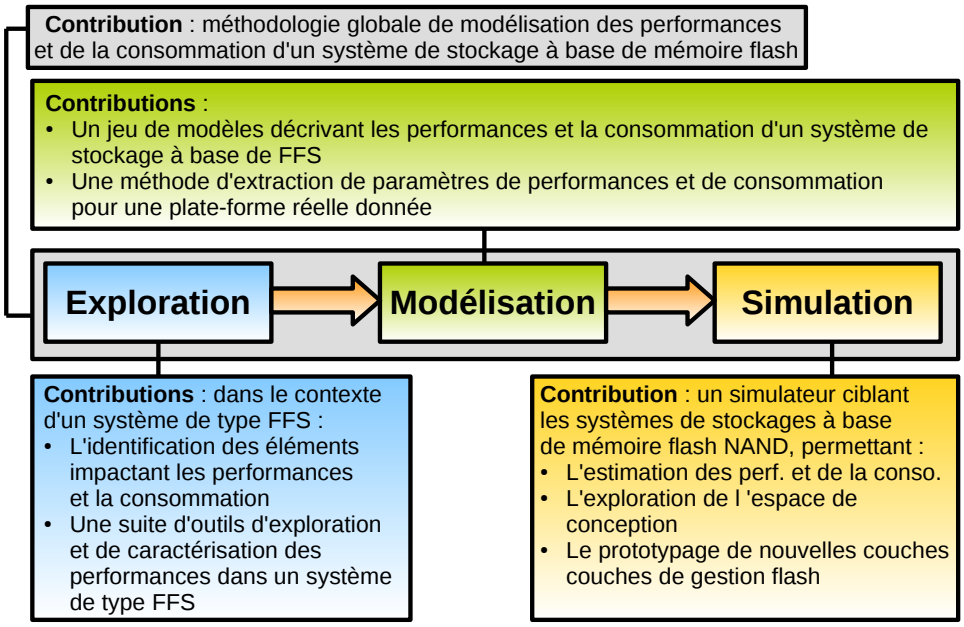


Figure 1 – Méthodologie adoptée dans le travail de thèse et contributions

sont très utiles dans certains contextes où des mesures ne peuvent être effectuées. Grâce à ces estimations on peut effectuer de l'exploration de l'espace de conception de systèmes embarqués, optimiser des systèmes existants, et prototyper de nouveux systèmes. Dans le contexte de ce travail de thèse, il est nécessaire de proposer des méthodes de caractérisation et de modélisation des performances et de la consommation dues à la gestion de la mémoire flash via un système de type FFS. On constate également un besoin d'outils de simulation implémentant ces modèles pour l'obtention d'estimations concernant les métriques de performances et de consommation.

2 Contributions

Pour répondre aux besoins cités ci-dessus, les contributions apportées par ce travail de thèse s'articulent autour d'une méthodologie de modélisation pour l'estimation par la simulation des performances et de la consommation des systèmes de stockage à base de mémoire flash. Cette méthodologie est constituée de trois étapes principales, illustrée sur la figure 1 : (A) *l'exploration* des comportements de ces systèmes, concernant les performances et de consommation énergétique ; (B) *la modélisation* de ces comportements et (C) l'implémentation de ces modèles au sein d'un outil de *simulation* permettant d'obtenir des estimations.

2.1 Exploration des métriques de performances et de consommation dans les systèmes de stockage embarqués de type FFS

La première étape de la méthodologie mise en œuvre est l'identification des éléments logiciels et matériels ayant un impact significatif sur les performances et la consommation du système de stockage embarqué de type FFS. Cette étape est cruciale car elle permet de comprendre le comportement de performances et de consommation du système de stockage. Elle permet également d'extraire les principaux éléments impactant ces métriques : c'est sur ces éléments que l'effort de modélisation est par la suite effectué.

Comme énoncé précédemment la gestion de la mémoire flash dans un système de stockage est complexe. Le nombre de couches de traitement situées entre l'applicatif et la mémoire flash elle-

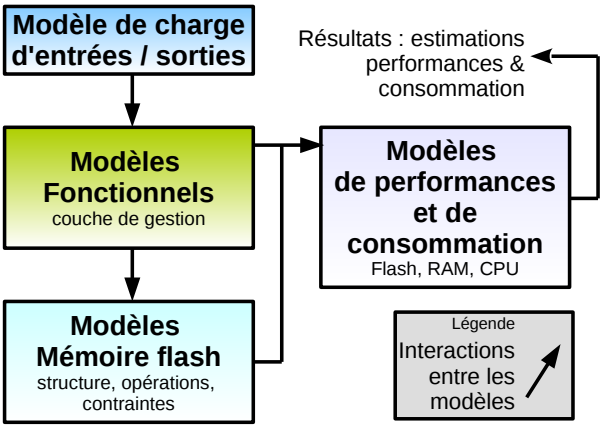


Figure 2 – Interactions entre les différents types de modèles présentés dans ce travail de thèse.

même est important. La méthodologie adoptée identifie et considère chacune de ces couches dans un contexte de type FFS sous Linux : (A) le système de fichiers virtuel (*Virtual File System*, *VFS*), (B) le FFS, (C) le pilote de la mémoire flash NAND et (D) la mémoire flash elle-même. On propose une suite d'outils permettant d'extraire des informations concernant les performances du système de stockage, et de caractériser ces performances, à tous les niveaux de gestion concernés. Ces outils sont par la suite utilisés dans les autres étapes de la méthodologie de modélisation présentée ici. Les métriques de consommation sont quant à elles extraites via la plate-forme de mesure de consommation à distance *OPEN-PEOPLE* (Senn et coll., 2012).

2.2 Modélisation des performances et de la consommation d'un système de stockage de type FFS

En se basant sur les conclusions de la phase d'exploration, l'étape suivante de la méthodologie correspond à la modélisation des différents éléments des systèmes de stockage impactant les performances et la consommation. Ces éléments sont multiples et de types variés, on définit donc plusieurs types de modèles, illustrés sur la figure 2, qui interagissent dans le but d'obtenir des estimations précises :

- Les modèles *structurels* et *opérationnels* représentent respectivement l'architecture et les opérations supportées par la ou les puce(s) flash contenue(s) dans le système ;
- Les modèles *fonctionnels* représentent les algorithmes de la couche de gestion flash (VFS, FFS, pilote) ;
- Les modèles de *performances* et de *consommation* représentent les profils de performances des différents composants matériels du système de stockage (flash, CPU, RAM) ;
- Enfin, le modèle de *charge d'entrée / sortie* (E/S) représente la charge de travail soumise par l'applicatif au système.

Les modèles abstraits cités ci-dessus sont génériques et adaptables à plusieurs types de FFS. Pour chacun des types de modèles abstraits, on donne (A) la méthodologie utilisée pour réaliser un modèle concret à partir de mesures sur une plate-forme matérielle réelle et (B) un exemple de modèle concret concernant le FFS JFFS2 exécuté sous Linux, tournant sur une plate-forme réelle, la carte *Mistral omap3evm* (Mistral Solutions, 2013). JFFS2 a été choisi premièrement car c'est un FFS mature et largement utilisé dans les systèmes embarqués, et deuxièmement car les algorithmes définissant son fonctionnement sont relativement simples, ce qui en fait un premier cas d'étude idéal.

Concernant la réalisation de modèles de performances et de consommation, on peut distinguer deux types d'approches : les approches *orientées composant* prennent pour cible un composant matériel spécifique, par exemple le CPU (Kumar Rethinagiri, 2014). Les approches *orientées service* ciblent les performances et la consommation d'un système ou ensemble de composants, dans le cadre d'un

service logiciel particulier offert par un système d'exploitation : ordonnancement, communication inter-processus, etc (Douhib, 2009; Ouni, 2013). Notre approche est principalement orientée service, le service ciblé étant le stockage embarqué à base de mémoire flash. Le composant matériel représentant la mémoire flash à lui été modélisé de manière relativement détaillée, on adopte donc pour ce dernier une approche orientée composant.

On montre également comment les types de modèles présentés ci-dessus sont adaptables pour la représentation de systèmes à base de FTL. Ils couvrent ainsi l'ensemble du spectre d'applications des systèmes de stockage à base de mémoires flash actuels.

2.3 Intégration et validation des modèles dans un simulateur, *OpenFlash*

La troisième et dernière étape de la méthodologie est l'implémentation des modèles au sein d'un simulateur à évènements discrets, à base de traces d'E/S Jain (2008). Cet outil nommé *OpenFlash*, écrit en C++, permet d'obtenir des estimations concernant les performances et la consommation d'un système de stockage simulé. Il s'agit d'un simulateur à événements discrets implémentant l'intégralité des modèles présentés en phase précédente. Les simulateurs existants ciblent des systèmes à base de FTL. Le simulateur proposé ici reprend les points forts des outils existants, comme le support d'architectures et d'opérations flash avancées. De plus, le simulateur apporte des fonctionnalités nouvelles telles que le support des FFS, et d'évènements dits *asynchrones*. Ces événements permettent de représenter, entre autres, des mécanismes de ramasse-miettes en arrière plan présents dans de nombreux systèmes de gestion flash actuels, de type FFS comme FTL.

Ce simulateur prend en entrée (A) une description du système à simuler (ensemble de paramètres pour les modèles) et (B) une description de la charge d'E/S appliquée au système simulé. En sortie, des estimations variées concernant les performances et la consommation sont disponibles. L'un des rôles de l'outil est donc l'exploration de l'espace de conception de systèmes de stockage. L'outil est également construit de manière à fournir à l'utilisateur une infrastructure de prototypage et de développement de nouvelles couches de gestion flash (FFS et FTL), pour évaluer de nouvelles propositions et les comparer à l'existant.

Dans cette troisième étape on effectue également une validation des modèles concrets présentés à l'étape précédente. Dans 91 % des expériences réalisées en validation, l'erreur d'estimation des modèles et du simulateur est inférieure à 10 %, et dans 74 % des cas cette erreur est sous la barre des 5%.

3 Plan du mémoire

Ce document est organisé de manière suivante :

Dans le premier chapitre on présente des concepts généraux concernant les systèmes de stockage à base de mémoire flash NAND, en particulier les contraintes relatives à l'utilisation de ce type de mémoire, et les mécanismes de gestion mis en place pour palier ces problèmes.

Le second chapitre constitue un état de l'art sur plusieurs sujets en forte relation avec le travail de thèse : le benchmarking de systèmes de stockage, les systèmes de fichiers dédiés aux mémoires flash, ainsi que l'exploration, la modélisation, et la simulation des performances et de la consommation des systèmes de stockage à base de ce type de mémoire.

Le troisième chapitre présente la phase d'exploration de la méthodologie : on y présente la méthode d'identification des éléments d'un système de type FFS qui impactent de manière significative les performances et la consommation de ce système.

Dans le quatrième chapitre on présente la phase de modélisation : les différents types de modèles représentant chacun des éléments identifiés dans l'étape précédente sont détaillés. On montre également un exemple d'application de ces modèles à une plate-forme réelle.

Le cinquième et dernier chapitre décrit dans les détails l'outil de simulation proposé. On y présente son fonctionnement interne, en particulier la manière dont les modèles y sont implémentés. De plus on utilise également cet outil pour effectuer une validation des modèles présentés dans l'étape précédente.

CHAPITRE 1

Les systèmes de stockage à base de mémoire flash NAND

Dans ce premier chapitre on présente les concepts basiques des systèmes de stockage à base de mémoires flash présents dans les systèmes informatiques actuels. Ce chapitre introduit des notions nécessaires à la compréhension du travail de thèse. De plus, il constitue un état de l'art sur la mémoire flash de type NAND (sous-type de mémoire flash considéré dans le cadre de ce travail de thèse) de manière générale.

On effectue tout d'abord une présentation globale de la mémoire de type flash en introduisant des concepts tels que sa position dans la classification des mémoires à semi-conducteurs, les principes physiques mis en œuvre pour son fonctionnement, et les différents sous-types de mémoire flash existants. Le focus est mis sur le sous-type nommé NAND, c'est en effet le sous-type concerné par ce travail de thèse. Après avoir présenté, a un niveau d'abstraction relativement haut, la structure et les opérations qu'il est possible d'appliquer sur les mémoires flash NAND, on décrit les principales contraintes relatives à l'utilisation de ce type de mémoire. Ces contraintes les suivantes : (A) l'impossibilité de mettre à jour des données en place, (B) l'usure de la mémoire au fil de son utilisation en écriture, et (C) le manque de fiabilité de ce type de mémoire flash.

Les contraintes liées à l'utilisation de la mémoire flash NAND font que la présence d'une couche de gestion est indispensable dans tout système de stockage contenant ce type de mémoire. Après la présentation de ces contraintes, on donne les principes généraux implémentés par les couches de gestion pour palier ces contraintes.

*Ces couches de gestion peuvent être divisées en deux grandes classes : la couche de traduction des périphériques à base de mémoire flash (*Flash Translation Layer*, FTL) et les systèmes de fichiers dédiés (*Flash File Systems*, FFS) utilisés avec les puces flash embarquées dans de nombreux systèmes informatiques. Ce chapitre présente ces deux classes de couches de gestion, et les notions importantes associées.*

Sommaire

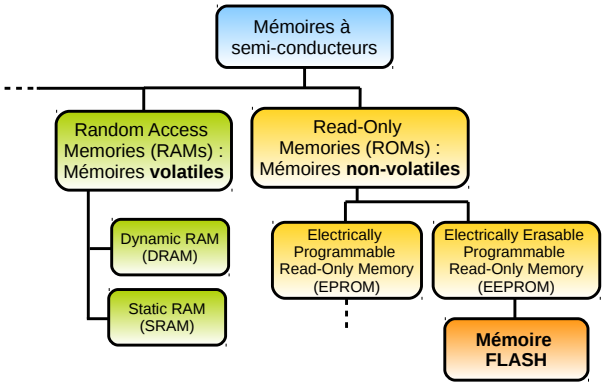


FIGURE 1.1 – Classification des mémoires flash dans l'arbre des mémoires à semi-conducteurs. Il existe d'autre types de mémoires à semi-conducteurs, non représenté dans ce schéma car sans rapport avec le travail présenté dans cette thèse.

1 Présentation générale des mémoires flash

1.1 La mémoire flash dans la classification des mémoires à semi-conducteurs

Dans un système informatique, le rôle d'une mémoire au sens large est, comme son nom l'indique, de mémoriser de l'information : c'est à dire d'assurer la persistance de données dans le temps. On peut classifier les mémoires à semi-conducteurs présentes dans les systèmes actuels en deux grandes catégories, selon leur propriété de volatilité. Les mémoires volatiles nécessitent une source d'alimentation externe pour pouvoir fonctionner, on parle également de mémoires vives. Les mémoires non-volatiles, ou mémoires mortes, ont la capacité de maintenir de l'information sans source d'alimentation externe (pour une durée de temps définie).

Comme illustré sur la figure 1.1, on distingue deux catégories principales de mémoires vives (*Random Access Memory*, RAM) : la mémoire vive dynamique (DRAM) joue aujourd'hui le rôle de mémoire principale dans la plupart des systèmes informatiques. On la retrouve également sous forme de mémoire tampon embarquée, accélérant les performances de certains périphériques de stockage comme les disques durs ou les SSD (*Solid State Drives*). La mémoire vive statique (SRAM) quant à elle est plus rapide pour ce qui est des temps d'accès que la mémoire DRAM, mais aussi plus coûteuse (prix au bit). On retrouve ainsi la mémoire SRAM là où les performances sont cruciales. C'est par exemple ce type de mémoire qui compose les registres et caches des microprocesseurs actuels.

Concernant les mémoires mortes (*Read-Only Memory*, ROM), on peut identifier plusieurs sous-catégories, classifiées selon la manière dont le contenu de la mémoire peut être modifié après sa construction. Une ROM classique n'est tout simplement pas modifiable après sa construction. Une mémoire PROM (*Programmable Read-Only Memory* est en général programmable une unique fois, contrairement aux mémoires EPROM (*Erasable Programmable Read-Only Memory*) et EEPROM ou E2PROM (*Electrically Erasable Programmable Read-Only Memory*) qui sont des mémoires effaçables et peuvent être reprogrammées de multiples fois. Alors que l'EPROM est effaçable via rayonnement ultra-violet, l'EEPROM est effaçable, comme son nom l'indique, par courant électrique. La mémoire flash est un sous-type d'EEPROM.

1.2 Différents types de mémoires flash

Les deux types principaux de mémoire flash sont nommés d'après la porte logique utilisée pour la construction d'une cellule mémoire de base : on distingue la flash NOR et la flash NAND, dont les caractéristiques sont représentées sur la figure 1.2.

La micro-architecture d'une puce de mémoire flash NOR fait qu'il est possible d'adresser le contenu

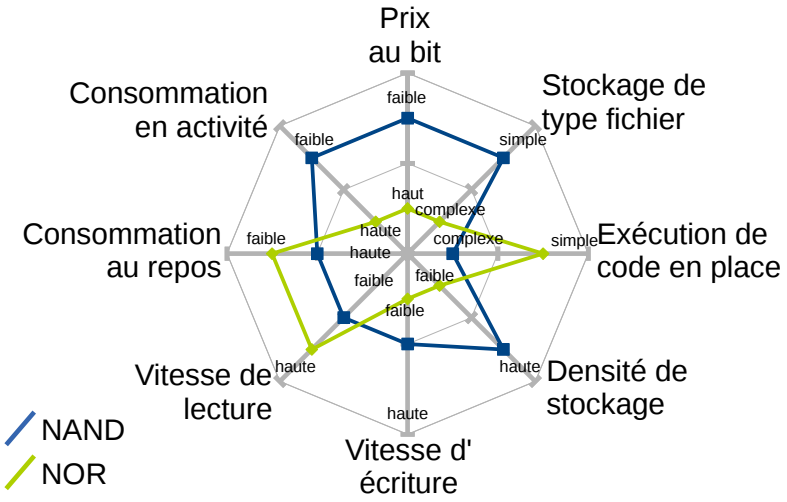


Figure 1.2 – Comparaison des caractéristiques des mémoires flash NAND et NOR selon différentes métriques. Image adaptée de Toshiba Inc. (2006).

de la mémoire de manière aléatoire (autrement dit à une granularité fine, par exemple un octet). La mémoire flash NOR présente de bonnes performances en lecture : Richter (2014a) rapporte une latence en lecture d'environ 20 ns. Ainsi, elle est utilisée dans un contexte où les performances en lecture sont cruciales : le stockage de code dédié à son exécution sur place. Cette technique d'exécution sur place (*eXecute In Place*, XIP (Forni et coll., 2007)) consiste à stocker et exécuter le code d'une application sur la flash, qui remplace donc une mémoire de type RAM. Les latences des mémoires flash NOR en écriture sont beaucoup plus grandes. On retrouve la flash NOR stockant le système d'exploitation de nombreux systèmes embarqués peu complexes, le code du BIOS des cartes mères de PC de bureau, ou encore le micro-logiciel (*firmware*) embarqué dans certains périphériques informatiques.

Les cellules des mémoires flash NAND sont, quant à elles, adressées par *pages* (paquets de données de taille fixée). Par rapport à la flash NOR, la densité de stockage de la flash NAND est plus élevée, et son prix au bit réduit. La flash NAND présente des latences en lecture plus grandes (entre 25 et 200 μs selon Grupp et coll. (2009) et Micron Inc. (2006)), mais également des latences en écriture plus équilibrées par rapport à celles de la flash NOR. Les mémoires flash NAND sont utilisées dans le contexte du stockage de données. Ce type de mémoire est très répandu dans les systèmes informatiques actuels. On retrouve la flash NAND jouant le rôle de stockage secondaire dans les systèmes embarqués complexes (par exemple les smartphones et tablettes actuels), dans les périphériques de stockage de données comme les disques SSD ou les clés de stockage USB. On trouve également de la mémoire flash NAND dans les cartes de stockage telles que les formats SD ou MMC, ou les lecteurs multimédias portables comme les lecteurs mp3.

La figure 1.2 présente un comparatif des mémoires flash NAND et NOR selon différentes métriques.

Il existe également d'autres types de mémoires flash moins répandus, utilisés notamment dans le domaine de l'embarqué (Hidaka, 2011). Ces sous-types de mémoires flash se caractérisent par la technologie utilisée définissant la structure de base d'une cellule mémoire, et les techniques nécessaires pour programmer et effacer une cellule. Chaque sous-type présente des bénéfices en performances, consommation énergétique, ou encore densité de stockage.

Le travail présenté dans cette thèse cible le stockage de données, on s'intéresse donc ici uniquement aux mémoires flash de type NAND.

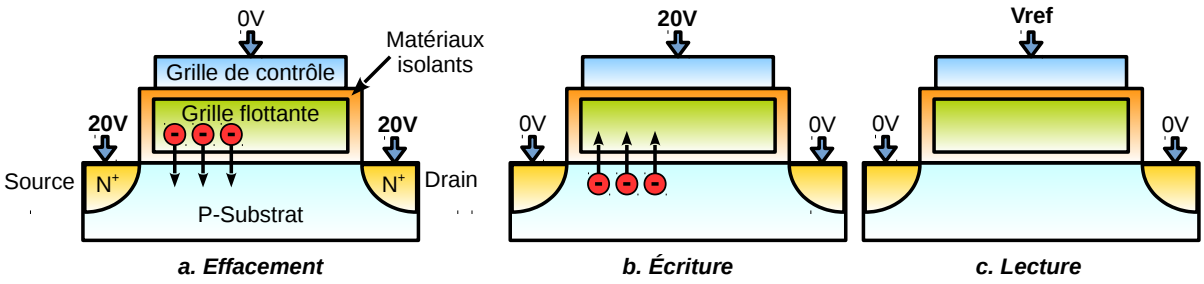


FIGURE 1.3 – Schéma d'une cellule de base de mémoire flash, composée d'un transistor à grille flottante, ainsi que les différentes opérations qui peuvent y être appliquées : effacement (a), programmation ou écriture (b), et lecture (c).

1.3 Principes physiques de fonctionnement

Dans cette sous-section on présente de manière générale les principes physiques de fonctionnement d'une cellule de mémoire flash. Les concepts présentés ici concernent les cellules de mémoire flash construites à partir de la technologie dite "à grille flottante". Une autre technologie existe également, appelée "piégeage de charge" (*charge trapping* en anglais) (Richter, 2014a; Tehrani, 2013).

Une cellule de base de mémoire flash (Bez et coll., 2003; Forni et coll., 2007) est composée d'un transistor à grille flottante. Comme illustré sur la figure 1.3, ce type de transistor est caractérisé par la présence d'une grille flottante, isolée électriquement car entourée par des matériaux diélectriques. Du fait de cette isolation, les électrons contenus dans cette grille flottante y sont maintenus pendant une grande période de temps, donnant à la mémoire flash sa propriété de non-volatilité. En appliquant une tension à la grille de contrôle ou au substrat, il est possible de charger et de décharger la grille flottante. Cela a pour effet la modification de la tension de seuil du transistor.

Il est possible d'appliquer 3 types d'opérations sur la cellule : l'effacement, la programmation (ou l'écriture), et la lecture. Ces opérations sont décrites ci-dessous et illustrées sur la figure 1.3.

— L'*effacement* consiste à vider la grille flottante de sa charge négative, c'est à dire les électrons qui y sont contenus. Cela est réalisé par émission par effet de champ (en anglais *Fowler-Nordheim tunneling*). Une tension élevée, d'environ 20 volts (Forni et coll., 2007), est appliquée au substrat alors qu'une tension nulle est appliquée à la grille de contrôle (voir figure 1.3 (a)) ;
— La *programmation* (écriture) consiste à charger négativement la grille flottante, et utilise également l'émission par effet de champ. Cette fois, une tension nulle est appliquée au substrat alors qu'une tension élevée est appliquée à la grille de contrôle. ce processus est illustré sur la figure 1.3 ;
— L'opération de *lecture* consiste à appliquer la tension de référence à la grille de contrôle du transistor. Si la grille flottante est chargée (négativement), le transistor est bloqué et le courant ne passe pas : cela correspond communément à un '0' logique stocké par la cellule. Si la grille n'est pas chargée le transistor est passant : c'est un 'l' logique.

On distingue plusieurs sous-types de cellules flash NAND (par conséquent plusieurs sous-type de puces flash NAND). Les puces de type *Single Level Cell* (SLC) stockent 1 bit par cellule mémoire. Le terme *Multi Level Cell* (Fazio et Bauer, 2007) dénote des puces pouvant stocker 2 bits par cellule, mais aussi de manière générale des puces pouvant stocker 2 bits ou plus dans cellule. Les cellules *Triple Level Cell* stockent 3 bits. L'augmentation de la densité de stockage et la baisse du coût au bit offertes par les cellules MLC et TLC sont contrebalancées par une baisse de la fiabilité, de l'endurance (capacité à supporter un nombre plus ou moins important d'effacements), et des performances. Les différences de performances et d'endurance entre les puces de type SLC et MLC sont présentées plus en détails dans les sections suivantes.

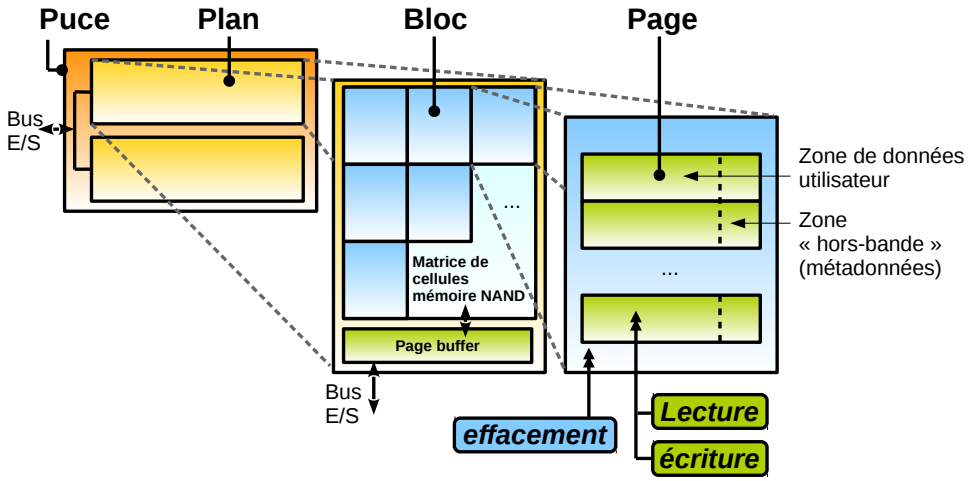


Figure 1.4 – Architecture simplifiée d'une puce de mémoire flash NAND.

1.4 Architecture hiérarchique simplifiée d'une puce de mémoire flash NAND

La figure 1.4 représente l'architecture simplifiée d'une puce de mémoire flash NAND, telle que vue dans ce travail de thèse. Il s'agit d'une vision haut niveau, relativement abstraite par rapport au niveau micro-architectural pour lequel on aurait une description beaucoup plus détaillée mais aussi plus complexe de l'architecture d'une puce.

Une puce de mémoire flash présente une structure hiérarchique. Une puce est composée d'un certain nombre de *plans*. Les plans contiennent des *blocs*, et les blocs contiennent des *pages*. La puce est munie d'un bus d'entrées / sorties (E/S) qui permet la réception de commandes et adresses, ainsi que le transfert de données depuis / vers le système informatique hôte. Ce bus est en général multiplexé ce qui signifie que les commandes, adresses et données partagent les mêmes broches. On retrouve, selon le modèle de puce, des bus de largeur 8 ou 16 bits.

a) Page

La page flash (appelée secteur dans certaines études) est la granularité avec laquelle on effectue des lectures et écritures sur la puce NAND. Une page flash est subdivisée en deux zones (voir figure 1.4), la première est appelée zone de données utilisateur (*user data area* en anglais). Cette zone constitue la plus grande partie de l'espace mémoire d'une page. La zone de données utilisateur des pages d'une puce contient les données effectivement stockées sur cette puce. La seconde zone constituant une page flash est la zone dite "hors bande" (en anglais *Out Of Band area*, OOB), autrement appelée *spare data area* (Sakui et Suh, 2007). C'est une zone contenant des méta-données relatives aux informations stockées en zone utilisateur. On y stocke par exemple des informations nécessaires au bon fonctionnement de la couche de gestion de la mémoire flash, comme des informations de parité pour codes correcteurs d'erreur, des méta-données pour la traduction d'adresses, des marqueurs de blocs usagés, et d'autres informations spécifiques aux différentes implémentations de mécanismes de gestion flash.

La taille d'une page flash (en octets) varie selon les modèles de puce NAND. On trouve au niveau des modèles de puces relativement âgés, nommés *small blocks* (Micron Inc., 2005) en anglais, des pages de 512 octets de données utilisateurs et 16 octets OOB. Aujourd'hui, une majeure partie des pages de puces de type SLC présente une taille de 2048 octets plus 64 octets OOB (modèles appelés *large blocks* en anglais, par opposition aux anciens). Pour les puces MLC, on retrouve des tailles de 2048 (64 OOB), 4096 (128 OOB), voir 8192 (128 OOB) octets (Grupp et coll., 2009)[1]. Sauf mention contraire,

1. Bien que datant de 2009, cette étude présente un nombre important de types de puces NAND (11), dont les caractéristiques sont toujours valables aujourd'hui.

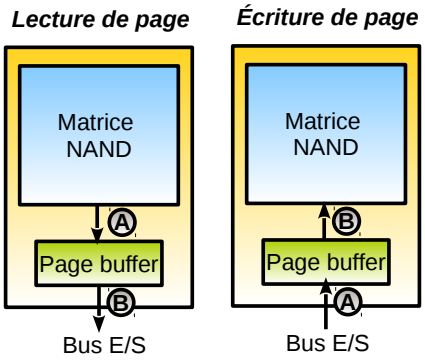


FIGURE 1.5 – Rôle du *page buffer* lors des opérations d'écriture et de lecture de pages flash.

dans ce document lorsque l'on parle de "taille de page", cela concerne la taille des données utilisateurs, indépendamment de la taille de la zone OOB.

b) Bloc

Un bloc flash contient un certain nombre (puissance de deux) de pages flash. Tout comme la taille de page ce nombre varie selon le modèle de puce. On constate en général un nombre de pages par bloc égal à 32 (*small blocks*), 64 ou 128 (*large blocks*).

c) Plan

Une puce de mémoire flash contient en général un seul ou deux plans (Grupp et coll., 2009). Certains modèles en contiennent jusqu'à 4 (Intel Corp. and Micron Inc., 2008). Une fois de plus le nombre de blocs par plan varie selon le modèle de puce NAND. Par exemple Micron Inc. (2005) présente une puce de type *small blocks* de taille 128 Mo au total et contenant un seul plan présente 8192 blocs dans cet unique plan. Un autre exemple plus récent, Grupp et coll. (2009), présente une puce de taille 1 Go, avec un plan contenant 4096 blocs.

On peut référer à l'ensemble de blocs contenus dans un plan sous la forme "matrice de transistors NAND" ou tout simplement "matrice NAND". Dans un plan, en plus de cette matrice de transistors on retrouve un registre appelé *page buffer* ou *page register* (voir figure 1.4). Ce registre a une taille égale à celle d'une page flash. Il est situé entre la matrice NAND et le bus d'E/S. Il est utilisé lors des opérations de lecture et d'écriture de pages flash (illustrées sur la figure 1.5) :

— Lors d'une lecture de page flash, le page buffer tamponne les données qui sont récupérées depuis la matrice NAND. Une fois que la page toute entière est présente dans le registre, son contenu est envoyé au système hôte via le bus d'E/S ;
— Lors d'une écriture, le page buffer tamponne les données arrivant de l'hôte sur le bus d'E/S. Une fois ce transfert effectué, la page est envoyée depuis le registre dans la matrice de transistors.

d) Notion de die (*dé*) et de LUN (*Logical Unit, unité logique*)

Certaines études et fiches techniques de modèles de puce introduisent la notion de *die* (*dé*). Un *die* est un ensemble de plans, constitué en général d'un ou deux plans (Grupp et coll., 2009). Une puce peut contenir un ou plusieurs *dies* : deux pour une puce de type *dual die package*, quatre pour le type *quad die package* (Jung et coll., 2012).

Un *die* est défini comme la plus petite unité de l'architecture hiérarchique pouvant exécuter des commandes de manière indépendante au sein d'une puce de flash NAND (Micron Inc., 2010). Une puce contenant plusieurs *dies* est parfois appelée *package*.

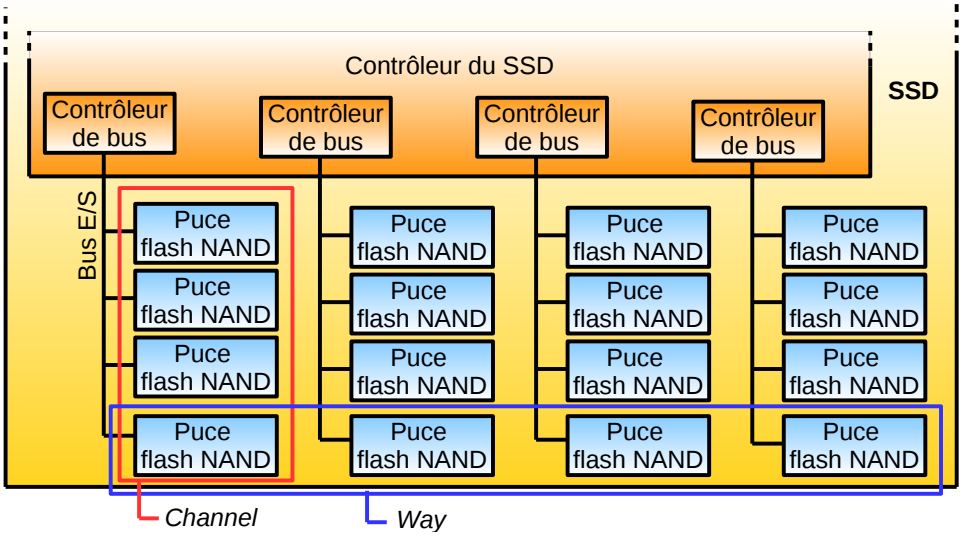


Figure 1.6 – Organisation en canaux (*channels*) et voies (*ways*) des LUNs flash au sein d'un SSD. Une voie correspond à l'ensemble des puces situées au même niveau dans les canaux contenants.

Dans la norme ONFI [2] (ONFI Workgroup, 2014), les notions de *die* et de puce sont abstraites en un seul et même concept : l'unité logique (*Logical Unit*, LUN). La LUN est ainsi définie : il s'agit d'une unité contenant un certain nombre de plans, partageant éventuellement un bus d'E/S avec d'autres LUN, et capable d'exécuter des commandes de manière indépendante par rapport aux potentielles autres LUN.

e) Structure des composants de stockage au sein d'un SSD

Les disques à base de mémoire flash (*Solid State Drives*, SSD, qui n'ont de disque que le nom car ils sont entièrement faits de composants électroniques) actuels contiennent plusieurs puces flash (ou LUN). Il en est ainsi pour deux raisons principales. Premièrement, la capacité d'une puce de mémoire flash étant limitée, les puces sont multipliées dans le SSD pour augmenter la capacité totale du SSD. Deuxièmement, l'architecture interne d'un SSD (c'est-à-dire l'organisation des puces flash et des bus les reliant au contrôleur) contenant plusieurs puces permet de paralléliser certaines opérations au sein du SSD et ainsi d'augmenter les performances du système de stockage.

Au sein d'un SSD, on retrouve des groupes de LUNs (Yoo et Park, 2011; Park et coll., 2009) reliées entre elles via des bus d'E/S appelés canaux (*channel*). Chaque canal est également relié au contrôleur du SSD. Cette architecture est illustrée sur la figure 1.6. Les canaux possèdent leur propre contrôleur, il est ainsi possible de lancer deux commandes sur deux LUNs situées chacune dans deux canaux différents de manière totalement parallèle. Il est également possible de lancer des commandes en mode dit "entrelacé" sur des LUNs situées dans le même canal.

Bien entendu le nombre de canaux et le nombre de LUNs par canal au sein d'un SSD sont très dépendants du modèle de SSD considéré. Pour exemple, Yoo et Park (2011) présentent des analyses sur la consommation énergétique de deux modèles de SSD, les auteurs rapportent 10 canaux et 2 LUNs par canal pour le SSD *Intel X25-M* (Intel Corp., 2009), et 8 canaux et 8 LUNs par canal pour le modèle *Samsung MXP* (Samsung Electronics Co., 2009).

2. Le groupe *Open Nand Flash Interface* est un groupement d'industriels fabriquant des puces NAND dont l'objectif est de normaliser les méthodes d'utilisation des puces fabriquées.

f) Résumé

La table 1.1 résume les différentes valeurs extraites de la littérature pour les différentes caractéristiques architecturales présentées dans les sous-sections précédentes. La majeure partie des chiffres présentés dans ce tableau sont tirés de Grupp et coll. (2009) et Wei et coll. (2011). La plupart des travaux scientifiques cités dans ce travail de thèse présentent des caractéristiques architecturales de puces flash se conforment à ces chiffres.

Caractéristiques architecturale	Plages de valeurs constatées dans la littérature
Taille de page flash	de 512 (+16 OOB) à 8192 (+128 OOB) octets (puissance de 2)
Nombre de pages par bloc	32 ou 64 ou 128
Nombre de blocs par plan	1024 ou 2048 ou 4096
Nombre de plans par LUN	1 ou 2 ou 4
Largeur du bus d'E/S	8 ou 16 bits

TABLE 1.1 – Plages de valeurs constatées dans la littérature pour un sous-ensemble des caractéristiques architecturales des puces flash NAND

En se basant sur ces paramètres, la capacité d'un système de stockage à base de mémoire flash se calcule simplement de manière suivante :

$$\begin{aligned}\text{taille d'une page} * \text{nombre de pages par bloc} * \text{nombre de blocs par plan} * \\ \text{nombre de plans par LUN} * \text{nombre de LUN}\end{aligned} \tag{1.1}$$

1.5 Opérations sur les mémoires flash

a) Opérations de base

Une puce flash NAND supporte un certain nombre de commandes (34 commandes sont définies dans la version 4.0 du standard ONFI (ONFI Workgroup, 2014)). Une fois de plus, le travail présenté dans cette thèse ce situe au niveau du système de stockage complet, c'est à dire à un niveau d'abstraction élevé. Ici on ne présente donc pas chacune de ces commandes en détail. On présente plutôt des opérations qui sont une abstraction de ces commandes. Cette simplification est réalisée dans tous les travaux sur la mémoire flash que l'on peut trouver dans la littérature et qui se placent au niveau du système de stockage global.

Une puce de mémoire flash NAND supporte 3 opérations principales, dites *legacy* :

— **La lecture** s'effectue au niveau de granularité d'une page. Une lecture consiste en (voir figure 1.4) :
  1. La lecture de la page dans la matrice NAND et son placement dans le page buffer du plan la contenant ;
  2. L'envoi de la page à l'hôte depuis le page buffer sur le bus d'E/S. Cela est fait octet par octet (8 bits) ou mot par mot (16 bits) selon la largeur du bus d'E/S.

— **L'écriture** s'effectue également au niveau de granularité d'une page. Une écriture consiste en (voir figure 1.4) :
  1. L'envoi par l'hôte sur le bus d'E/S de la page de données à écrire et son placement dans le page buffer ;
  2. L'écriture dans la matrice NAND.

— **L'effacement** se réalise au niveau de granularité d'un bloc complet. On efface donc toutes les pages contenues dans un bloc donné. Il n'est pas possible d'effacer une page particulière dans un bloc.

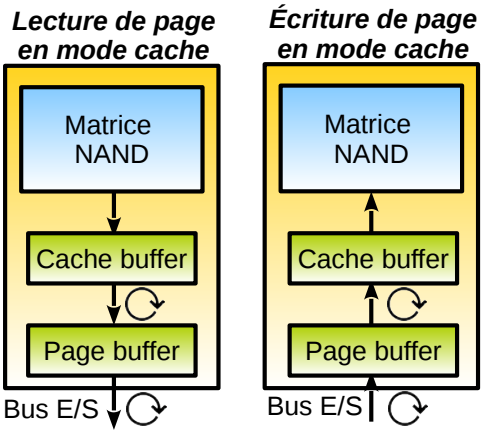


Figure 1.7 – Lecture et écriture en mode cache

Le temps d'une lecture est donc la somme du temps de transfert depuis la matrice dans le page buffer et du temps de transfert sur le bus d'E/S. Pour une écriture, son temps d'exécution est égal à la somme du temps de transfert sur le bus d'E/S avec le temps d'écriture dans la matrice. Pour ce qui est d'un effacement, c'est une opération directe mais généralement plus longue que la lecture ou l'écriture, en partie car elle met en œuvre un nombre plus important de cellules mémoires (un bloc entier).

Concernant le temps d'exécution et la consommation énergétique, les valeurs relatives à ces différentes opérations sont relativement variables. Elle dépendent de la taille de page, du nombre de pages par bloc, et plus généralement du modèle de puce NAND concerné.

A noter que certaines puces permettent l'écriture et / ou la lecture d'une partie d'une page (Samsung Electronics Co. (2007), Mohan (2010), UBI Developers (2009b)). On parle de lecture ou écriture partielle, en anglais *sub-page read* et *partial program*.

Des études ont mis en évidence le fait que dans le cas de puce de type MLC, on peut observer une variabilité dans les temps de latence (et donc la consommation énergétique) des opérations de lecture et d'écriture de pages flash réalisées sur la puce (Grupp et coll., 2009; Jung et coll., 2012). Ces variations ne sont pas indiquées sur les fiches techniques des fabricants des puces de type MLC. Elles sont dues a la micro-architecture des puces de type MLC, ainsi qu'au procédé particulier de programmation des cellules MLC (appelé *incremental step pulse programming* (Jung et coll., 2012)). Au final, on constate des variations relativement importantes sur les temps d'exécutions des opérations de lecture et d'écriture de page, variations dont l'amplitude est fonction de l'indice de la page concernée dans le bloc la contenant. Par exemple, deux puces NAND étudiées dans Jung et coll. (2012) présentent des variations en écriture allant de 250μs à 2200μs (puce 1) et de 440μs à 5000μs (puce 2). Les puces de type SLC ne sont pas affectées par ce phénomène.

b) Opérations avancées

Outre les opérations de lecture / écriture de pages et d'effacement de bloc (opérations classiques, *legacy* en anglais), il est possible d'appliquer des opérations dites *avancées* (Jung et coll., 2012; Hu et coll., 2011) au niveau (A) d'une puce et (B) d'une structure de puces et canaux comme celles contenues dans les SSD. On présente ici ces opérations de manière générale.

Opérations avancées au niveau d'une puce

— La lecture et l'écriture en *mode cache* (Micron Inc., 2004, 2006) sont supportées dans certaines puces qui contiennent un registre supplémentaire de la taille d'une page, le *cache buffer* situé entre la matrice de transistor NAND et le page buffer. Illustré sur la figure 1.7, ce registre additionnel permet de pipeliner (recouvrement temporel) le transfert d'une page à lire depuis la

matrice NAND dans le cache buffer, avec le transfert sur le bus d'E/S d'une page lue précédemment dans la matrice NAND et déplacée depuis le cache buffer dans le page buffer. L'opération inverse est également possible dans le cas d'une écriture ;
— Les opérations *multi-plans* permettent de lancer en parallèle plusieurs opérations d'un même type (lecture, écriture ou effacement) au sein de différents plans d'une même puce. Il ne s'agit pas de parallélisme pur car les plans au sein d'une puce partagent un même bus d'E/S, les transferts d'E/S sont donc entrelacés ;
— L'opération dite *copy-back* permet de déplacer les données contenues dans une page flash vers une autre page flash appartenant au même plan que la page source. Cela est fait par l'intermédiaire du page buffer, sans transfert aucun sur le bus d'E/S.

Opérations avancées au niveau d'une structure multi-puces et multi-canaux de type SSD

— Les opérations *multi-puces* et *dés-entrelacées* (*die-interleaved*) consistent à lancer plusieurs opérations de type legacy, cache ou multi-plans sur plusieurs puces d'un même canal, ou plusieurs dies d'une même puce. On peut abstraire ces opérations via la notion de LUN comme **LUN-entrelacées**. Les sous-opérations constituant une opération LUN-entrelacée peuvent être de types différents. Ces opérations sont dites entrelacées car les LUN touchées partagent le même bus d'E/S (le canal), il ne s'agit donc pas de parallélisme pur ;
— Les opérations **multi-canaux** consistent à lancer plusieurs opérations en parallèle sur plusieurs canaux différents. Il s'agit là de parallélisme pur.

Toutes ces opérations avancées ont des contraintes spécifiques relatives à leur utilisation (Jung et coll., 2012).

c) Latences et consommation des opérations flash

Les performances et la consommation d'une puce NAND sont très dépendantes du modèle de puce considéré. Pour donner une idée générale concernant ces métriques, le tableau 1.2 présente différentes valeurs pour les temps d'exécution et la consommation des trois opérations flash de base : la lecture, l'écriture et l'effacement. A noter que pour l'écriture et la lecture, ces latences correspondent au transfert depuis / vers la matrice de transistors NAND vers / depuis le page buffer au sein d'un plan. En d'autres termes ces valeurs ne contiennent pas le temps de transfert des données sur le bus d'E/S. Dans Grupp et coll. (2009), les auteurs effectuent des mesures sur différents modèles de puces, les autres valeurs sont extraites de fiches techniques. Certaines valeurs de consommation indisponibles sont notées "-". On a vu plus haut que les puces de types MLC peuvent présenter de fortes variations de latences, notamment en écriture. Pour ce type de puce les plages de valeur pour les latences sont indiquées dans le tableau lorsqu'elles sont disponibles dans l'article ou la fiche technique correspondante.

2 Contraintes et limitations relatives à l'utilisation de la mémoire flash NAND

Si la flash NAND possède de nombreuses caractéristiques qui la rendent très intéressante, un certain nombre de règles sont à respecter lors de l'utilisation de ce type de mémoire au sein d'un système de stockage. Ces limitations sont imposées d'une part par les principes physiques de fonctionnement des cellules mémoires de base, et d'autre part par la micro-architecture spécifique des puces de mémoire flash NAND. On peut classer ces contraintes en trois grandes catégories : (A) l'impossibilité de mettre à jour des données sur place ; (B) l'usure qui touche la mémoire au fur et à mesure des cycles d'écriture / effacements ; et (C) l'infiabilité des opérations de lecture et d'écriture (risque de *bit flipping*). Ces contraintes sont détaillées dans les sous-sections suivantes.

Type	Taille page (octets)	Taille bloc (Ko)	Latences (us)			Consommation (mW)				Source
			Lec.	Écr.	Eff.	Lec.	Écr.	Eff.	Repos	
SLC	4096	256	20	300	1200	19.1	56.0	25.5	13.3	Grupp et coll. (2009)*
SLC	2048	128	20	200	2000	58.8	78.4	47.6	17.0	Grupp et coll. (2009)*
SLC	2048	128	20	200	2000	41.1	59.9	35.5	7.1	Grupp et coll. (2009)*
SLC	2048	128	20	200	500	27.2	35.0	25.3	2.9	Grupp et coll. (2009)*
SLC	2048	128	20	200	1200	29.9	35.0	20.0	2.9	Grupp et coll. (2009)*
SLC	2048	128	20	200	2000	35.3	55.2	30.9	2.7	Grupp et coll. (2009)*
SLC	2048	128	25	200	2000	49.5 - 99			3.3	Liu et coll. (2010)
SLC	512	16	12	200	2000	-	-	-	-	Lim et Park (2006)
SLC	2048	256	60	800	1500	49.5 - 99			3.3	Lee et Lim (2011)
SLC	2048	128	77	252	1500	-	-	-	-	Kim et coll. (2012b)
SLC	4096	512	25	200	700	66 - 165				Lee et Kim (2013)
SLC	2048	128	25	200	1500	33 - 165			3.3	Park et coll. (2008)
MLC	4096	1024	30	300-1500	3500	75.9	94.7	70.6	8.5	Grupp et coll. (2009)*
MLC	4096	512	40	300-1500	3600	66.3	82.3	57.0	11.2	Grupp et coll. (2009)*
MLC	2048	256	20	300-1100	2800	54.0	58.9	42.4	12.7	Grupp et coll. (2009)*
MLC	4096	512	110	400-2000	2800	112.0	132.2	111.8	27.3	Grupp et coll. (2009)*
MLC	4096	512	165.6	905.8	1500	-	-	-	-	Kim et coll. (2012b)

Table 1.2 – Valeurs de latences et consommation des opérations flash de bases extraites de la littérature. *Les auteurs de Grupp et coll. (2009) ont effectué des mesures sur différents modèles de puces, tandis que les autres valeurs présentées sont extraites de fiches techniques (*datasheets*).

2.1 Contrainte A : la règle "effacement avant écriture"

La manière dont la flash NAND est construite fait qu'il est impossible d'écrire des données dans une page qui en contient déjà. Les mises à jour de données en place sont donc impossibles. Avant toute écriture dans une page contenant des données il est au préalable nécessaire d'effacer cette page. Les limitations imposées par cette contrainte sont renforcées d'une part par l'asymétrie entre la granularité de l'opération d'écriture et celle d'effacement, et d'autre part par la différence des performances (latences) relative à chacune de ces opérations.

Dans la littérature on trouve parfois référence à cette contrainte sous le nom (anglais) de *erase-before-write rule*. Il s'agit probablement de la contrainte la plus importante parmi celles relatives à la flash NAND, car elle mène à la mise en place de systèmes de gestion flash complexes au sein des système de stockage. Ces systèmes de gestion sont détaillés dans les sections suivantes.

2.2 Contrainte B : l'usure

Une cellule de mémoire flash ne peut supporter qu'un nombre limité d'effacements. Passé un certain seuil cette dernière ne peut plus être utilisée. Comme l'opération d'effacement est très liée à l'opération d'écriture (contrainte A), l'endurance se chiffre en nombre maximal de cycles d'écriture / effacement supportés. Cette limitation vient du fait que la fenêtre d'opération de la tension de seuil se réduit au fur et a mesure que la cellule est écrite et effacée, car de plus en plus d'électrons sont piégés dans la couche isolante (Richter, 2014b).

La limite concernant le nombre de cycles d'écriture / effacement dépend du type de flash NAND utilisé. Cette limite est de 100 000 cycles pour la NAND SLC, 10 000 pour la MLC à deux niveaux, et

5 000 pour la TLC (Bouganim et Bonnet, 2011).

2.3 Contrainte C : la fiabilité

Une puce de mémoire flash n'est pas fiable en elle-même. En effet, lors des opérations de lecture et d'écriture, des phénomènes d'inversement de bits (*bitflips*) peuvent être constatés sur les données stockées en flash, et également sur les données lues ou écrites en mémoire. Ces erreurs sont dues à la technologie utilisée pour réaliser la cellule de base des mémoires flash NAND et aux fortes tensions appliquées aux cellules lors des opérations de lecture, écriture et effacement (Richter, 2014b; Chen et coll., 2007).

Ce manque de fiabilité fait au final que, sans gestion particulière, des données écrites en mémoire flash et lues quelques temps plus tard ne seront pas forcément lues à l'identique. De plus, pour réduire les erreurs dues aux problèmes de fiabilité de la flash NAND, il est généralement conseillé, lors de l'écriture en flash, de programmer les pages au sein d'un même bloc de manière séquentielle. Il s'agit d'une contrainte forte pour les puces MLC, et d'une (forte) suggestion pour les puces SLC (Grupp et coll., 2009).

La rétention de données est une métrique indiquant le temps pendant lequel une cellule est capable de mémoriser son contenu depuis sa programmation. On rapporte un temps de rétention pour les cellules de mémoire flash entre 5 et 10 ans (Chen et coll., 2007). Il faut pondérer ces chiffres par le fait que la rétention d'une cellule de mémoire flash dépend de multiples paramètres, notamment le nombre d'effacement subits au moment de la programmation. Les chiffres cités précédemment supposent un nombre d'effacements égal à 0. La rétention peut baisser à 1 an pour des puces en fin de vie (Grupp et coll., 2009).

3 Systèmes de gestion des contraintes - Notions générales

Les contraintes appliquées à l'utilisation des mémoires flash de type NAND font qu'aujourd'hui tout système de stockage à base de mémoire flash intègre une série de composants logiciels et matériels contrôlant directement la mémoire flash et permettant de gérer ces contraintes. On regroupe ces composants sous le terme "couche de gestion". C'est une définition au sens large, c'est à dire qu'elle s'applique à l'intégralité des systèmes de stockage à base de mémoire flash. On peut identifier deux rôles principaux pour cette couche de gestion :

1. **Gestion des contraintes** évoquées précédemment, pour rendre utilisable le système de stockage à base de flash NAND en respectant les règles imposées par la technologie et la micro-architecture spécifiques à ce type de mémoire ;
2. **Intégration de la mémoire flash dans les systèmes informatiques**. La mémoire flash est une technologie relativement récente, et son mode opératoire est sensiblement différent des technologies de stockage secondaire plus anciennes, comme les disques durs ou le stockage optique. Il est nécessaire d'intégrer, de la manière la plus transparente possible, les systèmes de stockage à base de mémoire flash dans les systèmes informatiques actuels. C'est un des rôles de la couche de gestion qui interface le système de stockage flash avec le système informatique hôte, tout en respectant les spécificités relatives à ce type de mémoire.

Pour résumer, le rôle de la couche de gestion est d'abstraire les spécificités et contraintes de la mémoire flash pour permettre son utilisation de manière transparente dans un système informatique.

Dans cette section on présente les notions générales communes à tous les types de couches de gestion flash. Pour cela, on dépeint les mécanismes mis en œuvre pour gérer chaque contrainte.

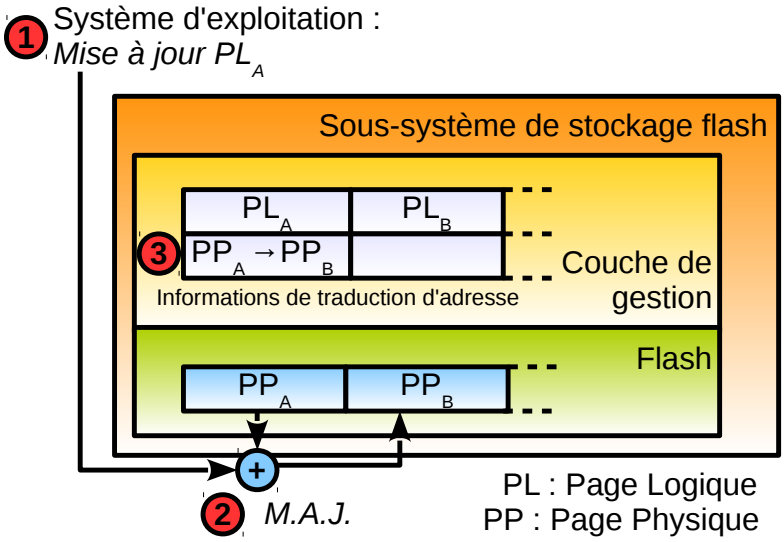


Figure 1.8 – Illustration du fonctionnement d'un mécanisme de traduction d'adresse.

3.1 Gestion de la contrainte d'effacement avant écriture

Il n'est pas possible de réaliser des mises à jour de données sur place dans une mémoire flash. Or, un système d'exploitation utilisant un système de stockage (flash ou pas) s'attend à pouvoir réaliser ce genre d'opération.

La gestion de cette contrainte se fait via la mise en place d'une traduction (*mapping*) d'adresses logiques vers des adresses physiques. Ce mécanisme fonctionne de manière suivante : l'hôte (le système d'exploitation) adresse le système de stockage flash avec des adresses logiques. La couche de gestion reçoit donc des requêtes de lecture et d'écriture de données à des adresses logiques. Grâce à la traduction d'adresse, la couche de gestion sait comment calculer l'adresse physique correspondant à une adresse logique donnée, et ainsi re-diriger une requête donnée vers l'emplacement physique (en flash) concerné. Cela permet d'effectuer des mises à jour de données dites "hors place" (*out-of-place*) en flash (adresses physiques), d'une manière totalement transparente pour le système d'exploitation (adresses logiques) qui lui pense réaliser des mises à jour en place.

a) Traduction d'adresses logiques vers physiques

Pour expliquer l'utilité de la traduction d'adresse, on prend le cas d'exemple illustré sur la figure 1.8. On considère ici le cas ou le système de stockage reçoit une requête d'écriture dans une page logique PL_A, correspondant à la page physique PP_A (point 1 sur la figure 1.8). Les contraintes flash font qu'il n'est pas possible de procéder physiquement à la mise à jour en place, c'est à dire dans PP_A. La couche de gestion va alors écrire la nouvelle version (contenant la mise à jour) des données contenues dans PP_A dans une autre page flash physique libre, PP_B (2). Cela fait, la couche de gestion met à jour les informations de traduction d'adresse indiquant que le contenu de PL_A se situe maintenant dans PP_B (3). Du point de vue du système d'exploitation, une mise à jour de données en place à été réalisée.

En continuant sur cet exemple, les données contenues dans PP_A sont obsolètes. Pour plusieurs raisons, il est peu probable que la couche de gestion efface directement le bloc flash contenant la page PP_A :

— C'est une opération très coûteuse en temps, et qui génère de l'usure sur la mémoire ;
— Il existe peut être dans le bloc qui contient PP_A d'autres pages contenant des données valides que l'on ne souhaite pas effacer. Avant de pouvoir effacer le bloc il faudrait déplacer les pages

(lecture puis écriture) contenant toujours des données valides, ce qui serait également coûteux en temps et en usure.

Pour cette raison PP_A est marquée comme *invalide* et le bloc la contenant sera effacé ultérieurement. L'introduction de l'état invalide fait qu'à tout moment, une page flash occupée peut être soit valide, soit invalide. On peut classer la totalité des pages flash contenues dans un système de stockage en 3 ensembles distincts : les pages libres, les pages valides, et les pages invalides. On peut aussi parler plus généralement de quantité d'espace libre / valide / invalide. A noter que les notions d'état valide et invalide ne sont vraies que du point de vue de la couche de gestion elle-même. Du point de vue de la mémoire flash, il n'y a que des pages libres et occupées, indépendamment de l'état valide ou invalide pour les pages occupées.

Depuis sa sortie d'usine, au fur et à mesure de l'utilisation du système de stockage la quantité de données valides et invalides augmente, alors que la quantité d'espace libre diminue. Il devient alors nécessaire de créer de l'espace libre en recyclant de l'espace invalide. C'est le rôle d'un mécanisme implémenté par tous les modèles de couches de gestion flash : le ramasse-miettes ou *garbage collector*.

b) Le ramasse-miettes

Le ramasse-miettes est exécuté lorsqu'il est nécessaire de recycler de l'espace invalide en espace libre. Les systèmes de gestion flash définissent généralement un seuil d'espace libre minimal qui, une fois atteint, déclenche le lancement du ramasse-miettes. De plus, certaines couches de gestion implémentent des mécanismes de ramasse-miettes s'exécutant en arrière plan : ces mécanismes profitent des temps morts entre les requêtes d'E/S envoyées au système de stockage pour recycler de l'espace invalide en espace libre. Néanmoins, la présence d'un ramasse-miettes en arrière plan ne permet pas de s'abstenir d'un ramasse-miettes basé sur seuil. En effet les caractéristiques non-déterministes des charges de travail soumises aux systèmes de stockage actuels font qu'il est très difficile d'estimer la quantité d'espace recyclé en arrière plan.

Le ramasse-miettes en arrière plan profitant des temps morts entre les requêtes d'E/S de la charge de travail, on peut dire que son impact sur le temps de réponse de ces requêtes est minime. Pour ce qui est du ramasse-miettes basé sur seuil, il faut savoir que son exécution est en général réalisée au cours d'une requête d'écriture. Lorsque le système de stockage reçoit une requête d'écriture, la couche de gestion vérifie si le seuil minimal d'espace libre est atteint. Si c'est le cas, avant de traiter la requête d'écriture, le ramasse-miettes est lancé. Comme expliqué dans les paragraphes suivants une opération de ramasse-miettes déclenche un certain nombre d'opérations (lectures / écritures / effacements) sur la flash et présente donc un temps d'exécution non négligeable. Ce temps d'exécution va donc impacter le temps de réponse de la requête d'écriture en cours de traitement.

Dans le travail présenté dans cette thèse, on fait référence au ramasse-miettes basé sur seuil comme ramasse-miettes *synchrone*, et au ramasse-miettes en arrière plan comme ramasse-miettes *asynchrone*. Les termes synchrone et asynchrone sont utilisés par rapport aux requêtes d'E/S de la charge de travail soumise au système de stockage.

Alors que les détails de l'implémentation du ramasse-miettes dépendent totalement du modèle de couche de gestion concerné, on peut décrire de manière simplifiée les étapes générales d'exécution de ce mécanisme. Deux exemples d'exécution du ramasse-miettes sont illustrés sur la figure 1.9.

Dans l'exemple de la figure 1.9, on considère une mémoire flash fictive contenant 6 blocs de 4 pages chacun. La mémoire contient une certaine quantité d'espaces valide / invalide / libre. Le ramasse-miettes commence par sélectionner un bloc à effacer, on l'appelle *bloc victime* (A sur la figure 1.9). Dans le cas d'exemple 1), le bloc est totalement invalide (l'intégralité des pages contenues dans le bloc sont dans l'état invalide). Il peut donc être effacé directement (B). Dans le cas où le bloc victime contient un certain nombre de pages invalides (par exemple 1, C), ces pages valides contiennent des données utilisateur qui ne doivent pas être perdues. Elles sont alors recopiées à un emplacement libre (D), et les informations de traduction d'adresses sont mises à jour. Une fois les copies terminées, le

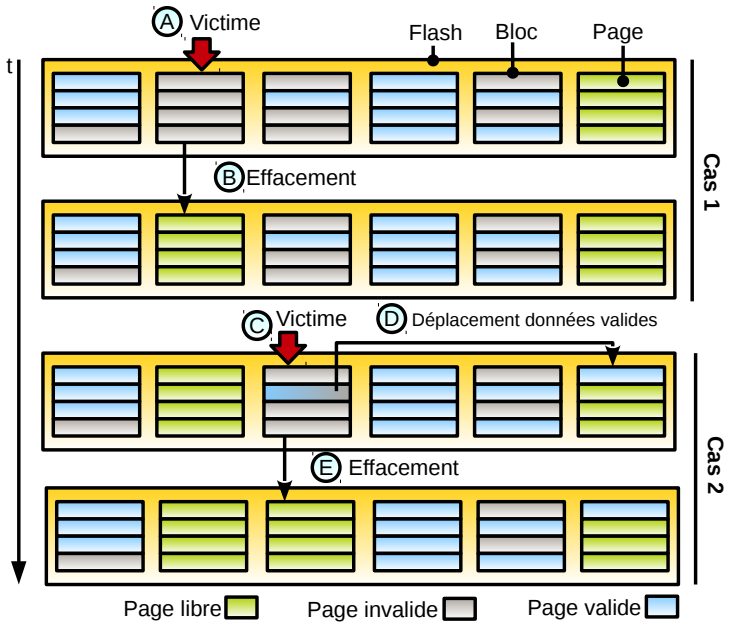


Figure 1.9 – Exécution du ramasse-miettes sur deux cas d'exemples.

bloc peut alors être effacé (E).

Le temps passé par le système de stockage à réaliser des opérations de ramasse-miettes est du temps d'auto-gestion (on parle d'*overhead*) : il s'agit de temps qui n'est pas passé à traiter des requêtes d'E/S. De plus, dans le cas du ramasse-miettes synchrone, le temps d'exécution d'une opération de ramasse-miettes va venir retarder le traitement de la requête d'écriture en cours de traitement. Les performances du ramasse-miettes sont donc critiques.

On peut voir qu'un paramètre important d'un algorithme de ramasse-miettes est le critère de sélection du bloc victime. La classe d'algorithmes de ramasse-miettes dits *gloutons* (*greedy* en anglais) (Wu et Zwaenepoel, 1994) considère uniquement le nombre de pages invalides au sein du bloc considéré, et sélectionne donc comme victime un bloc contenant une quantité importante de pages invalides. Cela est très bénéfique pour les performances car le nombre de recopies de données valides est réduit, voir nul. Néanmoins les algorithmes *greedy* ne prennent pas en compte l'usure des blocs, et leur application peut potentiellement mener à une usure prématurée de la mémoire. Les algorithmes de type coût/bénéfice (*cost/benefit*) (Kawaguchi et coll., 1995) quant à eux considèrent l'usure de la mémoire : ils calculent pour chaque bloc un ratio qui peut se simplifier de manière suivante :

$$\frac{\text{Estimation de l'usure actuelle du bloc concerné}}{\text{Nombre de pages invalides contenues dans le bloc}} \tag{1.2}$$

L'usure peut être estimée par la fréquence d'utilisation du bloc : certains systèmes considèrent le temps depuis la dernière invalidation au sein du bloc (Kwon et Koh, 2007), et d'autres le compteur d'effacement du bloc (Chiang et Chang, 1999). Ce ratio est donc utilisé pour calculer un score pour chaque bloc et déterminer le meilleur candidat à recycler.

Il existe bien d'autres implémentations de ramasse-miettes, qui sont tout à fait dépendantes de la couche de gestion au sein de laquelle elles sont intégrées.

3.2 Répartition de l'usure

La contrainte d'usure relative à la mémoire flash fait que si l'on souhaite maximiser la durée de vie des données stockées sur une puce, il est nécessaire de répartir au mieux les cycles d'écritures

/ effacements sur l'intégralité des blocs de la mémoire. En effet l'usure prématurée de certain bloc peut mener à des pertes et des inconsistances au niveau des données stockées en mémoire flash. Les couches de gestion implémentent des politiques dites de répartition de l'usure (*wear leveling*).

Les politiques de répartition de l'usure sont très dépendantes du modèle de couche de gestion utilisé. La répartition de l'usure ne correspond pas nécessairement à un composant logiciel ou matériel particulier, mais également à des règles qui influencent certaines décisions prises par la couche de gestion pour le traitement des différentes opérations dont elle est responsable. Par exemple, la classe d'algorithmes de ramasse-miettes *cost/benefit* présentée précédemment (Chiang et Chang, 1999) prend en compte l'usure des blocs comme critère de sélection du bloc victime. Il s'agit donc de répartition de l'usure. Plus généralement, les principales décisions et principaux mécanismes dans lesquels la répartition de l'usure est prise en compte sont les suivants :

— **La stratégie d'écriture** : lorsqu'il s'agit d'écrire une nouvelle page logique de données, la répartition de l'usure peut être prise en compte dans le choix de la page physique libre qui sera écrite ;
— **Le choix du bloc victime par le ramasse-miettes** : il s'agit des blocs effacés. A noter que dans certains systèmes la répartition de l'usure est découplée du ramasse-miettes ;
— **Certains mécanismes spécifiques dédiés à la répartition de l'usure** : des mécanismes dédiés sont implémentés dans certains modèles de couches de gestion. On peut par exemple citer des systèmes qui, au fil du temps, déplacent les données dites *chaudes* (fréquemment accédées) et les données *froides* (rarement accédées) (Kim et coll., 2012b). Cela est fait pour éviter un important écart de valeurs des compteurs d'effacement entre les blocs contenant des données (A) rarement accédées et (B) fréquemment accédées. En effet, la nature des données chaudes fait qu'elles sont fréquemment invalidées et l'espace contenant recyclé. A l'inverse, les données froides ont, sans gestion particulière, tendance à rester cloîtrées dans les blocs les contenant. En inversant régulièrement les emplacements flash des données chaudes et froides, on s'assure de limiter l'écart en nombre d'effacements entre les différents blocs de la mémoire flash.

Au niveau de la couche de gestion, on identifie les cellules de mémoires flash endommagées par l'usure au niveau du bloc les contenant : on parle de *bad blocks*. Malgré la répartition de l'usure, des bad blocks apparaissent au fil de la vie de la mémoire flash. Certains sont même présents à la sortie d'usine. Le rôle de la couche de gestion est également de marquer les blocks usagés comme tels, et de faire en sorte qu'ils ne soient pas utilisés pour stocker des données utilisateur (ST Microelectronics, 2004).

3.3 Gestion de la fiabilité

Ici on ne présente pas ce qui est fait au niveau micro-architectural pour gérer les problèmes de fiabilité de la flash NAND, mais au niveau système, c'est à dire au niveau de la couche de gestion.

Les problèmes de fiabilité des mémoires flash NAND se règlent essentiellement via la mise en place de codes correcteurs d'erreurs (*Error Correcting Codes*, ECC) au niveau de la couche de gestion. Ces codes correcteurs peuvent être réalisés en matériel via des circuits dédiés, où en logiciel par exemple dans le code du micro-logiciel exécuté sur le contrôleur de certains périphériques à base de mémoire flash. Dans une grande partie des cas, les codes de type *Hamming* sont utilisés (Hamming, 1986). Ces codes nécessitent le stockage de méta-données dans la zone OOB des pages de la puce flash concernée. La faible taille de la zone OOB limite le nombre d'erreurs pouvant être détectées et / ou corrigées par les codes de Hamming. Pour augmenter la robustesse des systèmes de stockage à base de flash face aux erreurs, l'utilisation de codes dits de *Reed-Solomon* (Kang et Miller, 2009) ou encore *Bose-Chaudhuri-Hocquenghem* (Macronix, 2014; Mariano, 2012), a été proposée.

Dans ce travail de thèse on ne s'intéresse pas aux problèmes de fiabilité ni à leur implication au niveau de la gestion.

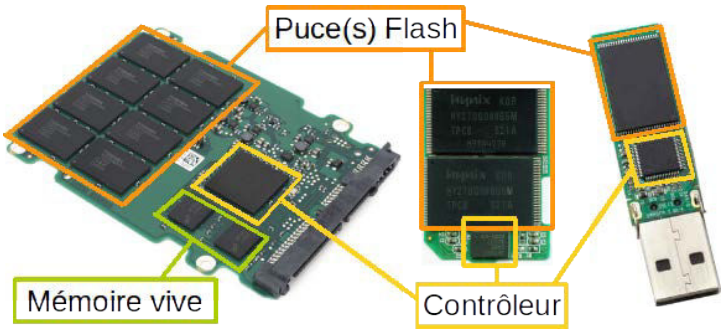


Figure 1.10 – Trois périphériques à base de FTL : l'intérieur d'un disque SSD (gauche), d'une carte SD (centre) et d'une clé de stockage USB (droite). Les positions des puces de mémoire flash NAND (orange) et du contrôleur exécutant la FTL (jaune) sont mises en évidence [4].

4 Implémentation des systèmes de gestion de contraintes : FTL & FFS

Après avoir présenté les concepts généraux implémentés par les systèmes de gestion flash pour faire face aux contraintes spécifiques imposées par ce type de mémoire, on présente dans les parties suivantes les deux grandes classes d'implémentations de systèmes de gestion flash : la couche de traduction (*Flash Translation Layer*, FTL) et les systèmes de fichiers dédiés (*Flash File Systems*, FFS). Bien entendu, les performances et la consommation du système de stockage sont très fortement impactées par le modèle d'implémentation de la couche de gestion. Dans le chapitre suivant seront présentés quelques exemples d'implémentations concrètes de FTL et FFS. Ici on présente les principes généraux de ces deux classes de couches de gestion.

4.1 Couche de traduction : *Flash Translation Layer* (FTL)

a) Présentation de la FTL

On retrouve la FTL dans une grande partie de périphériques à base de mémoire flash. Les principaux sont les clés de stockage flash USB, les cartes SD/MMC, les puces eMMC [3], et les disques SSD. Le point commun entre ces périphériques est que tous embarquent un contrôleur / micro-contrôleur / micro-processeur de complexité variable. La FTL correspond à l'ensemble des algorithmes de gestion de la flash qui s'exécutent sur ce contrôleur. La FTL est donc une couche matérielle et logicielle. La figure 1.10 illustre une vue interne de plusieurs périphériques à base de FTL, mettant en évidence les puces flash et le contrôleur exécutant la FTL.

La FTL gère les contraintes flash en effectuant une traduction d'adresses logiques vers physiques, et en mettant en œuvre des mécanismes de répartition de l'usure. L'application de codes correcteurs d'erreurs peut être réalisée par la FTL elle-même, ou déportée sur un circuit dédié.

Pour son fonctionnement, la FTL nécessite une certaine quantité de mémoire vive. Cette mémoire est utilisée pour maintenir des méta-données relatives à la FTL. Les contrôleurs de périphériques à base de FTL embarquent généralement pour ce faire de la mémoire de type SRAM (Gupta et coll., 2009). On peut également trouver de la mémoire de type DRAM, soudée par exemple sur la carte interne de périphériques de type SSD (voir figure 1.10). En plus du stockage des méta-données de la FTL, cette mémoire vive sert à tamponner les données utilisateur pour accélérer les performances du

3. Un système eMMC contient de la mémoire flash et un contrôleur exécutant une FTL, le tout au sein d'une unique puce.

4. Images adaptées de :
`http://techreport.com/review/24863/seagate-600-ssd-solid-state-drive-reviewed`,
`http://en.wikipedia.org/w/index.php?title=Secure_Digital&oldid=613106185` et
`http://www.usbcompany.co.uk/blog/the-technology-inside-a-branded-usb-stick/` (accédés 09/07/2014).

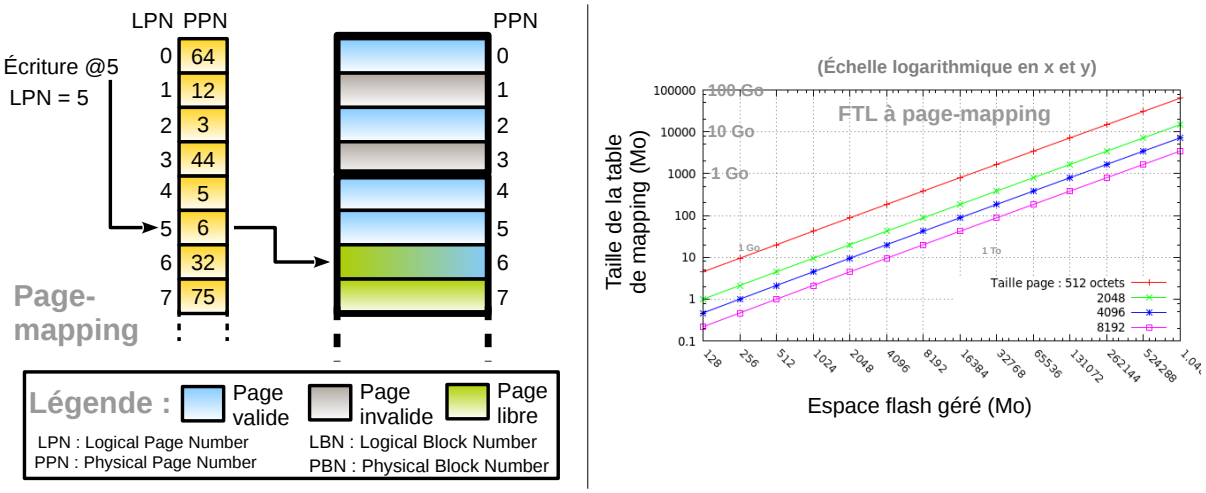


FIGURE 1.11 – Exemple de traduction par page (gauche), et évolution de la taille d'une table de traduction par page (droite), en fonction de la quantité d'espace flash géré et de la taille d'une page.

système de stockage. En comparaison avec la flash NAND, ces mémoires vives (SRAM, DRAM) sont très performantes, mais également très coûteuses (prix au bit), on tente donc généralement de minimiser leur quantité dans un périphérique. Cela est particulièrement vrai dans des périphériques produits en nombres très importants comme les clés USB et les cartes SD.

On peut classer les systèmes de FTL en trois catégories principales, en fonction de la granularité à laquelle la traduction d'adresses logiques vers physiques est effectuée. On distingue la traduction par page, la traduction par bloc et la traduction hybride (Chung et coll., 2009). A noter que les deux premiers types de traduction d'adresse sont conceptuels : en effet les limitations qu'ils présentent rendent les implémentations à base de pures traduction par page ou par bloc impossibles.

Traduction par page (*page-mapping*) La traduction par page (Ban, 1995; Chung et coll., 2009) fait correspondre à chaque page logique une page physique. La cible de la lecture et de l'écriture flash étant la page, il s'agit du niveau de granularité de plus fin. Une FTL à base de traduction par page permet à une page logique donnée d'être mise en correspondance avec *n'importe quelle page physique*. On peut faire l'analogie avec la flexibilité offerte par un cache complètement associatif (Gupta et coll., 2009).

La figure 1.11 (gauche) présente un exemple de traduction par page, réalisée sur une mémoire flash hypothétique présentant 4 pages par bloc. L'hôte demande à écrire des données dans la page logique d'indice 5. La FTL consulte la table de mapping qui définit la page logique 5 comme mappée sur la page physique 6. Dans notre exemple la page physique 6 est libre, elle est donc écrite. En cas de collision (dans le cas où la page physique 6 est occupée), la FTL peut écrire la page logique 5 dans n'importe quelle page libre et mettre à jour les informations de mapping.

Le page-mapping, de par sa flexibilité, offre les meilleures performances par comparaison aux autres types de traduction d'adresses. Néanmoins, la traduction par page implique une table de traduction (ou *table de mapping*) de taille proportionelle au nombre total de pages flash gérées. Il s'agit d'un nombre assez important, qui croit d'autant plus aujourd'hui avec l'augmentation des capacités des périphériques à base de mémoire flash. Or, la table de mapping de la FTL est située dans la mémoire vive du périphérique concerné. On peut aisément estimer la quantité de mémoire vive nécessaire pour gérer une capacité donnée de mémoire flash avec une FTL de type page-mapping. La taille d'une entrée de la table de mapping correspond au nombre de bits nécessaires pour coder un indice de page. La taille totale de la table de mapping est alors égale à la multiplication de la taille d'une entrée par le nombre total de pages. Le nombre de pages quant à lui dépend de la taille flash gérée et de la taille d'une

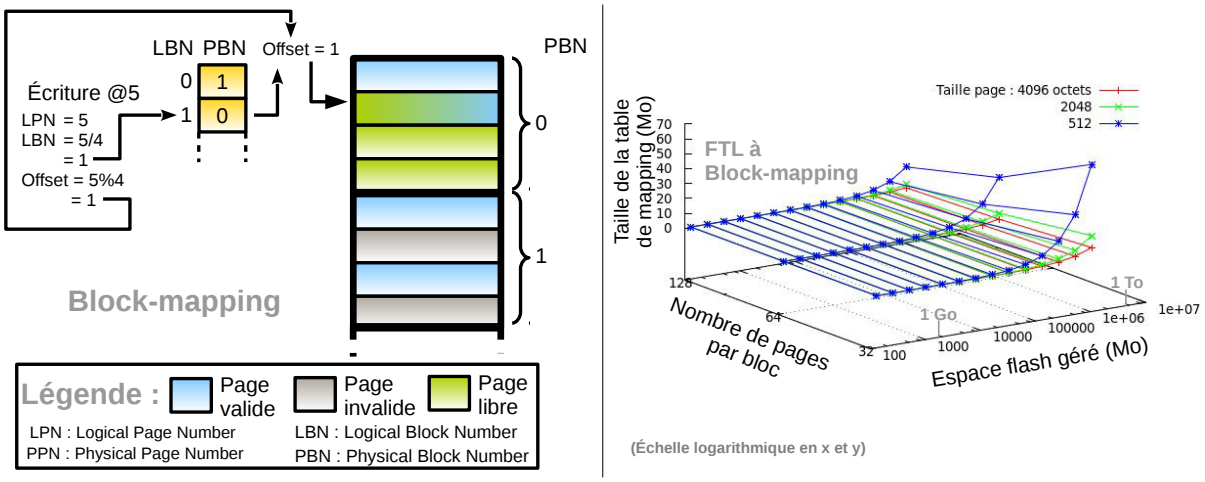


Figure 1.12 – Exemple de traduction par bloc (gauche), et évolution de la taille d'une table de traduction par bloc (droite), en fonction de la quantité d'espace flash géré, de la taille d'une page, et du nombre de pages par bloc.

page. Les équations 1.3 à 1.5 définissent le calcul du coût en mémoire vive pour la table de mapping de la traduction d'adresse par page. A Noter qu'il s'agit d'une borne minimale prenant uniquement en compte l'espace nécessaire pour stocker les indices de pages. Des méta-données supplémentaires peuvent être nécessaires selon le modèle de FTL.

$$N_{pages} = \frac{T_{flash}}{T_{page}} \quad \text{pages} \tag{1.3}$$

$$T_{entree} = \frac{\log_2 N_{pages}}{8} \quad \text{octets} \tag{1.4}$$

$$T_{table} = T_{entree} * N_{pages} = \frac{\log_2 \frac{T_{flash}}{T_{page}}}{8} * \frac{T_{flash}}{T_{page}} \quad \text{octets} \tag{1.5}$$

Dans cette série d'équations N_{pages} est le nombre total de pages de l'espace flash géré, T_{flash} correspond à l'espace flash géré en octets, T_{page} est la taille d'une page en octets, T_{entree} est la taille d'une entrée de la table de mapping et T_{table} est la taille totale de la table de traduction. L'évolution de ce coût en fonction de la taille d'une page et de la quantité d'espace flash géré par la FTL est illustré sur la figure 1.11 (droite). On peut voir que cette taille devient rapidement très importante lorsque l'espace flash géré augmente. Ce type de table de mapping ne pouvant être stocké en mémoire vive, les implémentations de FTL concrètes à base de page-mapping pur n'existent pas aujourd'hui. Cependant, comme énoncé précédemment, de par sa flexibilité la traduction par page offre les meilleures performances par rapport à tous les autres systèmes de traduction d'adresse. Le mapping par page est ainsi utilisé en évaluation de performances comme une borne supérieure de performances, pour pouvoir lui comparer d'autres algorithmes de FTL. C'est par exemple le cas dans un article présentant DFTL, une FTL à base de traduction par page stockant la table de mapping en flash (Gupta et coll., 2009).

Traduction par bloc (*block-mapping*) La traduction par bloc (Shinohara, 1999; Chung et coll., 2009) considère comme granularité du mapping un bloc. Il y a une entrée dans la table de mapping par bloc flash géré : la taille de la table est donc considérablement réduite. Néanmoins, le bloc mapping à la particularité de présenter des performances en écriture très mauvaises dues à sa limitation principale :

les données correspondant à une page logique quelconque ne peuvent être placées qu'à un seul emplacement physique donné dans le bloc la contenant.

Pour comprendre cela, on peut étudier le fonctionnement de la traduction par bloc, illustré sur la figure 1.12 (partie gauche). Lorsque l'hôte demande à écrire des données dans la page logique d'indice 5, le numéro de bloc logique est d'abord calculé. On l'obtient en divisant le numéro de page logique par le nombre de pages par bloc (1 dans notre exemple). On calcule de plus un *offset*, indice de la page logique dans le bloc logique la contenant. On l'obtient grâce au modulo entre le numéro de page logique et le nombre de pages par bloc (1 dans notre exemple). La table de mapping est consultée, dans notre cas le bloc logique 1 est mappé sur le bloc physique 0. Les données sont alors écrites dans ce bloc physique à l'offset calculé précédemment (1). L'offset d'une page dans son bloc est conservé lors de la traduction logique vers physique du mapping par bloc : l'offset logique doit être égal à l'offset physique. Cet offset détermine donc l'unique emplacement des données correspondantes à une page logique dans le bloc physique correspondant. On parle de *contrainte de conservation de l'offset* relative à la traduction par bloc.

Lors d'une collision, il est nécessaire de déplacer toutes les pages toujours valides du bloc physique concerné vers un bloc libre, et d'y écrire la nouvelle version des données. C'est une opération très coûteuse en temps : pour chaque page toujours valide, une recopie de page (une lecture suivie d'une écriture) est effectuée. Ainsi, la FTL à base de traduction d'adresse par bloc ne possède pas d'implémentation concrète car ses performances sont trop mauvaises.

La taille de la table de traduction dépend du nombre de blocs et de l'espace flash géré. Le nombre de blocs est fonction de la taille d'un bloc et de l'espace flash géré. On peut modéliser la taille de la table de traduction d'une FTL à base de traduction par bloc de manière suivante :

$$N_{blocs} = \frac{T_{flash}}{T_{bloc}} \quad \text{blocs} \tag{1.6}$$

$$T_{entree} = \frac{\log_2 N_{blocs}}{8} \quad \text{octets} \tag{1.7}$$

$$T_{table} = T_{entree} * N_{blocs} = \frac{\log_2 \frac{T_{flash}}{T_{bloc}}}{8} * \frac{T_{flash}}{T_{bloc}} \quad \text{octets} \tag{1.8}$$

Ici, N_{blocs} est le nombre total de blocs dans l'espace flash géré par la FTL, T_{flash} et T_{bloc} sont respectivement la taille de l'espace flash géré et la taille d'une page toutes deux en octets, T_{entree} est la taille d'une entrée de la table de traduction en octet et T_{table} est la taille totale de la table. Sur la partie droite de la figure 1.12, on peut voir l'évolution de la taille d'une table de mapping de traduction par bloc en fonction de la taille d'une page et du nombre de pages par bloc (le nombre de blocs pouvant être calculé à partir de la taille d'une page, du nombre de pages par bloc et de la taille totale de flash gérée). On peut voir que même pour de grands volumes flash gérés et un nombre de blocs importants (peu de pages de petites tailles par bloc), la taille de la table reste très limitée.

Traduction hybride (*hybrid-mapping*) Au vu des points forts et points faibles concernant les performances et la taille de méta-données (tables de mapping) pour les mécanismes de traduction par page et par bloc, des systèmes de FTL hybrides sont proposés (Chung et coll., 2009). Ces systèmes tentent de profiter des avantages offerts par les deux types de mapping précédemment présentés, tout en minimisant les limitations associées. On présente ici une famille importante d'algorithmes hybrides, les FTL *à base de blocs de log*.

Bien que les implémentations soient variées, on peut extraire les caractéristiques générales des FTLs à traduction hybride. Une certaine quantité d'espace flash est mappée par page, tandis que le reste de l'espace flash est mappé par bloc. L'espace mappé par page est relativement faible pour limiter la quantité de méta-données (tables de mapping) associées. Les blocs mappés par page sont généralement

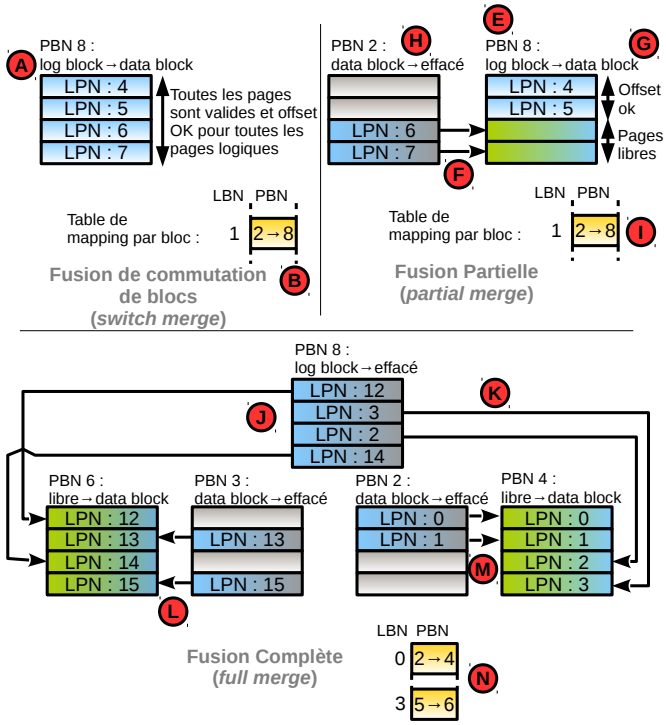


Figure 1.13 – Exemples des 3 types de fusions réalisées pendant l'exécution du ramasse-miettes du mapping hybride : la fusion de commutation de blocs (en haut à droite), la fusion partielle (en haut à gauche) et la fusion complète (en bas). Les explications relatives à ces schémas sont présentes dans le corps du texte.

nommés blocs de log (*log blocks*), et les blocs mappés par bloc sont nommés blocs de données (*data blocks*). Seul l'espace flash représenté par les blocs de données est présenté à l'hôte. Le maintient par la FTL d'un espace flash interne non visible par l'hôte (comme par exemple des blocs de log) est appelé *over-provisioning*. Le rôle des blocs de log est de recevoir les écritures (qui sont des modifications des données présentes dans les blocs de données). Le contenu de ces blocs de log est par la suite transféré dans les blocs de données.

La plupart des systèmes à base de mapping hybride font face au problème dit des *fusions*. Au fur et à mesure des écritures de données, les blocs de logs sont remplis. Lorsque l'espace libre dans les blocs de log devient critique, le ramasse-miettes est lancé. Il fusionne les blocs de log utilisés en blocs de données, et recycle des données invalides en espace libre à rajouter à l'espace des blocs de log. Comme les blocs de log sont mappés par page, n'importe quelle page logique peut se trouver dans n'importe quelle page physique au sein de cet espace. Pour ce qui est des blocs de données, la contrainte de conservation de l'offset doit être conservée (voir ci-dessus, section a) traitant de la traduction par bloc). Ainsi, le recyclage d'un bloc de log en bloc(s) de données peut se faire de 3 manières suivantes, selon le scénario : les fusions dites de commutation de blocs, partielles et complètes.

On s'appuie sur les exemples de la figure 1.13 page 27 pour expliquer les principes des 3 types de fusions. Lors d'une passe de ramasse-miettes d'une FTL hybride on commence par choisir un bloc de log victime à recycler. Le type de fusion à effectuer dépend de l'état du bloc de log victime.

— La **fusion de commutation de blocs** (en haut à droite sur la figure 1.13) peut être réalisée lorsque l'intégralité des pages logiques dans le bloc de log victime appartiennent toutes au même bloc logique, et que pour chacune de ces pages la contrainte de conservation de l'offset est satisfaite. Dans notre exemple, le bloc physique 8 (bloc de log victime) contient les pages 4 à 7 dans l'ordre, qui appartiennent au bloc logique numéro 1. Le bloc victime devient alors un bloc de données (commutation de l'état de bloc de log en celui de bloc de données, point A sur la figure 1.13), et la table de mapping par blocs est mise à jour (point B). L'ancien bloc physique

sur lequel était précédemment mappé le bloc logique 1 (le bloc physique 2 sur notre exemple) peut alors être effacé, et ajouté à la liste de blocs de log libres. La fusion de commutation de blocs est un cas idéal : en effet le coût d'une telle opération est d'un seul effacement.

— La **fusion partielle** (en haut à gauche sur la figure 1.13) est réalisée lorsque l'intégralité des pages logiques *valides* dans le bloc de log victime appartiennent toutes au même bloc logique. Dans notre exemple le bloc de log victime (bloc physique 8, point E sur la figure) contient les pages 4 et 5 dans l'ordre. Les pages 6 et 7 sont copiées depuis le bloc de données correspondant (point F, bloc physique 2 dans notre exemple). Le bloc de log victime est alors transformé en bloc de données (G), le bloc physique sur lequel était précédemment mappé le bloc logique correspondant est effacé (H), et les informations de mapping mises à jour. A noter que dans notre exemple il s'agit d'une fusion partielle idéale : la présence de pages invalides dans le bloc de log victime, la contrainte de conservation de l'offset, ou encore l'impossibilité de satisfaire la contrainte d'écriture séquentielle dans un bloc flash font qu'il faut parfois utiliser un bloc libre pour écrire les données provenant du bloc logique concerné. Le coût d'une fusion partielle est ainsi de 1 ou 2 effacements, plus un nombre relativement limité de copies de pages.

— La **fusion complète**, la plus coûteuse, est effectuée si l'on ne peut réaliser de fusion de commutation de blocs ou partielle. Dans ce cas, les pages logiques contenues dans le bloc de log victime proviennent de blocs logiques différents. La fusion s'effectue avec autant de blocs de données. Dans notre exemple (partie en bas de la figure 1.13), le bloc de log victime (bloc physique 8) contient des pages provenant des blocs logiques 0 (pages logiques 2 et 3) et 5 (pages logiques 12 et 14). Il est alors nécessaire de prendre deux blocs libres (ici les blocs physiques 6 et 4) dans lesquels on va réaliser la fusion des données provenant du bloc de log (points J et K sur la figure) et des blocs de données correspondants (dans notre exemple les blocs physiques 2 et 3, points L et M). Enfin, les deux anciens blocs de données sont effacés, ainsi que le bloc de log victime. La table de traduction est mise à jour (point N). On peut voir que la fusion complète est coûteuse car elle génère un nombre important d'opérations d'effacements, d'écritures et de lectures (recopies).

Globalement, les implémentations de FTL à base de mapping hybride tentent de maximiser le nombre de fusions de commutation de blocs et partielles, et de minimiser le nombre de fusions complètes. Cela peut se faire par exemple en adaptant la stratégie d'écriture dans les blocs de log en fonction de différentes caractéristiques de la charge d'E/S appliquée au système de stockage (par exemple le pattern d'accès séquentiel ou aléatoire). Un exemple d'implémentation notoire de FTL hybride est FAST (Lee et coll., 2007).

b) Intégration des systèmes de stockage à base de FTL dans les systèmes informatiques

La FTL émule un périphérique de type bloc. En d'autres termes, lorsqu'un périphérique à base de FTL est utilisé dans un ordinateur, l'hôte (le système d'exploitation) le considère comme un périphérique de type bloc (par exemple un disque dur), et utilise le périphérique à base de mémoire flash comme tel. Cela a deux avantages principaux :

— Les spécificités (contraintes) propres aux mémoires flash sont complètement abstraites à l'hôte par la FTL. La gestion de ces contraintes est réalisée de manière transparente par cette couche de gestion ;

— L'intégration dans les infrastructures logicielles de stockage existantes ne requiert aucun effort d'adaptation de la part de l'hôte. Par infrastructures logicielles de stockage on entend ici les différentes couches logicielles mises en œuvre dans la gestion du stockage : les systèmes de fichiers, les caches mémoire mis en place par l'OS, etc.

Les périphériques à base de FTL sont donc vus comme des disques durs. Ils sont formatés avec des systèmes de fichiers classiques (autrement dit pour disque dur) tels que Ext4, NTFS, etc.

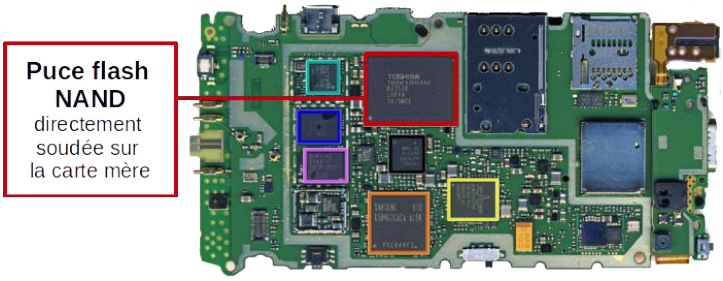


Figure 1.14 – Intérieur du smartphone Nokia N8. Une puce flash embarquée fait office de stockage secondaire [5].

4.2 Systèmes de fichiers dédiés aux mémoires flash : *Flash File Systems* (FFS)

a) Présentation des FFS

Dans le cas des systèmes de fichiers dédiés aux mémoires flash, tous les algorithmes de gestion sont implémentés au niveau du système de fichiers : ce dernier est conscient des spécificités du médium de stockage utilisé. On retrouve généralement les FFS dans les systèmes embarqués, ils sont en particulier utilisés dans des plates-formes constituées d'une carte mère embarquée sur laquelle une puce flash est directement intégrée (soudée). On parle alors de puce flash *brute* ou puce flash *embarquée*. A noter que de manière générale, le stockage secondaire de ces systèmes n'est constitué que d'une seule puce flash. Le système d'exploitation est exécuté sur le processeur et le FFS est intégré au code de l'OS. Les FFS sont donc une solution purement logicielle. Ils sont utilisés dans de nombreux systèmes embarqués comme par exemple les smartphones et tablettes actuels, ou encore les systèmes de types *Set Top Box* (décodeurs TV, routeurs ADSL, etc.). La figure 1.14 illustre la carte mère d'un smartphone contenant une puce flash embarquée.

Les trois rôles principaux d'un système de fichiers dédiés aux mémoires flash sont :

1. La gestion du stockage et des contraintes relatives à la mémoire flash ;
2. La satisfaction des contraintes propres à l'embarqué, en particulier les contraintes de ressources RAM et puissance CPU limitées, et la résistance aux démontages dits *brutaux* (démontages inopinés, souvent dus à une coupure de courant). La majorité des FFS sont, en effet, implémentés dans ce type de systèmes où la flash joue le rôle de stockage secondaire ;
3. La gestion traditionnelle du système de fichiers proprement dit : la manière dont les données sont écrites et retrouvées sur le média de stockage, organisées sous la forme d'une arborescence de fichiers et de répertoires.

Ci-dessous on présente des informations générales relatives à tous les FFS étudiés dans le cadre de ce travail de thèse.

Gestion du stockage et contraintes flash Le FFS gère la puce flash directement en prenant en compte les contraintes associées. Tout comme la FTL, il effectue des mises à jour de données hors place via un système de traduction d'adresse, et implémente des mécanismes de répartition de l'usure. Dans les systèmes à base de FFS les codes correcteurs d'erreurs peuvent être réalisés par un circuit dédié, ou en logiciel par le FFS lui-même.

Tout comme la FTL qui fait correspondre à des adresses logiques des adresses physiques, le FFS fait correspondre des portions de fichiers (vue logique) à des paquets de données écrits physiquement sur la flash (voir figure 1.15). La taille de ces paquets peut être fixe ou variable selon le FFS. Les paquets de données physiques peuvent se trouver à des emplacements variables en mémoire flash,

5. Image adaptée depuis `https://www.ifixit.com/Teardown/Nokia+N8+Teardown/3641#s17434` (accédé le 09/07/2014).

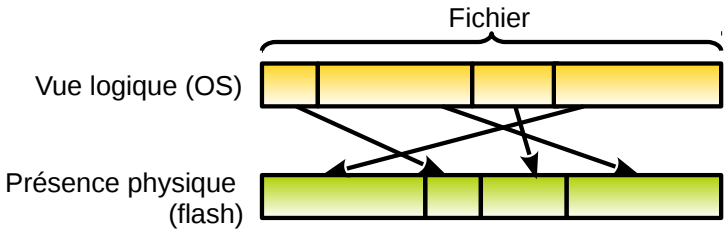


FIGURE 1.15 – La traduction d'adresse réalisée par un FFS fait correspondre des portions de fichiers (vue logique par l'OS) à des paquets de donnés physiquement écrits en flash.

la traduction d'adresse est donc nécessaire. Dans le cadre des FFS on parle plutôt d'indexation des fichiers et données pour désigner la correspondance logique vers physique. Dans le cas de l'OS Linux, un fichier est vu du point de vue logique (OS) comme une série de morceaux de données contiguës nommées des *pages*. La taille standard d'une page Linux aujourd'hui est 4 Ko. Bien entendu, il ne faut pas confondre une *page Linux* avec une *page flash*. Dans cette thèse, on prendra bien soin d'éviter la confusion en précisant, lorsque l'on parle d'une page, le type de la page (Linux ou flash).

Les techniques de répartition de l'usure sont très dépendantes du modèle de FFS et seront étudiées dans le chapitre suivant.

Contraintes embarquées et passage à l'échelle (scalabilité) des FFS Comme énoncé plus haut les FFS doivent satisfaire les contraintes propres aux systèmes embarqués, en particulier la contrainte de tolérance aux démontages brutaux, et la limitation de ressources.

On présente d'abord la contrainte de tolérance aux démontages brutaux. Il est nécessaire de monter un système de fichiers avant de pouvoir l'utiliser (par exemple au démarrage du système). Lorsque l'on a fini de l'utiliser il faut alors le démonter (par exemple à l'extinction du système). Un démontage brutal (*unclean unmount* en anglais) correspond à l'arrêt de l'exécution du système de fichiers sans lancement de la commande standard de démontage. Il s'agit typiquement d'une coupure de courant provocant l'extinction brutale du système. Ce genre d'évènement est susceptible de se produire à des instants complètement aléatoires lors de l'utilisation d'un système embarqué, souvent alimenté par une batterie. Le démontage brutal d'un système de fichiers non protégé contre ce genre d'évènement peut mener à des inconsistances des données / méta-données internes du système de fichiers, et donc mener à la perte d'une partie ou de la totalité des données. Les FFS doivent donc être tolérants aux démontages brutaux. Les techniques mises en œuvre pour assurer cette tolérance sont la journalisation, l'utilisation de structures basées sur des logs, et l'implémentation d'opérations atomiques.

Pour ce qui est de la limitation particulière de ressources propres aux systèmes embarqués (faible puissance CPU, faible quantité de RAM), les FFS se doivent de la prendre en compte. D'une part leur usage RAM doit être le plus réduit possible, sans impacter trop fortement les performances. Contrairement à la FTL qui utilise la mémoire vive embarquée dans le périphérique au sein duquel elle est implémentée, les FFS utilisent, pour stocker leur structures internes, la mémoire principale (RAM). D'autre part les algorithmes implémentés par les FFS (définissant la charge que ces FFS appliquent sur le CPU) se doivent d'être le moins demandant possible pour le processeur.

Les valeurs pour ces métriques sont bien entendu totalement dépendantes des implémentations des différents modèles de FFS. Néanmoins, on peut identifier une problématique particulièrement soulignée dans la littérature (Bityutskiy, 2005; Engel et Mertens, 2005; Park et coll., 2006b) : il s'agit du passage à l'échelle de de métriques comme la consommation mémoire (empreinte mémoire) et le temps de montage des FFS en fonction de divers paramètres, notamment la quantité d'espace flash géré (taille de partition).

Certains systèmes de fichiers voient leur consommation mémoire et temps de montage évoluer de manière linéaire avec la partition flash gérée. Comme vu précédemment la mémoire vive est une

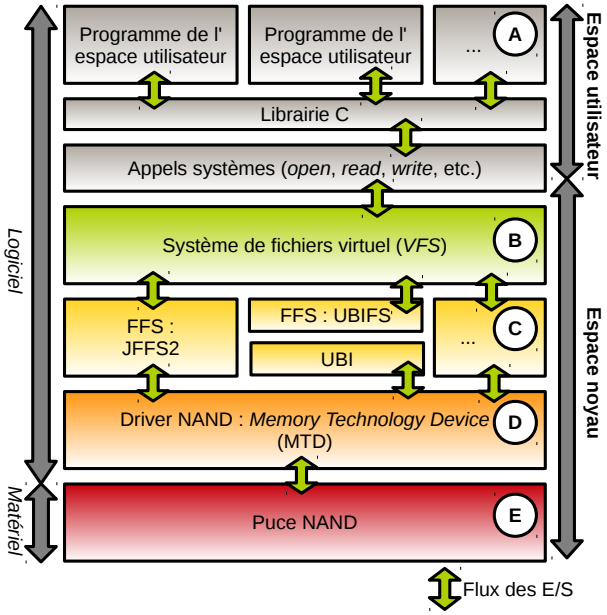


Figure 1.16 – Intégration des FFS dans la pile de couches logicielles et matérielles constituant le système de stockage à base de mémoire flash embarquée sous Linux.

ressource limitée dans les systèmes embarqués. Pour ce qui est du temps de montage, il se doit d'être le plus rapide possible dans des appareils redémarrés de manière régulière tels que les smartphones ou les tablettes. Un passage à l'échelle linéaire fait que, passée une certaine quantité d'espace flash géré, l'utilisation des FFS concernés devient impossible car (1) l'empreinte mémoire devient trop importante pour satisfaire les contraintes embarquées et (2) le temps de montage devient trop long. Certains FFS présentent au contraire un passage à l'échelle logarithmique. Cela leur permet de gérer des espaces flash beaucoup plus important.

b) Gestion du système de fichiers et intégration des systèmes de stockage de type FFS dans les systèmes informatiques

Le rôle d'un système de fichiers au sens propre est (1) de permettre le stockage de données sur un média et (2) de permettre à un utilisateur d'accéder aux données stockées. Les systèmes de fichiers organisent et présentent les données stockées sous la forme d'une arborescence de fichiers contenus dans des répertoires.

Le système d'exploitation gère différents types de systèmes de fichier. L'OS définit une série d'interfaces auxquelles doivent se conformer les systèmes de fichiers implémentés. Le rôle de ces interfaces est de permettre à l'utilisateur de manipuler les données stockées par le FS : création / suppression de fichiers / répertoires, écriture et lecture dans des fichiers, modifications des méta-données comme le nom ou les droits d'accès, etc.

Le système d'exploitation Linux supporte trois des FFS les plus populaires, JFFS2 (Woodhouse, 2001), UBIFS (Schierl et coll., 2009) et YAFFS2 (Manning, 2010). Linux est utilisé dans les travaux présentés dans cette thèse, on explique donc ci-dessous les principes de l'intégration d'un FFS dans la pile de couches logicielles et matérielles représentant le stockage à base de flash sous Linux embarqué.

Pour ces explications on s'appuie sur la figure 1.16. Les programmes de l'espace utilisateur (A sur la figure) accèdent aux fichiers via des appels systèmes tels que les primitives *open*, *read* ou *write*. Ces appels systèmes peuvent être encapsulés dans des librairies comme par exemple une librairie C. Les appels système constituent la frontière entre l'espace utilisateur et l'espace noyau. Ils sont reçus par une entité appelée le système de fichiers virtuel (*Virtual File System*, VFS, B sur la figure). On détaille

plus loin le fonctionnement de cette couche. Sous le système de fichiers virtuel, les requêtes d'E/S sont transmises au FFS (C). Le FFS traite ces requêtes et fait accès au média de stockage (flash) via un pilote de périphérique (D). Sous Linux, toutes les puces flash NAND supportées sont accédées via un pilote commun, la couche nommée *Memory Technology Device* (MTD), détaillée ci-dessous. Cette couche contrôle le matériel, c'est à dire la puce flash NAND (E).

Le système de fichiers virtuel de Linux, *VFS* VFS est la couche d'abstraction de tous les systèmes de fichiers supportés par Linux. VFS permet à différents types de système de fichiers de coexister dans le système d'exploitation en cours d'exécution. En effet, grâce à VFS, l'utilisateur est capable d'accéder de la même manière à deux fichiers situés chacun sur un média de stockage ou une partition formatés avec un système de fichiers différent : la lecture se fait toujours via l'appel système *read*, l'écriture avec *write*, etc. Comme présenté précédemment, VFS reçoit les requêtes de l'utilisateur et les transfère vers le système de fichiers concerné.

Outre ce rôle d'abstraction de différents systèmes de fichiers, on peut localiser dans la couche VFS des mécanismes d'optimisation des E/S relatives au stockage. Dans le cadre de ce travail de thèse, on s'intéresse particulièrement à trois mécanismes importants qui impactent fortement les performances et la consommation du système de stockage secondaire :

1. **Le *page cache*** : le page cache est un cache de données en mémoire vive. Ce dernier tamponne toutes les données lues et écrites dans les fichiers, pour accélérer les performances des E/S relatives au stockage secondaire. Toute donnée lue dans un fichier par un processus est lue depuis le page cache après y avoir été ramenée depuis le média de stockage. Toute donnée écrite par un processus dans un fichier est écrite dans le page cache avant d'être finalement inscrite sur le stockage secondaire ;
2. **L'algorithme *read-ahead*** : Lorsqu'un processus demande à lire une certaine quantité de données dans un fichier, le noyau Linux peut décider de lire une quantité supérieure à celle demandée pour la placer dans le page cache en prévision d'accès futurs. Ce pré-chargement anticipé est nommé read-ahead ;
3. **L'algorithme de *write-back* depuis le page cache** : ce mécanisme fait qu'une écriture de données dans un fichier par un processus est tamponnée dans le page cache pendant un certain temps. Les données ne sont pas directement écrites sur le média de stockage, elle le sont plus tard de manière asynchrone. Ce mécanisme permet d'améliorer les performances en écriture en profitant du principe de localité temporelle et en absorbant les potentielles mises à jour de données répétées dans le page cache.

Read-ahead et le write-back sont implémentés au niveau de Linux, dans la couche VFS, mais peuvent être désactivés au niveau du FFS. Les détails de l'implémentation de ces mécanismes, dans le contexte de ce travail de thèse, seront présentés dans le chapitre 3 traitant de l'exploration.

Le pilote NAND *MTD* MTD (pour *Memory Technology Device*) est une couche logicielle du noyau Linux qui centralise et abstrait l'ensemble des pilotes des puces flash NAND supportées par ce système d'exploitation. La figure 1.17 illustre les détails de la couche MTD.

MTD être vu comme une pile de couches logicielles. Les couches supérieures (point A sur la figure) représentent divers interfaces d'accès au périphérique flash : l'interface d'accès par les FFS (point B, nommée (*Application Programming Interface*, API noyau), mais aussi sous la forme de périphériques de type caractère et de type bloc. Ces deux interfaces (caractère et bloc) sont accessibles sous Linux via les périphériques virtuels */dev/mtd* et */dev/mtdbloc*. Le périphérique de type caractères peut être utilisé directement depuis l'espace utilisateur pour lire et écrire aux adresses physiques de la mémoire. Le périphérique de type bloc correspond à l'implémentation d'un algorithme de FTL basique (la correspondance par blocs) pour une utilisation via un système de fichiers "classique" (pour disques durs) (C).

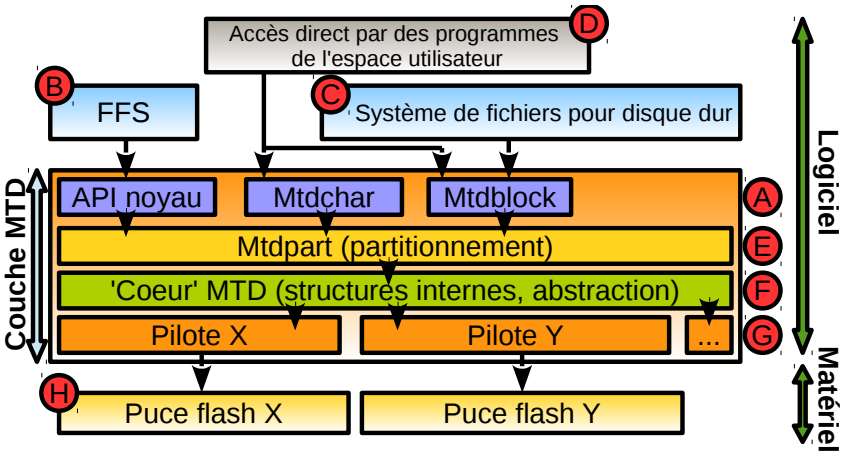


Figure 1.17 – Détails de la couche MTD, pilote générique pour les puces flash NAND supportées sous Linux

Au vu des faibles performances et de l'usure engendrée par l'utilisation d'une traduction par bloc (voir section précédente), l'utilisation de cette couche FTL est fortement déconseillée par les développeurs de la couche MTD (MTD Contributors, 2009).

MTD offre via la couche de partitionnement (E) la possibilité de partitionner une puce de mémoire flash en plusieurs volumes logiques, qui peuvent chacun être formatés séparément, par exemple avec deux FFS différents. La couche sous-jacente (F) contient des structures internes à MTD et gère l'abstraction des différents modèles de puces flash NAND supportés par Linux. Enfin, des modules pilotes, un par modèle de puce supporté, forment la couche inférieure de MTD (G). Ces pilotes gèrent les modèles de puces correspondants (H).

c) Implémentations concrètes de systèmes de fichiers dédiés aux mémoires flash dans le système d'exploitation Linux

Comme énoncé précédemment, on s'intéresse particulièrement à JFFS2, UBIFS et YAFFS2. Ils sont tous trois implémentés dans le système d'exploitation Linux, en général pour contenir le système de fichiers racine. Ils peuvent également servir à contenir des données sur une partition séparée. Ces FFS sont relativement stables et utilisés aujourd'hui dans bon nombre de systèmes embarqués en production. On présente également diverses autres propositions de FFS que l'on peut trouver dans la littérature actuelle. Tous les FFS présentés ici sont dédiés au stockage secondaire de fichiers sur puces flash brutes. A noter qu'il existe d'autres systèmes de fichiers gérant la mémoire flash comme média de stockage, qui ne correspondent pas à la définition de FFS présentée ici. On peut premièrement citer les systèmes de fichiers dédiés à être utilisés avec des périphériques à base de FTL comme *Flash Friendly File System* (F2FS) (Hwang, 2012; Brown, 2012). Deuxièmement, il existe des systèmes gérant de la mémoire flash en tant que mémoire principale, en remplacement partiel ou total de la mémoire vive (Jung et coll., 2005; Saxena et Swift, 2010; Park et coll., 2004). Il existe enfin des systèmes de fichiers gérant des architectures de média de stockage hybride contenant de la mémoire flash et des technologies plus récentes telle que la mémoire à changement de phase (*Phase Change Memory*, PCM) (Park et coll., 2008). Ces systèmes de fichiers sortent du cadre du travail présenté dans cette thèse.

A noter que le terme *système de fichiers* est utilisé soit pour nommer un modèle de système de fichiers (par exemple JFFS2), soit pour dénoter une partition formatée avec un système de fichiers donné (par exemple une partition JFFS2). On rappelle que le système d'exploitation Linux voit les données présentes dans un fichier comme une suite d'octets, de l'adresse 0 (premier octet du fichier) à l'adresse $taille\_du\_fichier - 1$ (dernier octet du fichier). Ces données sont divisées en pages Linux contiguës. Une page Linux a une taille de 4 Ko sur la plupart des systèmes. Cette division logique du fichier en

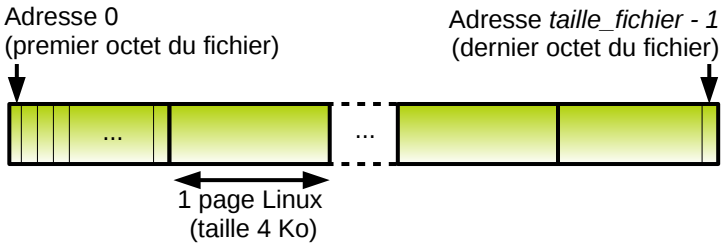


FIGURE 1.18 – Division logique, du point de vue de l'OS, des données contenues dans un fichier en pages Linux.

pages Linux est illustrée sur la figure 1.18.

JFFS2 JFFS2 (*Journaling Flash File System version 2*) (Woodhouse, 2001) est un FFS datant de 2001 (Linux 2.4.10). Sa maturité fait qu'il est toujours utilisé dans de nombreux systèmes. JFFS2, supportant les puces de types NAND et NOR, est le successeur de JFFS qui lui fonctionne uniquement avec la mémoire flash de type NOR.

JFFS2 - concepts généraux Avec JFFS2, chaque mise à jour du système de fichiers (qu'il s'agisse de l'écriture dans un fichier, la suppression d'un fichier, la création d'un répertoire, etc.) est contenue dans un paquet de données et / ou méta-données qui est écrit sur la flash. Ce paquet s'appelle une *node*. Les nodes sont écrites séquentiellement dans un bloc nommé le *bloc courant*. Il n'existe, à tout moment, qu'un seul bloc physique identifié comme bloc courant sous JFFS2. Lorsque ce bloc est plein, un nouveau bloc libre est sélectionné comme bloc courant. Un système de fichiers JFFS2 peut donc être vu comme une série de nodes (un *log*) classées par ordre temporel d'écriture, physiquement contiguës dans les blocs flash les contenants. Ces blocs flash formant le log ne sont pas forcément eux même contigus en mémoire flash physique.

Ce concept de log est à la base de l'implémentation de nombreux FFS : en effet il permet d'effectuer des mises à jour de données hors-place et ainsi de satisfaire la contrainte flash concernée. De plus, les écritures de pages sont séquentielles dans les blocs. La notion de systèmes de fichiers basé sur un log a été introduite par le système de fichiers *Sprite LFS* (Rosenblum et Ousterhout, 1992).

Sous JFFS2 chaque node est relative à un fichier (un répertoire est vu comme un fichier sans donnée). Les nodes peuvent être de deux types :

— Les nodes de données (*data nodes*) contiennent des paquets de données contigües dans le fichier (donc au niveau logique) auquel elles appartiennent ;
— Les nodes dites *dentry nodes* contiennent des métadonnées sur le fichier, et un pointeur vers le répertoire parent, des informations relatives aux droits d'accès du fichier, etc.

La taille des dentry nodes est relativement faible (quelques dizaines d'octets de méta-données plus la taille de la chaine de caractères correspondant au nom du fichier). La taille des nodes de données dépend de la taille de la requête applicative d'écriture qui a déclenché l'écriture de la node de données. Cette taille est néanmoins limitée à celle d'une page Linux, c'est à dire 4 Ko de données plus quelques dizaines d'octets de méta-données.

Au fur et à mesure des mises à jour de données dans les fichiers, de nouvelles nodes sont écrites et peuvent invalider totalement ou partiellement les anciennes (par exemple dans le cas d'écrasement de données du fichier par un programme). Chaque node contient un numéro de version qui permet de reconstruire l'ordre d'écriture des nodes et ainsi de déterminer quelle node est valide dans le cas où plusieurs nodes contiennent les données aux mêmes offsets logiques dans un fichier.

JFFS2 - indexation et problèmes de passage à l'échelle JFFS2 maintient en RAM une table de traduction effectuant la correspondance entre les pages Linux des fichiers et les nodes en flash

contenant les données / méta-données relatives à ces fichiers. Ce système d'indexation est à la base des deux principaux défauts de JFFS2 :

1. La table doit être reconstruite à chaque montage de la partition. Comme les nodes ont un emplacement variable et indéfini sur la flash, JFFS2 doit scanner la totalité de la partition flash concernée. C'est une opération (A) longue et qui plus est (B) dont le temps d'exécution évolue de manière linéaire avec la taille de la partition. Dans une étude (Engel et Mertens, 2005) les auteurs rapportent un temps de montage de 15 minutes pour une partition de taille 1 Go. A noter que les auteurs de JFFS2 proposent une fonctionnalité nommée *JFFS2 summary* (MTD Contributors, 2005) qui consiste à stocker dans chaque bloc flash un résumé des nodes contenues dans ce bloc. Ainsi au montage seules les informations de résumé sont scannées ce qui réduit le temps de l'opération de montage. On reporte (Opdenacker, 2010) par exemple une amélioration du temps de montage d'une partition de 128 Mo de 16 secondes à 0.8 secondes.
2. La taille de la table dépend du nombre de nodes présentes dans le système de fichiers : l'empreinte RAM de JFFS2 évolue de manière linéaire avec la taille du système de fichiers.

Alors que JFFS2 est très utilisé sur de petites tailles de partition (inférieures ou égales à 256 Mo), ces propriétés font qu'il supporte mal la mise à l'échelle sur des espaces flash plus grands.

JFFS2 - ramasse-miettes JFFS2 maintient également en RAM des listes chaînées d'objets qui sont des représentations des blocs flash sous-jacents. Chaque bloc est ainsi contenu dans une liste permettant de le classifier en fonction de son état. A tout moment, chaque bloc est présent dans une et une seule liste, exception faite du bloc courant. Les listes lient des blocs suivant leur état (libre, contenant des données valides et / ou invalides).

Le ramasse-miettes de JFFS2 est exécuté de deux manière différentes : (A) au cours d'une requête d'écriture, si JFFS2 détermine que l'espace libre écrit est critiquement faible, le ramasse-miettes est lancé pour recycler un bloc ; (B) via un *thread* noyau Linux, le ramasse-miettes profite des temps morts entre les requêtes d'accès au FFS pour effectuer son travail en arrière plan. Le ramasse-miettes commence par sélectionner un bloc victime. La plupart du temps il s'agit d'un bloc provenant de la liste contenant uniquement des nodes invalides. Il peut également s'agir selon le contexte d'un bloc de la liste contenant des nodes valides et invalides. Enfin, une fois sur 100 (Woodhouse, 2001), un bloc de la liste contenant uniquement des nodes totalement valides est choisi. Cela est fait pour des raisons de répartition de l'usure, pour éviter une différence de compteurs d'effacements entre les blocs contenant des données chaudes et des données froides. Lorsqu'un bloc contenant des données valides est choisit JFFS2 re-copie ces données dans le bloc courant.

JFFS2 - autres fonctionnalités JFFS2 est résistant aux démontages brutaux : en cas de coupure soudaine du courant, le scan au montage suivant est capable d'identifier les nodes erronées (les nodes en cours d'écriture lorsque la coupure de courant à eu lieu). Ces nodes sont alors marquées comme invalides, et l'état général du système de fichiers reste consistant.

JFFS2 propose une fonctionnalité intéressante qui est la compression des données à la volée : si activée, les nodes de données sont compressées avant écriture sur la flash, et décompressées avant lecture par l'application. Cela permet de réduire la charge d'E/S sur la mémoire flash, au prix d'une augmentation en besoins CPU.

Le système de fichiers JFFS2 occupe une place importante dans les phases d'exploration et de modélisation de ce travail de thèse. On donne dans le chapitre 3 les détails de l'implémentation de nombreux mécanismes de JFFS2 impactant les performances et la consommation d'un système embarqué utilisant ce système de fichiers.

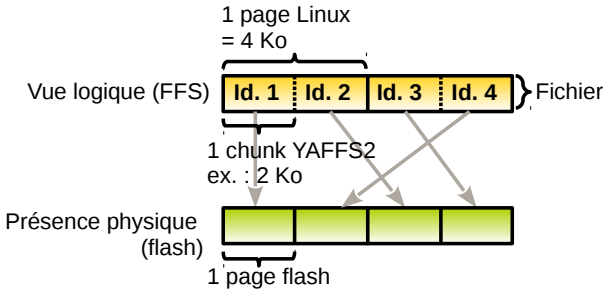


FIGURE 1.19 – Découpage d'un fichier en *chunks* par YAFFS, exemple avec une taille de page flash (et donc de chunk) de 2 Ko.

YAFFS2 YAFFS2 (*Yet Another Flash File System version 2*) (Manning, 2010; Wookey, 2007) est une évolution directe de YAFFS premier du nom. YAFFS date de 2001, et les spécifications de YAFFS2 sont datées de 2002. YAFFS est notamment utilisé aujourd'hui dans plusieurs version du système d'exploitation embarqué de Google *Android* (YAFFS2 Contributors, 2012), qui se base lui-même sur le noyau Linux. YAFFS2 supporte uniquement les puces flash de type NAND. Ce système de fichiers n'est pas officiellement contenu dans le code du noyau Linux. Il est néanmoins très simple d'intégrer YAFFS2 à Linux par le biais d'un patch fournit avec les sources du système de fichiers. Contrairement à JFFS2, YAFFS2 ne supporte pas la compression.

YAFFS2 voit toutes les entités stockées dans le système de fichiers (fichiers, répertoires, liens symboliques, etc.) comme des *objets*. Les données et métadonnées relatives aux objets sont stockées en flash via des structures de données nommées les *chunks* ("morceaux" en français). La taille d'un chunk est égale à celle d'une page flash sous-jacente, et les chunks sont toujours stockés dans des pages : il n'y a pas de chevauchement d'un chunk sur plusieurs pages. Chaque objet possède un chunk d'en-tête (*header chunk*) qui contient des métadonnées relatives à l'objet. Par exemple pour un fichier il s'agit de son nom, son répertoire parent, les droits d'accès associés, etc. De plus les objets correspondants à des fichiers possèdent également des chunks de données, contenant comme leur nom l'indique les données du fichier.

Tout chunk possède un *identifiant d'objet* liant le chunk à l'objet auquel il appartient. Chaque chunk possède de plus un *identifiant de chunk*, un entier positif ou nul. Un identifiant de chunk égal à zéro indique un chunk d'en-tête. Un identifiant positif non nul indique un chunk de données contenant les données. Cet identifiant lorsqu'il est non nul indique également la position logique des données contenues dans le chunk au sein du fichier correspondant. Prenons un exemple avec des tailles de pages flash de 2 Ko (et donc des chunks de même taille). Par exemple, le chunk d'identifiant 1 relatif à un fichier f comprend les données de f, de l'adresse 0 jusqu'à l'adresse 2047. Le chunk d'identifiant 2 contient les 2048 octets de données suivants, et ainsi de suite. Cela est illustré sur la figure 1.19.

Au fil des mises à jour du système de fichiers, les chunks sont écrits séquentiellement dans les pages au sein d'un bloc flash. Lorsque ce bloc est plein un nouveau bloc libre est choisi. On retrouve là la structure du système de fichiers sous forme de log comme c'est le cas pour JFFS2. Une mise à jour des données dans un fichier entraîne la ré-écriture complète du ou des chunks touché(s) par la mise à jour et l'invalidation des anciennes versions des chunks mis à jour. Un numéro de séquence est associé et écrit avec chaque chunk. Il s'agit d'un entier incrémenté à chaque fois qu'un nouveau bloc est choisit pour être écrit. Ce numéro de séquence, similaire au numéro de version de JFFS2, sert à deux buts distincts :

— Il permet de séparer les chunks invalides (contenant des données / méta-données écrasées) des chunks valides : par exemple si le système de fichiers contient deux chunks A et B, relatifs à un même fichier et possédant le même identifiant, celui avec le numéro de séquence le plus grand est le chunk valide, et l'autre est invalide. Si deux chunks d'un même fichier et de même identifiant ont le même numéro de séquence car ils sont dans le même bloc physique, alors

le chunk écrit à l'adresse physique la plus grande est le chunk valide, car par définition les chunks sont écrits séquentiellement dans les pages au sein d'un bloc ;

— Le numéro de séquence sert également en cas de démontage brutal, pour rejouer la séquence chronologique d'écriture des différents chunks du système de fichiers et restaurer un état stable.

YAFFS2 maintient en RAM une table faisant correspondre les adresses logiques dans les fichiers aux chunks sur la flash. Tout comme JFFS2 cette table est construite au montage via un scan complet de la partition flash. YAFFS2 présente donc un temps de montage et une empreinte mémoire qui évoluent de manière linéaire avec la quantité d'espace flash géré et la taille du système de fichiers. Néanmoins, YAFFS2 possède une fonctionnalité nommée le *checkpointing* ("point de contrôle") dont le rôle est proche de la fonctionnalité de *summary* de JFFS2. Lorsqu'un système de fichiers YAFFS2 est démonté, la table de correspondance et diverses autres méta-données relatives au système de fichiers sont directement stockées sur la mémoire flash dans des blocs. Ces blocs sont alors marqués comme contenant des données de *checkpoint*, et au prochain montage YAFFS2 lira uniquement ces données contenues dans ces blocs spéciaux pour reconstruire en RAM la table de correspondance.

Pour ce qui est de la répartition de l'usure, YAFFS2 n'implémente pas de mécanisme spécifiquement dédié à cette opération. Néanmoins, les auteurs de ce FFS indiquent que par définition la structure du FFS sous forme de log évite l'écriture répétée sur un ensemble de blocs restreints (Manning, 2010). On peut cependant arguer que le fait de ne pas séparer les données chaudes des données froides peut, indépendamment de la structuration en log, amener à une mauvaise répartition de l'usure sous des conditions particulières.

Le ramasse-miettes de YAFFS2 recycle les blocs contenant des chunks invalidés. Il est lancé à deux occasions :

1. Lors d'une mise à jour de données, si le bloc contenant les anciennes versions des chunks invalidés est complètement invalide (autrement dit s'il ne contient que des chunks invalides), alors ce bloc est effacé directement ;
2. Lorsque la quantité d'espace libre devient faible, YAFFS2 sélectionne un bloc avec une importante quantité de chunks invalides, copie les éventuelles données valides à un autre emplacement, et efface le bloc victime.

Ce type de ramasse-miettes est lancé pendant les requêtes d'E/S, l'overhead généré par le recyclage va donc augmenter d'autant les temps de traitement des requêtes. En 2010 a été introduit dans YAFFS2 une fonctionnalité de ramasse-miettes en arrière plan similaire à celle présentée dans JFFS2. Il s'agit d'un thread noyau qui profite des temps morts entre les requêtes d'E/S pour effectuer des opérations de ramasse-miettes.

UBI et UBIFS UBIFS (Hunter, 2008; Schierl et coll., 2009) (*Unsorted Block Images File System*) est relativement récent, datant de 2008 (Linux 2.6.27). Son implémentation adresse certaines limitations présentes dans JFFS2 et YAFFS2, son usage est donc en évolution croissante aujourd'hui.

Contrairement à JFFS2 et YAFFS2 qui s'appuient directement sur MTD pour accéder à la mémoire flash, UBIFS n'accède pas à la flash directement. UBIFS utilise une couche d'abstraction nommée UBI (*Unsorted Block Images*), présentée dans Gleixner et coll. (2006), qui manipule la mémoire directement via MTD. Cela est illustré sur la figure 1.16 page 31.

UBI est un gestionnaire de volume logique, c'est à dire que qu'UBI formate et gère la flash directement, tout en présentant aux couches supérieures (le FFS, dans notre cas UBIFS) un volume dit virtuel ou logique. Le rôle principal d'UBI est de gérer en partie les contraintes flash et ainsi de décharger le FFS de certaines responsabilités relatives à la gestion de la mémoire flash. UBI effectue une correspondance entre des blocs logiques, présentés au FFS, vers les blocs physiques de la mémoire flash gérée. Les blocs logiques présentés au FFS ont la même taille que les blocs flash sous-jacents. Il ne s'agit cependant que d'une simple correspondance à la granularité d'un bloc, ce qui signifie que le FFS doit

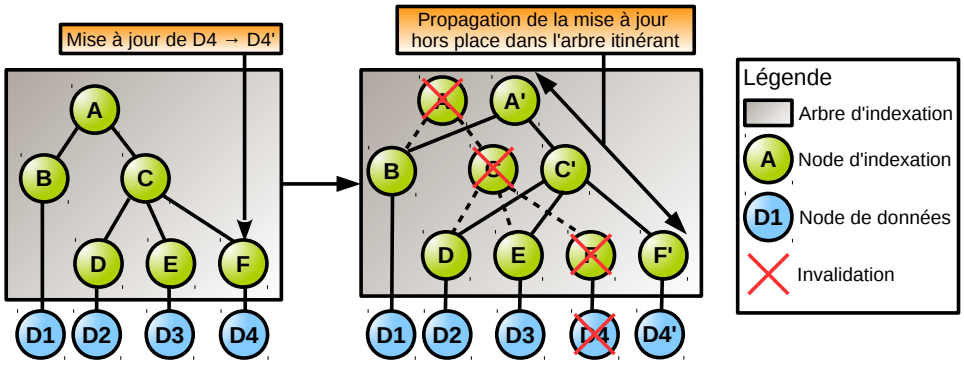


FIGURE 1.20 – Propagation d'une mise à jour de données dans l'arbre d'indexation de UBIFS. La mise à jour de la donnée D4 en D4' entraîne la mise à jour de toute la branche d'indexation dont la feuille pointe sur D4, et ce jusqu'à la racine. De part les contraintes flash les mises à jour de données en place sont impossibles, de nouvelles versions de chaque node (données et indexation) sont écrites en flash.

toujours effectuer des écritures séquentielles dans les blocs logiques, et qu'il ne peut y effectuer de mises à jour de données en place (UBI Developers, 2009a). UBI gère la répartition de l'usure via cette correspondance : les blocs logiques sont alloués à la demande, lorsque le FFS demande un nouveau bloc libre pour y écrire des données. UBI alloue les blocs en fonction des compteurs d'effacements. Il existe de plus un thread noyau relatif à UBI, qui déplace régulièrement les données froides, toujours pour des raisons de répartition de l'usure. Il est possible de créer plusieurs partitions pour plusieurs systèmes de fichiers sur un volume UBI, comme on pourrait le faire par exemple dans le cas d'un disque dur. La répartition de l'usure étant faite au niveau du volume UBI tout entier, elle est plus efficace que lorsqu'elle est réalisée au niveau local d'une seule partition, comme cela peut être le cas avec les autres FFS. En effet plus l'espace considéré par le mécanisme de répartition de l'usure est important, plus son travail est efficace. On oppose ainsi les termes de répartition de l'usure *locale* et *globale* (Homma, 2009). UBI gère également les blocs usagés, qui sont marqués comme tels lorsque le nombre d'erreurs d'écritures / de lecture dans ces blocs devient trop important.

UBIFS, travaillant sur UBI, est déchargé des responsabilités de répartition de l'usure et de gestion des blocs usagés. Tout comme JFFS2, UBIFS utilise la notion de node pour nommer un paquet de données et / ou méta-données stockées sur la flash. On a vu avec JFFS2 et YAFFS2 que l'indexation des nodes (ou chunks) dans une structure de type table mène à un temps de montage et une empreinte mémoire évoluant de manière linéaire avec la quantité d'espace flash géré et la taille du système de fichier. Ainsi ces systèmes de fichiers deviennent inutilisables lorsque l'espace flash géré dépasse un certain seuil.

Pour palier ce problème UBIFS utilise un arbre pour indexer les nodes de données. Il s'agit d'un arbre B+ (Havasi, 2011) contenant des nodes d'indexation, et dont les feuilles pointent vers les nodes de données contenant les données des fichiers. Pour éviter un scan coûteux au montage, l'arbre d'indexation est stocké en flash. Les nodes de *données* / *méta-données* et les nodes d'*indexation* contenant les informations relatives à l'arbre sont donc inscrites en flash. Le stockage de l'arbre en mémoire flash fait que, lorsqu'une partie des données d'un fichier est mise à jour, l'arbre d'indexation doit être lui aussi mis à jour. De part les contraintes flash il n'est pas possible d'écraser des données dans une node d'indexation, il faut donc écrire une nouvelle version de la node d'indexation (mise à jour hors place). Par conséquent la node d'indexation parente, si elle existe, doit elle aussi être mise à jour hors-place, et ainsi de suite jusqu'à la racine de l'arbre d'indexation. La propagation des mises à jour de données hors place dans l'arbre d'indexation est illustrée sur la figure 1.20. Les branches de l'arbre se déplaçant ainsi sur la flash, les auteurs de UBIFS appellent ce type d'arbre un arbre itinérant (*wandering tree*). En pratique, UBIFS tamponne les mises à jour de la structure d'indexation dans une image de l'arbre en RAM pour éviter des écritures flash trop fréquentes.

De part l'utilisation d'un arbre pour la structure d'indexation, le temps de montage et l'empreinte mémoire de UBIFS évoluent de manière logarithmique avec la taille de l'espace flash et du système de fichiers géré. A noter que le temps de montage de UBI évolue lui de manière linéaire.

Le ramasse-miettes de UBIFS est lancé lorsque le taux d'espace flash libre devient faible. Le ramasse-miettes sélectionne un bloc victime basé sur les informations contenues dans l'arbre de propriétés des blocs, notamment les compteurs d'effacements et les taux d'espace invalide. Les éventuelles données toujours valides contenues dans le bloc victime sont alors recopiées à un nouvel emplacement, et le bloc est effacé.

Contrairement à JFFS2 et YAFFS2, UBIFS supporte la fonctionalité de *write-back* dans le page cache (présentée plus haut dans la partie traitant de VFS), ce qui améliore les performances mais rend le système de fichiers vulnérable aux pertes de données dues aux démontages brutaux. Il est néanmoins possible de monter un système de fichiers UBIFS en désactivant le support du *write-back* (c'est à dire en mode d'écriture synchrone), via l'option de la commande de montage *mount* de Linux *sync*. Tout comme JFFS2, UBIFS supporte la compression à la volée des données contenues dans les nodes de données.

Le tableau 1.3 présente un résumé comparatif des caractéristiques des différents FFS présentés ici.

Caractéristique	**JFFS2**	**YAFFS2**	**UBIFS**
Type(s) de mémoire flash supportés	NOR, NAND	NAND	NOR, NAND
Type de périphérique (virtuel) utilisé	MTD	MTD	UBI (sur MTD)
Structure d'indexation des fichiers	Table	Table	Arbre itinérant
Algorithmes de compression supportés	LZO, Zlib, Rtime	Aucun	LZO, Zlib
Évolution du temps de montage	Linéaire	Linéaire	Linéaire (UBI)
Évolution de l'empreinte mémoire	Linéaire	Linéaire	Logarithmique
Intégration officielle dans le noyau Linux	Oui	Non (patch)	Oui

Table 1.3 – Comparatif des trois principaux systèmes de fichiers dédiés aux mémoires flash supportés sous Linux

Autres propositions de FFS On retrouve dans la littérature scientifique actuelle de multiples propositions de systèmes de fichiers dédiés aux mémoires flash. On présente ici brièvement les caractéristiques principales des FFS suivants : *the Journaling Flash File System* (JFFS) premier du nom (Axis Communications, 1999), *the Enhanced NAND Flash memory File System* (ENFFiS) (Park et Kim, 2013), FlashLight (Kim et Kim, 2012), *the Flexible flash File System* (FlexFS) (Lee et coll., 2009a; Lee et Kim, 2013), LogFS (Engel et Mertens, 2005), *the Scalable Flash File System ScaleFFS* (Jung et coll., 2008a), *the Microsoft Flash File System* (Barrett et coll., 1995), *Data Node Encrypted File System* (DNEFS) (Reardon et coll., 2012), *the Core Flash File System* CFFS (Lim et Park, 2006), *the Multimedia NAND Flash File System* (MNFS) (Kim et coll., 2009a), *the Reliable Compressing Flash File System* (RCFFS) (Kang et Miller, 2009), ainsi qu'un autre FFS sans nom particulier : Park et coll. (2006b).

JFFS1 (Axis Communications, 1999) et *the Microsoft Flash File System* (autrement nommé FFS2) (Barrett et coll., 1995) sont d'anciens systèmes de fichiers dédiés aux mémoires flash datant des années 1990. Ils sont uniquement compatibles avec la mémoire flash NOR, et présentent de sérieuses limitations. JFFS1 voit la mémoire flash toute entière comme un log séquentiel dans lequel il écrit des nodes représentant les mises à jour du système de fichiers. Le défaut principal de JFFS1 est le ramasse-miettes qui efface les blocs flash en queue de log. Les données toujours valides sont déplacées en tête de log. Alors que ce système réalise, en théorie, une répartition de l'usure quasi parfaite, cela doit être pondéré par la charge en écriture générée par le déplacement de quantités potentiellement importantes de données toujours valides. L'impact sur les performances est également non négligeable. FFS2, quant

à lui, voit la flash (NOR) comme un volume dont chaque octet est inscriptible une fois (on rappelle que la flash NOR est adressable au niveau d'un octet). L'effacement se fait au niveau de la mémoire toute entière et correspond donc à un formatage. FFS2 utilise un liste chaînée pour indexer les fichiers, et à chaque mise à jour une partie de cette liste doit être parcourue, ce qui mène à de mauvaises performances en écriture. On rapporte une chute de performance linéaire par rapport à la taille du fichier écrit (Douglis et coll., 1996).

Le temps de montage est une problématique adressée dans plusieurs propositions de FFS. Le but est d'éviter une augmentation linéaire de ce temps avec la taille de l'espace flash géré et la taille du système de fichiers lui-même. Pour adresser ce problème une proposition que l'on retrouve fréquemment est le maintient d'une zone particulière en flash contenant les informations nécessaires à l'opération de montage. Cette zone, de taille réduite, est scannée au montage, évitant une lecture de l'intégralité de la partition. Certains systèmes utilisent des blocs dédiés (Park et Kim, 2013; Lim et Park, 2006; Park et coll., 2006b), d'autres utilisent la zone OOB de la première page d'un sous-ensemble de blocs, par exemple ScaleFFS (Jung et coll., 2008a) ou MNFS (Kim et coll., 2009a).

L'effort est également mis sur les structures d'indexation des fichiers, et le stockage de ces structures en mémoire vive qui détermine l'empreinte mémoire du FFS. Pour éviter l'augmentation linéaire des structures d'indexation, LogFS (Engel et Mertens, 2005) propose tout comme UBIFS d'utiliser un arbre plutôt qu'une table. Certains systèmes comme ScaleFFS proposent de stocker un maximum d'informations d'indexation en flash tout en gardant le minimum en RAM. MNFS allège l'empreinte RAM des structures d'indexation en utilisant un mécanisme proche des techniques de mapping hybrides utilisées par les systèmes de FTL. FlashLight (Kim et Kim, 2012) améliore les performances des opérations d'indexation en stockant les données d'indexation dans les mêmes pages que les méta-données (nom du fichier, etc.) des objets qu'elles indexent.

Certains systèmes proposent une optimisation des coûts dus au ramasse-miettes. FlashLight maintient des informations sur la validité et l'invalidité des pages flash gérées. Ces informations font que FlashLight n'a pas à lire les données contenues dans les pages du bloc victime pour déterminer la validité ou l'invalidité de la page, contrairement à des systèmes comme JFFS2. Pour limiter le coût du ramasse-miettes, CFFS effectue une séparation des données chaudes et froides. Les méta-données relatives aux fichiers sont écrites dans des blocs dédiés. Les méta-données sont considérées comme des données chaudes car mises à jour plus fréquemment que les données elles-mêmes. Ces dernières sont donc considérées comme des données froides et écrites dans des blocs distincts. Lors de son exécution le ramasse-miettes sélectionne donc en majorité des blocs de méta-données car ils contiennent potentiellement une quantité importante de pages invalides.

Il existe également d'autres systèmes de fichiers proposés pour répondre à des besoins dans des contextes particuliers et spécifiques. On peut citer, entre autre, FlexFFS (Lee et coll., 2009a; Lee et Kim, 2013) qui opère sur les mémoires flash de type MLC et exploite leur capacité à être programmées comme des puces SLC (Roohparvar, 2008). Programmées comme des cellules SLC, les cellules MLC ne stockent qu'un bit mais peuvent offrir des performances similaires aux SLC. FlexFFS partitionne l'espace flash en un espace SLC présentant de bonne performances et une faible capacité de stockage, et un espace MLC (performances inférieures et large capacité de stockage). Le partitionnement est dynamique et peut être adapté selon les besoins applicatifs. Dans le contexte de la sécurité informatique, DNEFS (Reardon et coll., 2012) est une amélioration de UBIFS qui propose des fonctionnalités de suppression sécurisée des données. Une suppression sécurisée est réalisée lorsque les données supprimées par l'utilisateur ne sont absolument plus récupérables sur le média de stockage, en particulier via des outils d'analyse spécialisés. Cette fonctionnalité est implémentée par DNEFS en cryptant chaque node avec une clé unique. Lors de la suppression des données la clé est détruite, ce qui évite l'effacement des données elles-mêmes et l'impact sur les performances associé à cette coûteuse opération.

5 Conclusion

Dans ce chapitre, on a d'abord présenté les caractéristiques générales des mémoires flash. Le sous-type de mémoire flash *NAND*, dédié au stockage de données, est celui auquel on s'intéresse dans ce travail de thèse. On a présenté la structure et les opérations supportées par les puces de mémoire flash NAND.

L'utilisation de ce type de mémoire flash impose de satisfaire des contraintes qui sont l'impossibilité de mettre à jour des données sur place (contrainte A), l'usure de la mémoire au fil des écritures (contrainte B), et le manque de fiabilité (contrainte C). Les systèmes de stockage à base de mémoires flash NAND contiennent une couche de gestion pour satisfaire ces contraintes. La couche de gestion met en place une traduction d'adresses logiques vers physiques pour répondre à la contrainte A, des politiques de répartition de l'usure (contrainte B) et implémente des mécanismes de codes correcteurs d'erreurs (contrainte C).

Il existe deux classes de couches de gestions : La FTL (*Flash Translation Layer*) et les FFS (*Flash File Systems*).

La FTL est une couche logicielle et matérielle implémentée dans le contrôleur des périphériques de stockage à base de mémoire flash. Cette couche émule un disque dur et masque les spécificités / contraintes de la flash au système informatique hôte. On a vu que les différents modèles de FTL peuvent se classifier selon la granularité à laquelle la traduction d'adresse est effectuée : par page, par bloc, ou hybride. Chaque type de traduction d'adresse présente des avantages et inconvénients.

Les FFS représentent une solution purement logicielle : ils sont intégrés dans le code des systèmes d'exploitation de plates-formes embarqués, équipées de puces flash brutes. On a vu que les FFS, en plus du rôle de gestion des contraintes flash, doivent également jouer le rôle de systèmes de fichiers. Les FFS doivent de plus satisfaire les contraintes imposées par leur environnement d'exécution embarqué.

Une large partie des connaissances présentées dans ce chapitre ont été compilées sous la forme d'un chapitre du livre *Embedded Computing Systems : Applications, Optimization, and Advanced Design* (Olivier et coll., 2013c).

Dans le chapitre suivant, on effectue un état de l'art concernant plusieurs thèmes abordés dans ce travail de thèse : le benchmarking de systèmes de stockage, l'exploration des performances et de la consommation des systèmes intégrant de la mémoire flash, les modèles de performances et de consommation relatifs au stockage flash, et les simulateurs pour mémoires flash.

CHAPITRE 2

Exploration, modélisation et simulation des performances et de la consommation des systèmes de stockages à base de mémoire flash : État de l'art

Dans ce chapitre on effectue un état de l'art sur les différentes thématiques abordées dans ce travail de thèse. Ces thématiques sont liées aux différentes étapes de la méthodologie de travail générale présentée en introduction. On rappelle que ces étapes sont dans l'ordre : les phases d'exploration, de modélisation, d'implémentation et de validation via la simulation.

Au niveau de la phase d'exploration, on détermine les opérations et paramètres ayant un impact sur les performances et la consommation du système de stockage à base de mémoire flash. Une grande partie de ce travail consiste à explorer le comportement du système soumis à une charge de travail (workload en anglais) via des mesures. Il est important de définir avec précision les caractéristiques de la charge de travail, car ces dernières ont un impact considérable sur les métriques mesurées. On retrouve la définition de ces charges de travail et de leurs caractéristiques dans les études traitant du benchmarking de systèmes de stockage. C'est le sujet de la première section de ce chapitre. Dans une seconde section, on s'intéresse à l'exploration et aux mesures des différentes métriques de performances et de consommation des systèmes de stockage à base de mémoire flash.

Durant la phase de modélisation, les opérations déterminées comme impactant les performances et la consommation sont caractérisées de manière formelle. Il existe dans la littérature de nombreuses études présentant des modèles de performances et de consommation pour les systèmes de stockage à base de mémoire flash. Ils sont présentés dans la troisième section de ce chapitre.

La phase de validation consiste à raffiner les modèles et à s'assurer de leur fonctionnement correct et représentatif de la réalité. Une nouvelle fois les benchmarks sont utiles en phase de validation pour vérifier la concordance des sorties des modèles avec les comportements de systèmes réels soumis à des charges de travail représentatives de la réalité. Les modèles peuvent alors être implémentés dans des programmes, des simulateurs, et êtres concrètement utilisés pour de multiples buts. En particulier, ces simulateurs permettent l'estimation des métriques de performances et de consommation des systèmes de stockages modélisés. On présente en quatrième section un état de l'art des simulateurs de systèmes de stockage à base de mémoires flash.

Enfin, dans une dernière section, on expose le positionnement du travail présenté dans cette thèse par rapport aux multiples travaux référencés dans cet état de l'art.

Sommaire

1 Benchmarking des systèmes de stockage à base de mémoire flash

Un benchmark est une série de tests appliqués à un système pour en mesurer de multiples métriques de performances. C'est, du point de vue de l'analyse de performances, un point de référence qui peut servir à évaluer les performances de ce système, et à comparer les performances de plusieurs systèmes de même type, comme expliqué dans Jain (2008). Des benchmarks sont utilisés lors de plusieurs phases du travail présenté dans cette thèse. Dans cette section on effectue un état de l'art sur les benchmarks traitant des systèmes de stockage. Les métriques de performances (ainsi que les métriques de consommation) seront-elles présentées dans la section suivante traitant de l'exploration.

La majeure partie de la phase d'exploration consiste à mesurer des métriques de performances et de consommation de systèmes de stockages en environnement réel soumis à une charge de travail donnée. On utilise lors de cette phase une charge de travail produite par des benchmarks. Les caractéristiques de cette charge sont connues et définies dans les détails pour assurer une compréhension et une interprétation correcte des mesures réalisées. De plus, il n'est pas rare de devoir en exploration lancer une expérience de multiples fois pour obtenir des résultats représentatifs et / ou stables. Par définition la charge de travail d'un benchmark doit donc pouvoir être reproductible.

En phase de validation, on souhaite pouvoir vérifier le bon fonctionnement des modèles en comparant leurs estimations avec le comportement de systèmes réels auxquels on fait subir des charges de travail, elles-mêmes représentatives de la réalité. C'est le but de certains benchmarks de reproduire de manière synthétique les comportements d'applications réelles.

Il existe différents types de benchmarks ciblant différents éléments des systèmes informatiques. Ici on s'intéresse aux benchmarks ciblant les performances des systèmes de stockages.

Dans Traeger et coll. (2008), les auteurs effectuent un examen des techniques de benchmarking pour les systèmes de stockage, en particulier les systèmes de fichiers. Dans cette étude sont notamment passés en revue un nombre important de benchmarks de stockage utilisés dans de nombreux articles scientifiques. Selon les auteurs les benchmarks peuvent être classifiés en trois catégories principales :

1. *Les macro-benchmarks* sont des tests composés de multiples opérations de types variés. Ces tests sont censés être représentatifs d'une charge applicative réelle, provenant d'un contexte particulier de l'utilisation d'un système informatique. Par exemple, certains macro-benchmarks produisent des charges de travail censées représenter les accès faits par une base de données commerciales à large échelle, de type *transactionnelle en ligne*. Ce type de benchmark est utile pour obtenir une idée générale des performances d'un système en production ;
2. *Les micro-benchmarks* sont des tests appliquant un petit nombre d'opérations, en général une ou deux, sur le système de stockage. Le but est là d'étudier à une granularité fine l'impact de ces opérations spécifiques sur les performances du système, par exemple à des fins de caractérisation et / ou d'optimisation ;
3. *Les traces* sont des séries de requêtes d'E/S qui ont été enregistrées (tracées) sur un système réel en production soumis à une charge donnée. L'enregistrement peut être réalisé à différents niveaux (système de fichiers, pilote, etc.). Ces traces sont alors censées être représentatives d'un comportement applicatif réel, et peuvent être rejouées lors d'une opération de benchmarking. L'utilité des benchmarks à base de traces est similaire à celle des macro-benchmarks pour un domaine spécifique.

Dans Traeger et coll. (2008), les auteurs donnent également des conseils sur la pratique du benchmarking, qui s'appliquent également dans le cadre du travail de cette thèse où l'on utilise des benchmarks en phase d'exploration et de validation. Premièrement, si l'on souhaite utiliser un benchmark dans le cadre d'une étude, il est fondamental de donner un maximum de détails sur la manière dont ce dernier est exécuté : quelle configuration, sur quel système, avec quelles caractéristiques, dans quel environnement, etc. Ainsi, d'autres peuvent reproduire les expériences menées et valider les résultats. On peut arguer que les résultats provenant du lancement d'un benchmark impossible à reproduire, car

mal décrit, sont sans intérêt réel. Deuxièmement, il est important de justifier le choix du benchmark lancé et des caractéristiques associées (configuration, environnement d'exécution, etc.). Une fois de plus une expérience mal justifiée est difficile à valider par d'autres membres de la communauté scientifique. A noter que si ces assertions sont vraies pour l'exécution de benchmarks, elles le sont également pour toute expérience en général. Dans ce travail de thèse on prend bien soin d'expliquer dans les détails et de justifier tous les choix effectués lors des expériences.

Dans les sous-sections ci-dessous, on présente une liste relativement importante de benchmarks pour les systèmes de stockage. Une partie de ces benchmarks provient de Traeger et coll. (2008), complétés par d'autres non présentés dans cette étude. Il est parfois difficile de déterminer si un benchmark est plutôt un macro-benchmark ou un micro-benchmark. En effet certains ne représentent pas de contexte applicatif particulier, mais appliquent un nombre important d'opérations sur le système de stockage. On utilise donc la règle suivante : si l'on peut extraire de la description d'un benchmark un contexte applicatif dont le benchmark tente de reproduire le comportement d'E/S, alors c'est un macro-benchmark.

1.1 Micro-benchmarks

Les micro-benchmarks se focalisent sur un nombre réduit d'opérations. Leur utilité est d'évaluer finement l'impact des opérations testées sur les performances du système de stockage. Les micro-benchmarks présentés ici sont séparés en deux catégories : les tests ciblant un média de stockage donné (en général un disque dur) indépendamment du système de fichiers, et les tests s'effectuant au niveau du système d'exploitation, mesurant les performances du périphérique de stockage et des couches logicielles de gestion de ce périphérique. Les micro-benchmarks accédant directement au média de stockage sont dits *niveau bloc*, un bloc représentant la granularité d'accès à un périphérique de stockage secondaire. Les autres des benchmarks sont notés niveau *système de fichiers* car ils font intervenir les couches de gestion logicielles, en particulier le système de fichiers. Ces deux types de micro-benchmarks sont illustrés sur la figure 2.1.

a) Micro-benchmarks niveau bloc

Ces micro-benchmarks effectuent des requêtes d'E/S directement sur le périphérique testé, c'est à dire sans passer par les couches logicielles de gestion qui sont : (A) le système de fichiers virtuels et (B) le système de fichiers. La plupart des micro-benchmarks accédant directement au média de stockage ciblent un périphérique de type disque dur. On peut par exemple citer des outils comme et *Open Storage Toolkit* (Mesnier, 2011), ainsi que *VDBench* (Vandenbergh, 2012). Sous Linux, l'accès au média de stockage se fait via les fichiers virtuels créés dans le répertoire */dev*, rendant possible la lecture et l'écriture aux adresses physiques du périphérique.

uFlip (Bouganim et coll., 2009) est un outil de micro-benchmarking ciblant explicitement un périphérique à base de mémoire flash. Les tests sont définis sur la base d'une série de motifs (*patterns*) d'accès aux périphériques. Ces motifs étant eux mêmes définis via des paramètres qui sont : la taille des requêtes d'E/S, le type de requête, le motif d'accès (séquentiel / aléatoire), etc. Deux exemples de tests définis par uFlip sont illustrés sur la figure 2.2. Le premier, à gauche sur la figure, présente la schématisation d'une écriture séquentielle de taille 4 Ko. Le second, à droite sur la figure, schématise une lecture aléatoire de taille 2 Ko.

b) Micro-benchmarks niveau système de fichiers

Ces micro-benchmarks effectuent et mesurent les performances d'exécutions d'appels systèmes relatifs au stockage. La charge d'E/S des benchmarks niveau système de fichiers est donc appliquée à haut niveau dans le système de stockage, par opposition aux benchmarks niveau média (voir figure

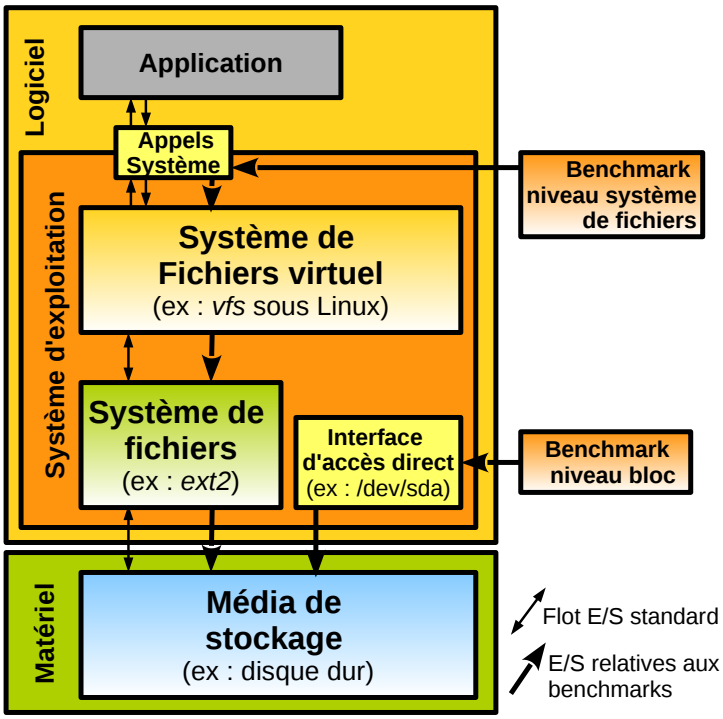


Figure 2.1 – Les deux types de micro-benchmarks classifiés selon le niveau d'application de leur charge d'E/S : niveau système de fichiers et niveau bloc.

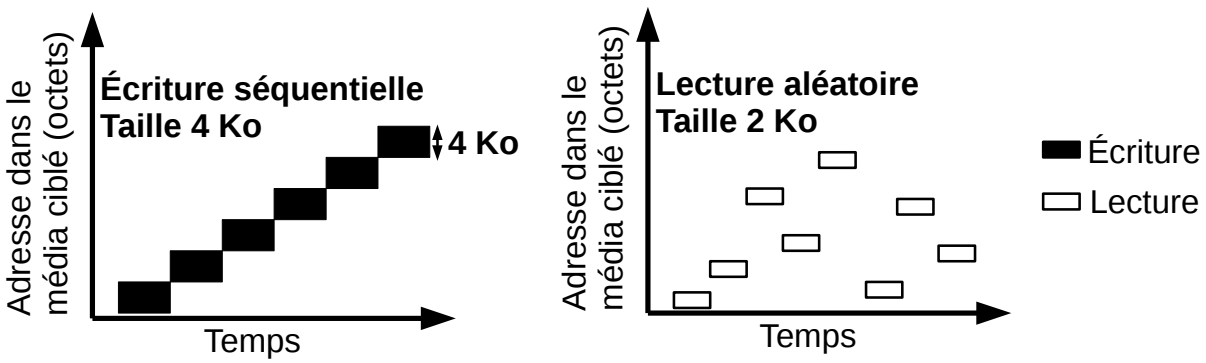


Figure 2.2 – Deux exemples de tests effectuées par uFlip, un benchmark niveau bloc effectuant des accès directs à des médias de stockage à base de mémoire flash. A gauche, une écriture séquentielle ; à droite une lecture aléatoire. Image adaptée de Bouganim et coll. (2009).

2.1). Cela permet notamment à ces benchmarks de tester les performances des couches logicielles de gestion, en particulier les systèmes de fichiers.

Ils existe de nombreux micro-benchmarks niveau système de fichiers. Si l'on s'intéresse aux types d'opérations testées, on constate que la grande majorité des benchmarks niveau OS étudiés dans le cadre de ce travail de thèse (Bray, 1996; Coker, 2009; OpenBenchmarking.com contributors, 2013; Rosenblum et Ousterhout, 1992; Axboe, 2014; Norcott et Capps, 2003; Heger et coll., 2008; Kuoppala, 2002; Park et coll., 1990) supportent au moins le test des opérations classiques de lecture et d'écriture sur un ou plusieurs fichier(s). C'est à dire pour Linux les appels système *read()* et *write()*. Des benchmarks simples comme *Bonnie* (Bray, 1996), *Bonnie++* (Coker, 2009), *IOStone* (Park et coll., 1990), ou *Sprite LFS large / small files benchmarks* (Rosenblum et Ousterhout, 1992) testent les performances d'un système de fichiers via la création, la lecture, et l'écriture d'un ou plusieurs fichiers. Ces benchmarks offrent en général la possibilité de varier le pattern d'accès (séquentiel / aléatoire). Des benchmarks plus évolués ajoutent les fonctionnalités suivantes :

— La possibilité d'utiliser les fonctionnalités d'E/S asynchrones est offerte dans des benchmarks comme *Aio-stress* (OpenBenchmarking.com contributors, 2013), *Fio* (Axboe, 2014) ou *Iozone* (Norcott et Capps, 2003) ;
— L'accès aux fichiers via les fonctionnalité de cartographie mémoire (*memory mapping*) offerte par les systèmes d'exploitations est possible dans Fio et IOZone. Sous Linux cela est réalisé via l'appel système *mmap()* ;
— Certains benchmarks sont multi-threadés ou multi-processus, pour simuler le comportement d'E/S de plusieurs programmes s'exécutant dans un système d'exploitation, on peut citer Fio, IOZone, *the Flexible File System Benchmark* (Heger et coll., 2008) ou encore TioBench (Kuoppala, 2002).

Alors que la majorité des benchmarks niveau système de fichiers mettent le focus sur les performances des accès aux données, certains se concentrent sur l'impact des méta-données sur les performances du système de fichiers. Des benchmarks comme *Listrates* (Lensing et coll., 2013) ou *Metarates* (UCAR, 2004) s'appliquent à définir des environnements de tests particuliers, avec notamment des nombres de fichiers et répertoires importants au sein d'une arborescence de hauteur et largeur élevées.

c) Micro-benchmarks - conclusion

En conclusion de cette sous-section concernant les micro-benchmarks, on peut premièrement voir que les benchmarks niveau bloc sont majoritairement tournés vers les disques durs. uFlip fait exception, ciblant un périphérique à base de mémoire flash. Néanmoins, uFlip cible un périphérique de type bloc, ce qui implique l'utilisation d'une FTL. Dans le cadre de ce travail on s'intéresse principalement au stockage sur puce embarquée via FFS. uFlip n'est donc pas adapté au contexte de notre étude.

Pour ce qui est des benchmarks niveau système de fichiers, on constate que les opérations les plus testées sont les traditionnelles lecture et écriture de données dans des fichiers (appels systèmes *read()* et *write()* sous Linux). Dans le cadre du travail présenté dans cette thèse, les systèmes embarqués complexes, il est assez raisonnable de penser que ces opérations constituent une grande partie des accès au stockage secondaire. Ainsi, parce que l'on se focalise sur la pile de gestion du stockage flash via FFS sous Linux, c'est sur ces opérations que l'on se concentre dans la phase de modélisation de ce travail de thèse.

1.2 Macro-benchmarks

Les macro-benchmarks soumettent le système de stockage testé à un nombre important d'opérations formant une charge de travail complexe censée être représentative par rapport à une application réelle. Un macro-benchmark est donc toujours relatif à un contexte applicatif donné, par exemple une base de données, un serveur web, etc. On peut donc classer les macro-benchmarks selon le domaine applicatif

ciblé. Deux contextes principaux ressortent : (A) les serveurs mail et le contexte des systèmes de fichiers en réseau ; et (B) les bases de données, notamment transactionnelles en ligne. Notons que nous sommes là très loin du domaine de l'embarqué.

a) Serveurs mails et systèmes de fichiers en réseau

Postmark (Katcher, 1997) est un benchmark pour systèmes de fichiers représentant la charge d'E/S d'un serveur mail appliquée à un système de stockage. Sa simplicité d'utilisation et sa notoriété font qu'il est utilisé dans de nombreuses études, notamment dans les systèmes embarqués (Liu et coll., 2010; Kim et Kim, 2012; Lim et Park, 2006; Kang et Miller, 2009). Postmark est relativement ancien et sa configuration par défaut n'est plus adaptée aux performances des systèmes informatiques actuels (Traeger et coll., 2008). Il est donc nécessaire de le configurer correctement si l'on souhaite l'utiliser aujourd'hui. *Filemark* (Bryant et coll., 2002) est une amélioration de Postmark apportant le support du *multi-threading* (Postmark n'exécute qu'un processus) et une granularité de mesure de temps plus fine pour le calcul des résultats. Dans le domaine des serveurs de messagerie, on peut également citer JetStress (Johnson, 2013) qui reproduit l'activité d'un serveur *Microsoft Exchange*.

De nombreux benchmarks sont définis pour tester les performances des systèmes de fichiers en réseau. On peut citer SPECSFS (SPEC Contributors, 2008), NetBench (Memik et coll., 2001) et DBench (Tridgell, 2008), ou encore FStress (Anderson et Chase, 2002). Ces benchmarks ciblent des systèmes de fichiers comme NFS (Unix / Linux) et CIFS / Samba (Windows).

b) Bases de données et traitements transactionnels en ligne

Dans le contexte des bases de données, en particulier celles intégrées dans les serveurs web actuels, les systèmes de stockage sont soumis à des charges d'E/S potentiellement très importantes. Ils sont également considérés comme un goulet d'étranglement pour les performances de ces systèmes informatiques. Il existe ainsi un grand nombre de macro-benchmarks reproduisant le comportement de bases de données. Certains benchmarks comme les benchmarks *TPC* (TPC Contributors, 2014) et *SPC* (SPC Contributors, 2013) sont très standardisés par des comités d'experts et reconnus de manière assez unanime dans la communauté industrielle. Ce sont des benchmarks ciblant des systèmes de stockages contenant des bases de données accédées via le langage SQL. Ils sont proposés sous formes de spécifications, et c'est à l'utilisateur de mettre en place l'implémentation (création des tables, exécution des requêtes SQL, etc.). *FileIO* est un benchmark de la suite *SysBench* (Kopytov, 2012) proposant un mode dit OLTP (*On Line Transaction processing*, traitement transactionnel en ligne) ciblant une base de donnée de type MySQL. Toujours dans le domaine des bases de données on peut également citer *IOBench* (Wolman et Olson, 1989). De manière générale, les benchmarks présentés dans ce paragraphe produisent une charge applicative censée représenter celle de systèmes à assez large échelle, comme des bases de données commerciales ou de sites web de tailles importantes.

Au niveau du domaine de l'embarqué on retrouve une catégorie de benchmarks s'intéressant aux performances du Système de Gestion de Bases de Données (SGBD) *SQLite*. Ce SGBD est très utilisé dans l'embarqué notamment de par sa simplicité de mise en œuvre (la base de données est un fichier simple). *Androbench* (Kim et Kim, 2012), ciblant l'OS Android, est un exemple de ce type de benchmarks. Néanmoins, Androbench n'émule pas de comportement applicatif particulier et se rapproche plus d'un micro-benchmark.

c) Autres macro-benchmarks pour systèmes de stockage

On peut retrouver dans plusieurs études des benchmarks dits de *compilation* pour évaluer les performances d'un système de fichiers. Le procédé appliqué consiste à compiler diverses applications tout en mesurant les performances du système de stockage, par exemple le temps total de compilation. Les logiciels compilés sont généralement SSH, le noyau Linux, Am-Utils, ou encore Emacs. Comme

indiqué par les auteurs de Traeger et coll. (2008), le fait que ces benchmarks présentent une charge CPU très importante peut perturber les résultats des tests lorsque l'on s'intéresse uniquement au système de stockage. Il est, de plus, difficile de comparer des résultats de benchmarks de compilations lancés sur des machines avec des CPU et / ou des chaînes de compilation différents. Le benchmark *Andrew Benchmark* a été créé pour évaluer spécifiquement le système de fichiers distribué *AndrewFS* (Howard et coll., 1988). Ce benchmark est censé reproduire la charge applicative de multiples utilisateurs d'un système de type Unix.

La frontière micro / macro peut parfois être ténue. Certains micro-benchmarks, comme par exemple FileBench (Wilson, 2008) offrent la possibilité de décrire les tests à effectuer sous la forme de profils. FileBench offre des profils prédéfinis correspondant à des charges applicatives données. Il existe par exemple un profil permettant de simuler la charge appliquée par une base de données de type *MongoDB*. De manière similaire, *Bonnie++* (Coker, 2009) applique au système testé une série d'opérations simples (création, lecture, écriture de fichier). L'auteur du benchmark explique que dans son ensemble, la série de tests peut être vue comme simulant le comportement de serveurs de type proxy / mail / *usenet* (usenet est un système de forums en réseau). Ainsi, certains micro-benchmarks peuvent également être vus comme des macro-benchmarks.

d) Macro-benchmarks - conclusion

Pour conclure sur le sujet des macro benchmarks, on peut dans le cadre de ce travail de thèse noter plusieurs observations. Premièrement, contrairement à certains domaines applicatifs comme les bases de données, il n'existe pas de benchmarks unanimement reconnu pour l'étude des performances des systèmes de fichiers embarqués. De plus, si l'on exclut le cas particulier de SQLite, il n'existe aucun benchmark produisant une charge applicative telle que celles que l'on peut trouver dans les systèmes embarqués. Pour preuve, on peut regarder les articles traitant des FFS présentés dans le premier chapitre de cette thèse. La plupart des articles (8) présentent une évaluation de performances en fin d'article. Sur les 8, seulement 3 utilisent des macro-benchmarks : il s'agit de Postmark dans ces trois cas. Postmark n'est absolument pas dédié à l'embarqué, mais il semble que sa popularité fasse qu'il soit assez utilisé dans ce domaine.

1.3 Les traces d'E/S

L'utilisation de traces est la troisième et dernière catégorie de benchmarks. Elle consiste en (1) la collecte de traces d'accès à un système de stockage source en environnement réel ; et (2) la "re-exécution" (*replay*) de ces traces sur le même système ou un autre système cible, tout en mesurant les performances. Les traces peuvent également être analysées à des fins de caractérisation des comportements d'E/S des applications exécutées sur le système tracé.

Rejouer une trace s'apparente à un macro-benchmark, dans le sens où une trace enregistrée est censée être représentative d'une charge de travail réelle : le système source peut être tracé lors de son exécution standard, il s'agit alors d'une trace en environnement réel. Le système tracé peut également exécuter un macro-benchmark. Les traces peuvent être capturées à différents niveaux dans la pile gestion de stockage, comme indiqué par les auteurs de Traeger et coll. (2008). On retient deux niveaux principaux : le niveau *appels système* (VFS), et le niveau *pilote*. Les traces du niveau appels systèmes représentent une charge de travail applicative pure, et les traces au niveau pilote représentent une charge applicative modulée par les couches de gestion logicielles, les caches systèmes au niveau VFS et le système de fichiers. Ces niveaux peuvent être mis en relation avec les deux niveaux de micro-benchmarking précédemment présentés : le niveau de trace appels système correspondant au niveau système de fichiers, et le niveau de trace pilote correspondant au niveau bloc.

a) Traces d'E/S fréquemment utilisées dans le domaine du stockage

Certaines traces sont relativement populaires et sont utilisées dans de nombreuses études présentant de nouveaux systèmes de stockage, notamment pour en évaluer les performances. *WebSearch* et *Financial* (UMASS, 2009) sont des traces disponibles sur le site de l'université du Massachusets et proviennent de la trace d'un moteur de recherche, et d'une application financière de traitement transactionnel en ligne. La trace *Cello99* (HP Labs, 1999) contient un an d'activité du serveur Cello tournant dans le laboratoire *HP Labs*. Il existe également d'autres traces niveau blocs disponibles sur le site de l'association SNIA (*Storage Networking Industry Association*) (SNIA, 2011).

Néanmoins, il faut noter que (1) ces traces proviennent toutes d'applications à large échelle, et (2) ces traces sont enregistrées au niveau pilote pour des périphériques de type bloc. Elles sont utilisées notamment dans des études présentant des mécanismes de FTL (par exemple dans Gupta et coll. (2009)) et ne sont ni adaptées au domaine de l'embarqué en général, ni adaptées au contexte des systèmes de fichiers pour mémoires flash.

b) Récolte des traces d'E/S

Les traces peuvent être récoltées via l'instrumentation du code du système d'exploitation aux différents niveaux concernés, ou encore en utilisant des outils dédiés (qui eux même représentent une instrumentation du système d'exploitation). Au niveau pilote, *BlkTrace* (Brunelle, 2007) permet de tracer sous Linux les accès aux périphériques de type bloc. Au niveau appels systèmes, on peut citer *FileMon* (Russinovich et Cogswell, 2006) sous MS Windows, *TraceFS* (Aranya et coll., 2004) et *VFS Interceptor* (Wang et coll., 2008) sous Linux. *ReplayFS* (Joukov et coll., 2005) permet de rejouer les traces enregistrées par TraceFS.

Dans le domaine de l'embarqué, les auteurs de Lee et Won (2012) présentent *Mobile Storage Analyzer* (MOST), un outil permettant de récolter des traces relatives au stockage dans des systèmes embarqués utilisant l'OS Android (et donc le noyau Linux). MOST cible les systèmes de stockage à base de FTL de type eMMC. MOST est constitué d'un noyau Linux modifié et de l'outil BlkTrace traçant les appels niveau bloc au périphérique de stockage. MOST trace donc au niveau pilote. Néanmoins, certains efforts sont faits pour lier les évènements tracés au niveau pilote à des informations de haut niveau, notamment les processus initiateurs des E/S tracées. Toujours sur Android, les auteurs de Kim et coll. (2012a) utilisent également BlkTrace pour récolter des traces au niveau pilote pour un système de stockage sur carte SD (FTL).

Aucune des solutions présentées ci-dessus n'est concrètement applicable dans le cadre de ce travail de thèse. Le stockage à base de FFS ne passe pas par la couche de type bloc de Linux, ce qui exclut les traceurs niveau pilote présentés. Pour ce qui est des traceurs niveau appels système, TraceFS est incompatible avec les versions du noyau Linux compatibles avec les plates-formes matérielles utilisées pendant cette thèse. Le code de VFS Interceptor n'est pas disponible sur internet. Une infrastructure composée de plusieurs outils de trace et d'outils d'analyses associés a donc été développée au cours de la thèse, pour utilisation au cours des différentes phases de travail, en particulier les phases d'exploration, de modélisation et de validation.

1.4 Benchmarking des systèmes de stockage à base de mémoire flash : conclusion

De l'étude sur les micro-benchmarks, on retient le fait que la plupart des benchmarks niveau pilote ciblent un disque dur comme média de stockage, ce qui est incompatible avec le travail présenté ici. Les micro-benchmarks niveau appels systèmes sont néanmoins utilisables dans l'embarqué. On constate que ce type de micro-benchmarks teste en large majorité les opérations de lecture et d'écriture de données dans les fichiers, les appels systèmes *read()* et *write()* sous Linux. C'est sur ces opérations que se concentre le travail présenté dans les chapitres suivants. Dans le cadre de ce travail les micro-benchmarks sont utilisés en phase d'exploration et de modélisation pour évaluer l'impact de

mécanismes à granularité fine sur les performances et la consommation du système de stockage. Il sont également utilisés en phase de validation pour vérifier les modèles de manière précise.

Contrairement à certains domaines comme les bases de données à large échelle, il n'existe pas de macro-benchmark ou de trace unanimement reconnue pour mesurer et explorer les performances des systèmes de fichiers. Il existe pourtant de nombreux macro-benchmarks dédiés aux systèmes de fichiers en général. On peut cependant constater qu'aucun macro-benchmark ne reproduit un comportement applicatif de type embarqué. Il n'existe pas non plus de trace ou macro-benchmark ciblant les systèmes de fichiers dédiés aux mémoires flash, car la plupart suggèrent un disque dur comme média de stockage. Le macro-benchmark *Postmark* semble cependant assez populaire dans le domaine de l'embarqué de par sa simplicité de mise en œuvre et d'utilisation.

2 Mesure des performances et de la consommation des systèmes de stockage à base de mémoire flash

L'exploration des métriques de performances et de consommation est la première étape du processus scientifique adopté lorsque l'on souhaite caractériser, modéliser et / ou optimiser les valeurs de ces métriques dans un système informatique. Dans cette section on présente d'abord les principales métriques de performances et de consommation. Par la suite on effectue un état de l'art des différentes études traitant de l'exploration (1) des performances et (2) de la consommation des systèmes de stockage à base de mémoire flash. L'exploration des performances de ces systèmes est un sujet très large, on se concentre donc ici sur le contexte des FFS. Pour ce qui est de la consommation, un état de l'art général sur les systèmes de stockage à base de mémoire flash est effectué.

2.1 Métriques de performances et de consommation des systèmes de stockage

a) Métriques de performances dans le contexte des FFS

Les métriques de performances d'un système de fichiers pour mémoires flash sont nombreuses. En observant les différents travaux scientifiques étudiés dans la section de présentation des systèmes de fichiers pour FFS, trois catégories de métriques ressortent : les métriques de performances pures (généralement liées aux vitesses de transferts de données), les métriques propres aux contraintes subies par les FFS (nécessité d'une faible empreinte RAM, passage à l'échelle de l'empreinte RAM et du temps de montage, etc.) et les métriques relatives à la répartition de l'usure. Il existe également des métriques fortement spécifiques à un benchmark en particulier : par exemple Postmark définit la notion de transaction comme un lot d'opérations de lectures et d'écritures, et donne des résultats en nombre de transactions par seconde. Ces métriques ne sont pas abordées ici car elles ne sont pas généralistes.

Métriques d'évaluation des performances pures On retrouve ici des métriques classiques de l'évaluation de performance de systèmes de stockage : à très haut niveau, le *temps d'exécution total d'un benchmark* (ou plus généralement d'un test) est souvent utilisé pour comparer plusieurs FFS entre eux, ou plusieurs configurations d'un même FFS. A un niveau de granularité plus fin, on peut également mesurer les *temps d'exécution d'une opération en particulier*, par exemple un appel système *read()*. Cela est généralement présenté sous forme de moyenne de plusieurs appels à cette opération, ou d'une distribution de temps d'exécutions de plusieurs appels à l'opération pour une analyse plus fine. Le *nombre d'opérations flash* (lecture / écriture / effacement) généré par une opération plus haut niveau, par exemple l'exécution d'un appel système *write()* ou encore d'un benchmark complet, est également une métrique qui revient souvent.

Les *débits* en lecture et écriture sont également une métrique assez classique, ils peuvent être relevés à différents niveaux : par exemple les niveaux applicatif, FFS, ou encore au niveau du pilote.

L'amplification en écriture est une métrique qui dénote du fait que lorsque l'applicatif demande à écrire une certaine quantité de données, cela peut avoir pour effet l'écriture en flash d'une quantité supérieure d'information. Cela est notamment dû aux potentielles opérations de ramasse-miettes qui peuvent survenir, mais aussi pour des raisons de gestion interne de la couche de gestion flash. En d'autres termes l'amplification, en écriture correspond aux opérations d'écritures flash qui ne sont pas directement déclenchées par la charge de travail appliquée au système de stockage. A noter que la présence de caches en amont de la couche de gestion (par exemple le page cache de Linux situé au dessus du FFS) peut faire que l'on écrit moins de données sur le média de stockage que demandé par l'applicatif. Ainsi, il est en général préférable de considérer, lors du calcul de l'amplification en écriture, la charge de travail directement en entrée de la couche de gestion (indépendante du cache). On peut retrouver plusieurs méthodes pour calculer l'amplification en écriture (Hu et coll., 2009; Chiao et Chang, 2011) d'une charge de travail donnée sur un système de stockage donné. Une formule intuitive étant :

$$\text{Amplification en écriture} = \frac{\text{Taille totale flash écrite}}{\text{Espace écrit par la charge de travail}} \tag{2.1}$$

On peut de la même manière définir l'amplification en lecture. A noter qu'une forte amplification en écriture ou lecture ne dénote pas forcément un impact directement négatif sur les performances, en termes de débit ou de temps de réponse. Par exemple, un mécanisme de ramasse-miettes en arrière plan déclenche des écritures et lectures supplémentaires par rapport à la charge d'E/S, sans pour autant impacter les performances.

Métriques d'évaluation propres aux FFS Ces métriques sont directement liées aux fonctionnalités et contraintes relatives aux FFS, comme par exemple le fait qu'ils soient utilisés dans un environnement de type embarqué. Les métriques relatives à la contrainte de répartition de l'usure sont quant à elles présentées de manière distincte plus loin dans ce document.

Au niveau des contraintes propres au domaine de l'embarqué, comme présenté précédemment, le *temps de montage* du système de fichiers est une métrique particulièrement importante. L'*empreinte mémoire* (en octets) l'est également.

Pour les systèmes de fichiers supportant la compression, on peut évaluer l'*efficacité de la compression* en comparant la taille d'une arborescence de fichiers / répertoires avec la taille flash physique occupée par une partition nouvellement créée contenant cette arborescence. L'*impact de la compression sur les performances* peut lui être évalué en comparant les performances d'un FFS avec et sans compression (Liu et coll., 2010).

Enfin, certaines études (Homma, 2009) évaluent la taille des méta-données propres au FFS en flash, c'est à dire la taille flash indisponible pour l'utilisateur.

Métriques d'évaluation de la répartition de l'usure Diverses métriques permettent de mesurer et d'estimer la répartition de l'usure. Intuitivement on pourrait penser au nombre total d'effacements subits par la puce, ou encore la moyenne des compteurs d'effacements par bloc, pour une charge de travail donnée. Certes, ces valeurs peuvent donner une idée globale de l'usure, et permettent de comparer les performances de différentes couches de gestion. Néanmoins, elles ne sont pas indicatives de la répartition des effacements, c'est à dire la différence plus ou moins importante entre les valeurs des compteurs d'effacement des blocs de la mémoire considérée. Pour prendre en considération cette valeur, il faut s'intéresser à des valeurs comme l'écart type de la distribution des compteurs d'effacements (Liu et coll., 2010), ou encore la différence entre le bloc le plus effacé et le bloc le moins effacé (Chang et Kuo, 2002).

Pour illustrer ces concepts, prenons l'exemple d'une expérience fictive dans laquelle on applique une charge de travail donnée à deux systèmes de gestion flash A et B, gérant chacun une puce flash

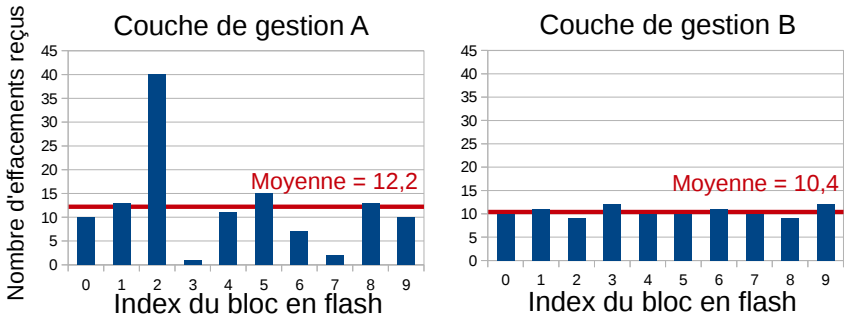


Figure 2.3 – Résultats concernant la répartition de l'usure pour une expérience fictive comparant deux systèmes de gestion flash. On peut voir que la répartition de l'usure du système A (gauche) est moins bonne que celle du système B (droite) du fait de l'envergure de la différence (écart-type) entre les compteurs d'effacements des blocs flash.

fictive composée de dix blocs. A la fin de l'expérience, les compteurs d'effacements sont les suivants : $E_A = \{10, 13, 40, 1, 11, 15, 7, 2, 13, 10\}$ et $E_B = \{10, 11, 9, 12, 10, 10, 11, 10, 9, 12\}$. Ces listes sont définis comme suit : $E_X(i)$ (i-ème membre de E_X) correspond au nombre d'effacements effectués sur le bloc flash d'indice i dans la puce gérée par le système de gestion X.

Métrique de répartition de l'usure	**Résultats pour le système de gestion A**	**Résultats pour le système de gestion B**
Nombre total d'effacements	122	104
Moyenne des compteurs d'effacements par bloc	12.2	10.4
Écart-type de la distribution des compteurs d'effacement	10.80	1.07
Différence entre le nombre d'effacements subis par le bloc le plus effacé et le bloc le moins effacé	39	3

Table 2.1 – Résultats chiffrés pour l'exemple d'expérience illustrant les métriques de répartition de l'usure

Les valeurs des métriques de répartition de l'usure citées plus haut sont présentées dans la table 2.1. On peut voir que les nombres totaux d'effacements et les moyennes des compteurs sont relativement similaires pour les deux systèmes. A l'inverse, les deux autres métriques montrent que le système B effectue une répartition de l'usure beaucoup plus efficace : l'écart type de la distribution des compteurs d'effacements est beaucoup plus faible pour B, ainsi que la différence d'effacements entre le bloc le plus effacé et le moins effacé. Cela peut se confirmer si l'on représente pour chaque bloc le nombre d'effacements subits comme illustré sur la figure 2.3.

Dans le contexte des FFS sous Linux, les auteurs de Homma (2009) présentent les notions de répartition de l'usure globales et locales. Dans un système où le stockage secondaire est composé d'une puce contenant plusieurs partitions, selon le modèle de FFS utilisé on peut avoir plusieurs configurations :

— Conf. 1 : Avec UBIFS et une couche d'abstraction telle que UBI, un unique volume UBI est créé sur la puce flash et des volumes UBIFS sont créés sur le volume UBI ;
— Conf. 2 : Avec d'autres systèmes de fichiers (par exemple JFFS2 ou YAFFS2), des partitions MTD (zones physiquement distinctes) sont créées et un volume JFFS2 / YAFFS2 est créé dans chaque partition.

Si l'une des partitions d'un tel système reçoit une charge en écriture supérieure à la charge reçue par les autres partitions (voir exemple sur la figure 2.4), alors la configuration 2 aboutira à une concentration importante d'effacements sur l'espace physique correspondant à cette partition, ce qui donne une mauvaise répartition de l'usure du point de vue de l'espace flash total. On parle de répartition de l'usure *locale*, par rapport à une partition. De son coté, la couche UBI (qui on le rappelle

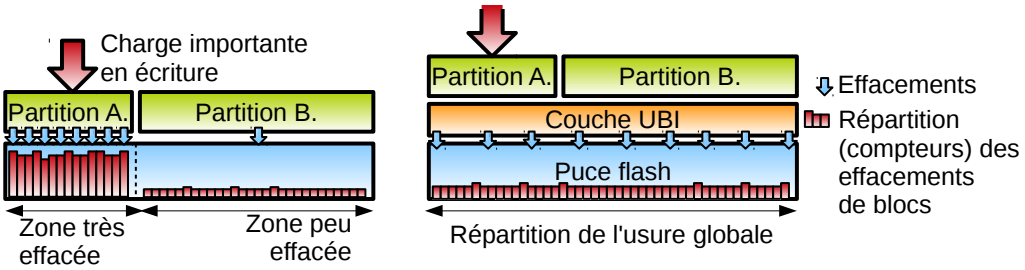


Figure 2.4 – Répartition de l'usure locale (droite) et globale (gauche)

s'occupe entre autres de la répartition de l'usure) de la configuration 1 permet une répartition sur l'intégralité de l'espace flash géré : on parle de répartition de l'usure *globale*.

b) Métriques de consommation énergétique

On peut extraire de la littérature deux métriques principales concernant la consommation énergétique des systèmes de stockage : la puissance et l'énergie.

La puissance Dans un circuit électrique, la puissance, dont l'unité est le Watt (W), dénote la consommation d'un circuit à un instant donné. On peut calculer la puissance P pour un circuit électrique via la loi suivante :

$$P = U * I \tag{2.2}$$

I étant l'intensité et U la tension aux bornes du circuit. On peut donc mesurer dans un circuit la puissance en plaçant un voltmètre en parallèle du circuit et un ampèremètre en série sur le chemin de l'alimentation du circuit. Il est également possible de mesurer la puissance P grâce à la tension U_{res} aux bornes d'une résistance R située sur le chemin de l'alimentation du circuit, en utilisant la formule suivante :

$$P = \frac{U_{circuit} * U_{res}}{R} \tag{2.3}$$

Lorsque l'on souhaite mesurer la puissance consommée par un élément d'un système informatique soumis à une charge de travail donnée, il est important de caractériser la puissance dite "à vide" ou "au repos" (*idle power* en anglais). Cela permet alors par soustraction de déterminer le coût effectif en puissance dû à l'activité générée par la charge de travail sur le composant.

L'énergie L'énergie E, dont l'unité est le joule (J) est la quantité de puissance consommée en une période de temps donnée.

$$E(t) = \int_0^t P(t)\,\mathrm{d}t \tag{2.4}$$

En pratique, la mesure de P est discrète, toujours effectuée avec une certaine fréquence d'échantillonage. On peut alors obtenir une approximation de l'énergie par la formule suivante :

$$E(t) = \sum_{i=0}^{n} P_i * \Delta t \tag{2.5}$$

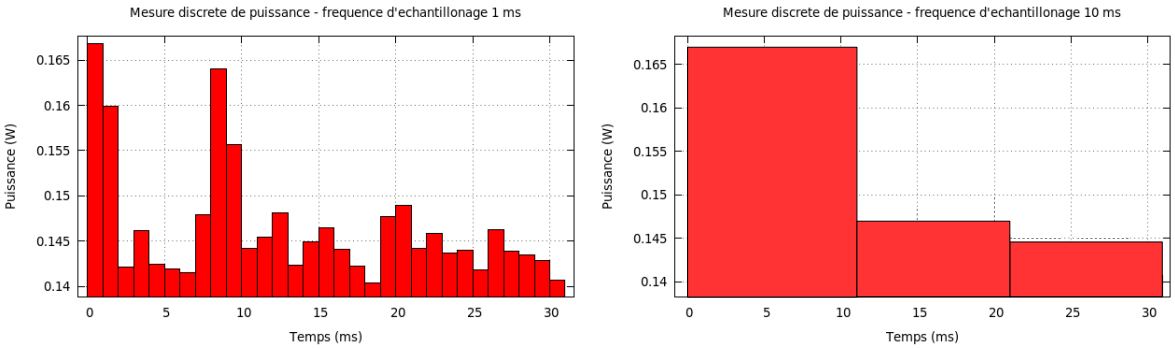


Figure 2.5 – Deux mesures fictives de la puissance au cours du temps pour un évènement quelconque, avec deux fréquences d'échantillonage différentes : un échantillon tous les millisecondes à gauche, et un échantillon tous les 10 millisecondes à droite.

Bien entendu la fréquence d'échantillonage détermine la précision de la mesure. Un exemple de deux mêmes mesures de puissances (fictives) en fonction du temps avec différentes fréquences d'échantillonage est présenté sur la figure 2.5.

2.2 Exploration des performances dans le contexte des FFS

L'exploration des performances des systèmes de stockage étant un contexte très large, on se concentre ici sur le domaine des FFS. L'exploration est réalisée en appliquant une charge de travail niveau appels système, un benchmark, sur le FFS tout en mesurant certaines métriques de performances. La charge de travail peut être appliquée via un benchmark *nommé* (c'est à dire un benchmark existant, par exemple Postmark) ou un benchmark dit *ad-hoc* (Traeger et coll., 2008), c'est à dire développé explicitement pour les besoins de l'étude. On retrouve le thème de l'exploration des performances des FFS dans des études spécifiquement dédiées au benchmarking de FFS (Liu et coll., 2010; Homma, 2009; Toshiba Inc., 2009; Opdenacker, 2010), ainsi que dans des articles présentant de nouveaux FFS et les comparant à l'existant via une étude de performances (c'est le cas pour les divers FFS présentés au chapitre précédent).

les métriques utilisées dans le cadre de l'évaluation de performances des FFS sont listées dans la section précédente. Sur un total de 19 articles présentant des études traitant des FFS, on peut lister les métriques les plus populaires dans la table 2.2. Les articles sont les suivants : Kang et Miller (2009); Park et coll. (2006b); Liu et coll. (2010); Lim et Park (2006); Reardon et coll. (2012); Kim et coll. (2009a); Park et Kim (2013); Homma (2009); Toshiba Inc. (2009); Opdenacker (2010); Park et coll. (2008); Jung et coll. (2008a); Kim et coll. (2012b); Kawaguchi et coll. (1995); Hwang (2012); Lee et coll. (2009b); Lim et coll. (2006); Lee et Kim (2013); Jung et coll. (2005).

On peut classifier ces métriques comme suit :

— Les mesures de temps (temps d'exécutions de diverses opérations, temps de montage, débits, etc.) ;
— Les mesures d'occurrences d'évènements et des paramètres associés à chacun de ces évènements (nombre et adresses cibles d'opérations flash, distributions d'effacements, etc.) ;
— Diverses autres métriques concernant l'empreinte RAM, le coût en espace flash des métadonnées, l'efficacité de la compression.

a) Mesures de temps

De manière générale, le temps d'exécution d'une opération est calculé grâce à une horloge au niveau de laquelle on observe le temps de début de fin de l'opération. Le temps d'exécution de l'opération

Métrique	Nombre d'articles utilisant cette métrique
Temps de montage	**12**
Temps d'exécution d'un benchmark ou d'une opération à granularité plus fine	**11**
Débit en lecture / écriture	**10**
Empreinte mémoire	6
Distribution des effacements ou autre métrique de répartition de l'usure	6
Métrique propre à un benchmark donné (par ex. transactions / sec pour Postmark)	5
Nombre d'opérations flash pendant un test	4
Coût de stockage des méta-données	4
Répartition données chaudes / froides	2
Amplification en écriture / lecture	1
Compression	1

Table 2.2 – Popularité des métriques d'évaluation des performances sur un total de 19 articles traitant du benchmarking de FFS, présentant de nouveaux FFS ou des optimisations relatives aux FFS.

correspond alors à la différence entre le temps de fin et le temps de début :

```
temps_début = getTime(horloge);
/* exécution de l'opération */
temps_fin = getTime(horloge);
temps_exécution = temps_fin - temps_début;
```

Au niveau de Linux, l'horloge utilisée est généralement le temps système. On obtient la valeur du temps système via une commande ou fonction de type *GetTime*. Parmi les commandes utilisées pour obtenir ce temps, on peut citer la commande *time* de Linux. La précision (granularité) de cette commande dépend des implémentations, mais elle est généralement de l'ordre de la milliseconde, ce qui dans le cas de certaines opérations rapides n'est pas suffisant. L'appel système Linux *time()* (à ne pas confondre avec la commande du même nom présentée précédemment) présente une granularité de l'ordre de la seconde. Pour une mesure de temps efficace, l'appel système *gettimeofday()* peut être utilisé, donnant un résultat en microsecondes. Il faut également noter que la fréquence à laquelle le temps système est mis à jour impacte également les mesures de temps : si cette fréquence est trop faible, même une fonction comme *gettimeofday()* ne donnera pas de bons résultats : dans le cas d'appels trop fréquents à *gettimeofday()* avec une horloge mise à jour à faible fréquence, on risque de voir apparaître plusieurs mesures présentant le même temps. Pour une mesure à granularité fine, par exemple si l'on souhaite mesurer le temps d'exécution d'un appel de fonction dans le noyau, il est nécessaire (1) d'utiliser des outils de trace dédiés comme par exemple *Ftrace* (Rostedt, 2008) ou (2) d'instrumenter le code dont on souhaite mesurer le temps d'exécution, comme par exemple dans Reardon et coll. (2012). Dans cette même étude les auteurs indiquent, à raison, qu'il est important de minimiser les latences supplémentaires induites par l'instrumentation elle-même (*overhead*).

Cela dit, il est intéressant de noter que de nombreuses études utilisant des benchmarks ad-hoc n'indiquent pas la méthode de mesure du temps utilisée. Par exemple, sur les 12 articles traitant du temps de montage, un seulement (Opdenacker, 2010) indique la commande utilisée pour la mesure du temps (commande *time*).

Dans le cas de benchmarks comme Postmark (utilisé dans Liu et coll. (2010); Kim et Kim (2012); Lim et Park (2006); Kang et Miller (2009); Opdenacker (2010)), les mesures de temps sont directement

réalisées par le benchmark. Les benchmarks utilisent eux-mêmes des méthodes de mesures de temps comme présentés précédemment. Le benchmark Postmark utilise quant à lui l'appel système *time()* en début et en fin de benchmark, puis calcule la différence entre ces deux valeurs pour obtenir le temps d'exécution total du benchmark, à partir duquel il calcule d'autres métriques de performances comme des débits.

b) Mesures d'occurrences d'évènements et autres métriques

La mesure d'occurrences d'évènements se fait via l'utilisation d'outils de traces (sous Linux on peut citer des outils comme *SystemTap* (Eigler et coll., 2005)), ou l'instrumentation du code du FFS, c'est à dire du noyau Linux dans notre cas (Reardon et coll., 2012; Toshiba Inc., 2009; Homma, 2009). Encore une fois, dans plusieurs articles les méthodes d'obtention de certaines informations comme le nombre de lectures / écritures / effacements flash ne sont pas clairement indiquées.

L'empreinte mémoire d'un FFS peut être estimée en regardant la taille du segment code occupée dans les fichiers objets représentant le résultat de la compilation des sources du système de fichiers (Liu et coll., 2010), par exemple via l'outil Linux *size*. On peut également observer via *free* la quantité de mémoire vive occupée dans un système après le montage d'un FFS (Opdenacker, 2010). Pour ce qui est du sur-coût flash des méta-données nécessaires au fonctionnement du FFS, il est estimé en mesurant la taille maximale que peut prendre un fichier écrit sur une partition nouvellement créée (compression désactivée) (Homma, 2009).

La qualité de la répartition de l'usure peut être évaluée en traçant les occurrences des opérations d'effacement, et les indices des blocs sur lesquelles elles sont appliquées Homma (2009).

2.3 Exploration de la consommation des systèmes de stockage à base de mémoire flash NAND

Comme énoncé plus haut on s'intéresse ici aux études sur la consommation des systèmes de stockage à base de mémoire flash de manière générale. On rappelle que les métriques principales de consommation sont la puissance et l'énergie, présentée en section précédente (voir la section concernant les métriques de consommation 54). L'exploration des métriques de consommation par la mesure peut être réalisée dans un but d'exploration, pour comprendre et étudier les profils de consommation du système de stockage ou de l'un de ses composants. On retrouve ainsi de nombreuses études ayant pour thème principal l'exploration de la consommation. Deuxièmement, les mesures de consommation peuvent être réalisées dans une étude pour valider un modèle de consommation, ou mesurer l'efficacité d'une proposition comme un nouveau système de stockage ou une optimisation ciblant l'économie d'énergie. On se concentre ici sur les études traitant de l'exploration de la consommation d'un système de stockage à base de mémoire flash comme sujet principal.

La mémoire flash étant une technologie relativement récente, de nombreuses études présentent des séries de mesures de consommation ciblant des systèmes de stockage à base de mémoire flash. Le but est de caractériser les profils de consommation de ces systèmes. De plus, certaines études poussent le travail plus loin en analysant les mesures dans le but d'identifier les éléments ayant un impact important sur la consommation, et les éléments dont l'impact est à l'inverse négligeable. La mise en évidence de ces éléments est un premier pas indispensable dans tout travail d'optimisation des performances ou de la consommation.

a) Au niveau d'une puce flash

Dans Grupp et coll. (2009), les auteurs présentent des résultats de mesures de consommation réalisées au niveau d'une puce flash NAND. Six modèles de puce SLC et cinq modèles de MLC ont été étudiés, ces modèles présentant des caractéristiques architecturales variées (taille d'une page, nombre

de pages par bloc, capacité totale, etc.). L'énergie consommée pendant les opérations de lecture et d'écriture de pages / d'effacement de blocs est mesurée. On peut ainsi observer que les mesures présentent des chiffres qui s'éloignent parfois de manière très significative des valeurs données sur les fiches techniques des différents constructeurs de puces flash. Les auteurs notent également que la question des latences variables en écritures pour certaines puces MLC a un impact important sur l'énergie consommée. Plus généralement, le coût en puissance et en énergie des accès MLC est supérieur aux accès SLC (Grupp et coll., 2009; Shin et Shin, 2010).

Les auteurs de Mathur et coll. (2009) analysent et comparent l'énergie des différentes opérations flash de base sur des puces flash NAND et NOR. On peut constater que la mémoire flash NOR présente en activité (lors de l'application d'opérations) un coût énergétique sensiblement plus important que la NAND, en particulier pour les opérations écritures.

b) Au niveau d'un disque SSD

Plusieurs études effectuent des mesures de consommation au niveau d'un SSD. Dans certaines études (Seo et coll., 2008; Shin et Shin, 2010; Bjørling et coll., 2010a; Yoo et coll., 2011), les auteurs mesurent la consommation lors de l'application de micro-benchmarks accédant directement au périphérique de type bloc qu'est le SSD. Les conclusions sont multiples.

Premièrement, on constate que la consommation à vide des SSD considérés dans ces études est loin d'être négligeable. La consommation à vide représente la consommation du système lorsqu'il n'est pas soumis à une charge d'E/S. Dans Shin et Shin (2010), les auteurs mesurent une puissance à vide de 0.6 W pour l'un des SSD étudiés, ce qui selon les auteurs correspond à 60 % de la consommation à vide d'un ordinateur de type *netbook*. Cette consommation à vide importante est due au fait que la consommation du SSD comprend entre autres la consommation du contrôleur qui embarque un processeur et potentiellement un cache de mémoire DRAM. De plus, en analysant la puissance mesurée au fil du temps lors de l'application des benchmarks, on constate que les SSD présentent des phases d'activité en arrière plan (Bjørling et coll., 2010a) : il s'agit d'opérations de gestion interne des métadonnées de la couche de gestion (par exemple l'exécution du ramasse-miettes), lancées lorsque le SSD ne reçoit pas de requêtes d'E/S.

Comme vue précédemment, l'organisation interne des puces flash au sein d'un SSD permet l'application d'opérations parallèles. Les auteurs de Shin et Shin (2010); Yoo et Park (2011) observent que la puissance mesurée dépend du nombre d'éléments mis en œuvre lors d'une opération sur le SSD. Plus le nombre de canaux et puces prenant part à l'opération est important, plus la puissance est élevée. Ainsi la méthode d'allocation des données à écrire dans les puces flash contenues dans le SSD détermine la consommation et les performances relatives à l'opération.

Plusieurs études rapportent (Bjørling et coll., 2010a; Shin et Shin, 2010) une énergie consommée lors des accès en écriture aléatoire beaucoup plus importante que pour tout autre type d'accès (écriture et lecture séquentielles, lectures aléatoires). Les performances (latences) se dégradent également lors d'écritures aléatoires. Ces études notent également que l'état de la mémoire flash au départ du benchmark conditionne de manière importante la consommation (et les performances) constatée pendant l'exécution du test. Un SSD contenant une importante quantité de données invalides présentera des performances réduites et une consommation plus importante par rapport à un SSD totalement effacé avant le test. Les explications derrière ces phénomènes se situent au niveau des algorithmes implémentés dans la FTL du SSD. On a vu précédemment que des mécanismes comme le mapping par bloc ou hybride génèrent sous une charge en écriture aléatoire un nombre très important d'opérations flash supplémentaires. Pour ce qui est de l'état de départ, un nombre important de données invalides force la couche de gestion à réaliser des opérations de ramasse-miettes pendant le test, ce qui augmente la consommation et provoque une baisse des performances.

Des études (Hui et coll., 2011; Seo et coll., 2008) analysent également la consommation de SSD en appliquant des macro-benchmarks : dans ces cas le SSD est formaté avec un système de fichiers

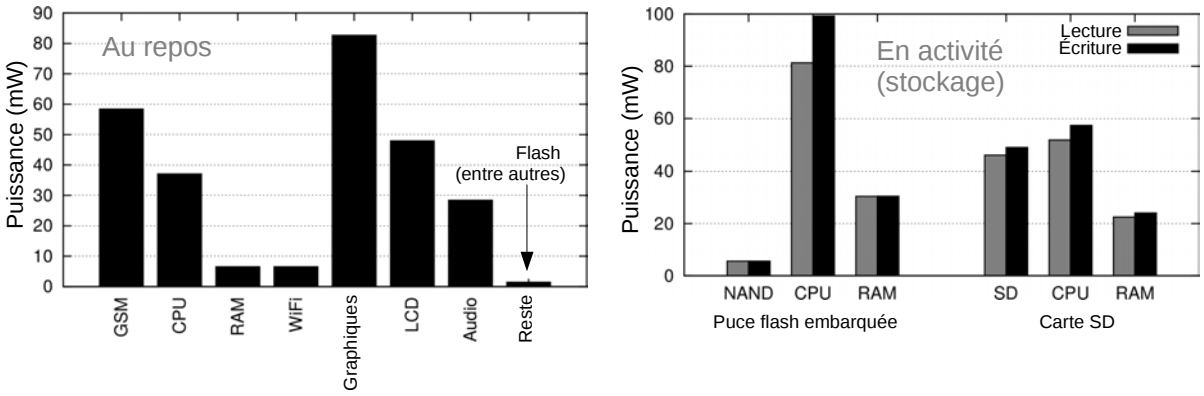


Figure 2.6 – Consommation des différents composants matériels d'un smartphone (modèle *Openmoko Neo Freerunner*) au repos (gauche) et des composants entrant en jeu lors d'opérations sur le système de stockage (droite). Image tirée de Carroll et Heiser (2010).

donné et reçoit des requêtes de la part d'une application ou d'un générateur synthétique. Les auteurs constatent que les caractéristiques de la charge d'E/S applicative déterminent la consommation et les performances : alors que les charges dominantes en lecture présentent une consommation plus faible et de meilleures performances que les charges en écriture, en particulier en écriture aléatoire.

c) Comparaison de la consommation des SSD et des disques durs

Les auteurs de Davis et Rivoire (2010); Lee et coll. (2009a); Schall et coll. (2010) comparent la consommation de systèmes de stockage à base de SSD avec celle de systèmes contenant des disques durs (HDD). Ces études sont réalisées dans le cadre de systèmes de stockage à large échelle, dans le contexte de bases de données sur des grappes RAID de plusieurs SSD ou HDD. Toutes les études montrent très clairement qu'en plus de proposer des performances supérieures aux HDD, les SSD présentent une consommation largement inférieure. Cela est aussi bien vrai pour la consommation au repos que pour la consommation en activité.

d) Domaine de l'embarqué

Alors qu'il existe de multiples études réalisant des mesures de consommation pour valider un modèle ou vérifier l'efficacité d'une optimisation / d'un nouveau système de stockage proposé (Mathur et coll., 2006; Tseng et coll., 2013; Choudhuri, 2003; Goyal, 2005; Song et coll., 2009), il existe relativement peu d'articles présentant une caractérisation de la consommation du stockage flash dans le domaine de l'embarqué.

Les auteurs de Carroll et Heiser (2010) présentent une série de résultats de mesures de consommation concernant les différents composants matériels composant un smartphone, contenant entre autres une puce flash gérée via un FFS. On peut y apprendre que, sur la plate-forme matérielle utilisée par les auteurs, la consommation de la puce flash NAND en elle-même est négligeable par rapport à celle du reste des composants, lorsque le système est au repos. Les auteurs de l'article appliquent des micro-benchmarks sur le système de stockage et l'on constate que les E/S génèrent une consommation non négligeable sur les composants RAM et CPU : en effet dans ce type d'environnement (FFS) ces composants sont mis à contribution dans l'exécution de la couche de gestion. La figure 2.6 présente des résultats obtenus dans cette étude. On peut voir sur la courbe de gauche (consommation au repos) que la consommation due à la puce de mémoire flash est au repos négligeable au niveau du système entier. En appliquant un micro-benchmark sur le système de stockage (lecture d'un fichier de 64 Mo par blocs de 4 Ko et écriture d'un fichier de 8 Mo par blocs de 4 Ko), on peut observer la consommation des

composants entrant en jeu dans le traitement des transferts d'E/S. Deux types de systèmes de stockage à base de mémoire flash sont étudiés : une puce flash brute, et une carte SD.

Dans Nguyen (2013), les auteurs s'intéressent à la consommation globale d'un smartphone de type *Android* lors de l'application de plusieurs macro-benchmarks sur le stockage secondaire (puce *eMMC*). Les auteurs varient les politiques d'ordonnancement des entrées / sorties ainsi que la taille de la file d'attente au niveau du pilote eMMC qui détermine le nombre maximal de requêtes d'E/S en attente. Ces deux paramètres ont un impact sur la consommation et les auteurs concluent qu'en fonction des caractéristiques de la charge d'E/S, certaines politiques / tailles de files sont plus efficaces que d'autres du point de vue de la consommation.

2.4 Conclusion concernant l'exploration de la consommation et des performances des systèmes de stockage à base de mémoire flash

Concernant les performances, on s'est uniquement intéressé au domaine des FFS car l'étude des performances des systèmes de stockage à base de mémoire flash est un sujet très large. On retient deux types d'études. On observe premièrement des études comparatives utilisant des benchmarks pour comparer des types de FFS existants. Deuxièmement, les articles présentant de nouvelles proposition de FFS effectuent généralement une analyse des performances du système proposé (parfois via une étude comparative avec des FFS existants). Différentes métriques sont utilisées dans le cas de l'étude des performances de FFS : on peut noter le temps de montage, le temps d'exécution d'un benchmark ou d'une opération à granularité plus fine (par exemple un appel système), ou encore les débits en lecture et écriture. On peut également observer des métriques comme l'empreinte RAM, le coût de stockage des méta-données, ou l'efficacité de la compression pour les FFS qui la supporte. Enfin, on observe des métriques relatives à la répartition de l'usure comme la distribution des effacements, le nombre d'effacements réalisés pendant un test, la répartition des données chaudes et froides, et l'amplification en écriture.

Concernant la consommation, on peut premièrement noter que les SSD sont un sujet d'étude très important. Cela est logique au vu de l'explosion sur le marché de ce type de périphérique : les auteurs de Wong (2014) présente l'évolution des revenus du marché des SSD pour PC et tablettes, de 600 millions de dollars en 2009 vers une prévision de 7 milliards en 2015. De plus les études prouvent que les SSD apportent, en plus d'un gain de performances, une économies d'énergie non négligeable par rapport aux disques durs. La consommation des SSD est due aux composants internes qui sont le contrôleur (processeur embarqué), l'éventuel cache de DRAM, et l'ensemble des puces de mémoires flash elles-mêmes. La consommation des puces est particulièrement impactée par l'utilisation potentielle du parallélisme inhérent à la structure interne des SSD. Pour ce qui est des systèmes embarqués, il semble ne pas exister d'étude explorant en détail la consommation dans le contexte des systèmes de stockage embarqués à base de FFS.

Les études présentées ici mettent en évidence les différents paramètres impactant la consommation et les performances dans les systèmes de stockage à base de mémoire flash :

1. La *charge d'E/S* subie par le système ainsi que ses caractéristiques (type et taille de requêtes, motif d'accès séquentiel ou aléatoire) ;
2. Les algorithmes implémentés par la *couche de gestion* déterminent les accès flash et l'utilisation des composants matériels CPU (contrôleur) et RAM (caches) du système de stockage. Ce sont également ces implémentations de couches de gestion qui définissent l'application d'opérations en arrière plan et donc une partie de la consommation du système de stockage au repos ;
3. L'état du système à un moment donné détermine également la consommation et les performances d'opérations appliquées au système de stockage à ce moment précis : en particulier, la quantité de données invalides ou d'espace libre peut forcer le lancement du ramasse-miettes

qui impacte la latence et la consommation des E/S reçues par le système de stockage durant son exécution.

Dans le chapitre suivant, traitant de l'exploration des performances et de la consommation des FFS, on confirme l'impact de ces éléments dans le contexte de ces systèmes de fichiers. La modélisation de cet impact est présentée au chapitre 4.

On peut noter que la couche de gestion joue un rôle central dans ces trois paramètres influant sur la consommation des systèmes de stockage à base de mémoire flash : en effet c'est la couche de gestion qui traite la charge d'E/S applicative et effectue des accès flash. C'est également la couche de gestion qui, au fil des écritures, détermine l'état du système de stockage. Les couches de gestion au sein des SSD (FTL) sont des boîtes noires propriétaires au sujet desquelles il est difficile d'obtenir des informations. C'est pour cela que plusieurs des études présentées ici se contentent d'observer la consommation et de tenter d'expliquer les phénomènes constatés. Les FFS sous Linux sont à l'inverse ouverts, l'accès aux sources permettant une étude fine des algorithmes implémentés pour pouvoir expliquer en détail les comportements de consommation et de performances. Dans les modèles présentés au chapitre 4, la couche de gestion (via l'exemple du FFS JFFS2) est modélisée dans les détails.

Finalement, on peut dire que la phase d'exploration est indispensable à tout travail de modélisation. Comme vu dans Grupp et coll. (2009), les modèles basés sur des mesures réelles sont beaucoup plus précis car des différences importantes peuvent exister entre les chiffres donnés sur les fiches techniques (ou toute autre source) et les mesures en environnement réel. Les modèles présentés au chapitre 4 sont basés sur des mesures.

3 Modélisation des performances et de la consommation énergétique

Dans cette section, on présente les différents travaux réalisés dans le contexte de la modélisation des performances et de la consommation des systèmes de stockage à base de mémoire flash NAND. Les modèles sont des ensembles d'équations et / ou d'algorithmes construits pour estimer les performances ou la consommation du système ou du sous-système de stockage modélisé. Une fois les modèles construits, leur utilisation présente l'avantage de simplifier les opérations de récolte d'information concernant les performances et la consommation :

1. Pour des raison de *coût* : acheter du matériel pour réaliser les mesures et du matériel sur lequel les effectuer peut être très coûteux ;
2. Pour des raison de *temps* : la mise en place d'un environnement de mesure peut être long et fastidieux.

La précision du modèle peut potentiellement contrebalancer ces avantages : elle indique la fidélité des résultats obtenus à partir des modèles par rapport à ce qui se passe effectivement en environnement réel. Les modèles de performances et de consommation sont très utilisés dans le domaine des systèmes de stockage pour obtenir des estimations de manière non coûteuse et rapide. Ils le sont également pour estimer l'efficacité de propositions comme des nouveaux systèmes de stockage ou des optimisations de systèmes existants, dans certains contextes où la mise en place de larges campagnes de mesure est trop coûteuse en temps ou argent.

De nombreuses études présentent des modèles de performances et de consommation pour les systèmes de stockage à base de mémoire flash. Dans certains articles le modèle est le sujet principal de l'étude, dans d'autres, le modèle est construit et utilisé dans le cadre de l'étude, par exemple afin de valider une optimisation. Dans certains articles les auteurs implémentent des modèles au sein d'un simulateur. Selon la précision de la description du simulateur, il n'est pas toujours aisé d'extraire des informations au sujet des modèles implémentés dans ledit simulateur.

Pour caractériser les modèles de performances et de consommation étudiés ici, on se base sur les attributs illustrés sur la figure 2.7 :

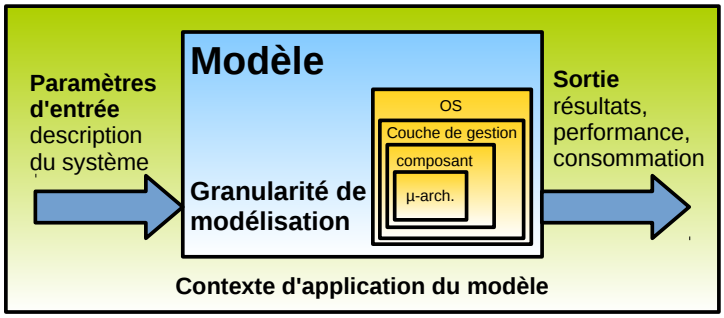


Figure 2.7 – Caractéristiques générales des modèles de performances et de consommation étudiés

— La *granularité de modélisation* adoptée. Certains modèles décrivent en détail les paramètres micro-architecturaux d'une mémoire flash (par exemple dans Mohan et coll. (2010)), d'autres ciblent un niveau d'abstraction bien plus élevé comme une structure multi-puces complexe telle que celle que l'on retrouve au sein d'un SSD (par exemple dans Hu et coll. (2011)) ;
— Le *contexte d'application* du modèle. Des modèles généralistes sont destinés à être utilisés pour estimer les performances et / ou la consommation de systèmes de stockages à base de flash de manière générale (dans Jung et coll. (2012) par exemple). D'autres modèles très ciblés sont développés dans le cadre d'une étude spécifique, pour vérifier l'efficacité d'une optimisation ou d'un nouveau système de stockage. Ces derniers ne sont alors parfois pas destinés à être utilisés dans un autre cadre que l'étude en question (par exemple dans Tseng et coll. (2006)). A noter que le contexte d'application du modèle peut être différent du niveau de granularité de modélisation ;
— Les *paramètres d'entrée* du modèle, c'est à dire la manière dont l'utilisateur décrit le système au sujet duquel il souhaite obtenir des estimations ;
— Les *sorties* du modèles : les différentes métriques de performances et de consommation estimées.

Ici nous avons fait le choix de classifier les modèles par granularité de modélisation, on retient les niveaux suivants :

— Le niveau *micro-architectural*, niveau de granularité le plus fin ;
— Le niveau *puce*, dans lequel une ou plusieurs puces de mémoire flash est (sont) considérée(s), sans couche de gestion ;
— Le niveau *couche de gestion + flash*. On est ici au niveau d'un périphérique de stockage à base de mémoire flash, comme une clé USB ou un SSD. Dans ce dernier cas une architecture de stockage potentiellement complexe (plusieurs puces) est présente. La couche de gestion FTL est prise en compte, ainsi que l'impact potentiel d'un éventuel cache de mémoire vive embarqué, par exemple au sein d'un SSD ;
— Le niveau *système d'exploitation*, dans lequel on considère le système de stockage à base de mémoire flash ainsi que ses interactions avec le système d'exploitation (en particulier le système de fichiers et les caches du système d'exploitation).

3.1 Modèles niveau micro-architectural

FlashPower (Mohan et coll., 2010, 2013; Mohan, 2010) est un modèle de consommation au niveau micro-architecture, ciblant les mémoires flash NAND de type SLC et MLC. FlashPower s'appuie sur CACTI (*Cache Access and Cycle Time*, présenté dans Wilton et Jouppi (1996)), un outil présentant un ensemble de modèles pour calculer la consommation et les performances au niveau micro-architecture de composants mémoire en général. FlashPower prend en entrée un certain nombre de paramètres décrivant la micro-architecture de la puce de mémoire flash concernée (par exemple la taille d'une page,

le nombre de pages par bloc, etc.), ainsi que des entrées concernant les latences de diverses opérations, des valeurs de tension pour divers sous-composants de l'architecture décrite, et une description de la charge d'E/S appliquée à la puce. FlashPower calcule l'énergie consommée par la puce. Au niveau micro-architecture, on peut également citer NVSim (Dong et coll., 2012), ciblant la consommation et les performances de plusieurs types de mémoires émergentes : ReRAM, CPRAM, STT-RAM. Il supporte également la mémoire flash NAND.

Les problèmes de fiabilités et d'endurance des mémoires flash NAND étant bien connus, on peut également citer des études (Jung et coll., 2008b; Mohammad et Saluja, 2001) qui proposent des modèles de fautes relatifs aux différents types de dysfonctionnement d'une mémoire flash NAND, au niveau micro-architectural.

3.2 Modèles niveau puce

Les auteurs de Ross (2008) présentent un modèle de performances nommé *EWOM* (Erasable Write Once Memory). Ce modèle est spécifiquement dédié à l'étude de la contrainte d'effacement avant écriture des mémoires flash. EWOM modélise le coût de différentes stratégies d'écritures en mémoire flash, en prenant en compte les différentes contraintes relatives à l'opération d'écriture : la granularité de l'opération (une page entière ou la possibilité d'effectuer une écriture partielle), l'impossibilité de mettre à jour une page contenant déjà des données, et le fait qu'une page invalidée sera finalement effacée à un moment donné. Ce modèle calcule les quantités d'opérations de lecture / écriture / effacements flash réalisées, permet d'explorer l'efficacité de différentes stratégies d'écritures pour couches de gestion flash.

Au niveau des modèles de consommation, dans Doh et coll. (2009) les auteurs évaluent l'efficacité concernant la réduction de l'énergie consommée d'une proposition de cache de mémoire flash pour les méta-données d'un système de fichiers. Les auteurs de cette étude utilisent un modèle de consommation simple proposé dans Du et coll. (2005), qui calcule la consommation de la puce flash en fonction du nombre d'opérations et de l'énergie consommée pendant une opération unitaire. Les valeurs d'énergies sont tirées de fiches techniques de puces flash. Dans Ji et coll. (2008), les auteurs proposent un modèle permettant d'estimer la puissance moyenne au niveau de la mémoire flash lors de l'application d'une charge de travail donnée. Ce modèle prend en entrée des valeurs de puissances réalisées pendant chacun des types d'opérations flash unitaires, et une représentation de la charge de travail sous la forme de pourcentages d'opérations de lecture / écriture / effacement. Chaque valeur de puissance est alors multipliée par le pourcentage correspondant pour obtenir la puissance moyenne. Dans Pallister et coll. (2014), les auteurs proposent également un modèle de consommation assez spécifique au contexte de l'exécution de code en place dans de la mémoire flash NAND.

NandFlashSim de Jung et coll. (2012) est un modèle généraliste de performances et de consommation pour une puce ou un ensemble de puces flash tel que présent dans les SSD. NandFlashSim permet de décrire dans les détails une architecture de stockage via de multiples paramètres comme la taille d'une page, le nombre de pages par bloc, le nombre de puces et leur organisation en canaux, etc. Il s'agit d'un modèle au cycle près (*cycle-accurate*), qui prend en entrée une série de commandes flash, des paramètres relativement précis sur les latences des phases composant une opération flash de base (envoi des commandes, adresses, envoi / réception des données), et des paramètres de tension et intensité relatifs à la consommation électrique des puces décrites. NandFlashSim supporte les puces SLC et MLC, en particulier les variations de latences en écritures associées à ces dernières. NandFlashSim donne des résultats sous forme de débits, latences des commandes flash, et énergie consommée.

3.3 Modèles niveau couche de gestion + flash

Les modèles présentés à ce niveau sont très nombreux. En effet, c'est à partir de ce niveau de modélisation que la couche de gestion est prise en compte. Or, on a vu précédemment que la couche

de gestion est l'élément central déterminant les performances et la consommation d'un système de stockage à base de mémoire flash. On retrouve donc à ce niveau, en plus des modèles de consommation et de performances, des modèles fonctionnels décrivant les algorithmes des couches de gestion. Ces modèles fonctionels interragissent avec les modèles de performances et de consommation, représentant l'impact de la couche de gestion sur ces métriques.

On peut ici différencier deux catégories d'études : (A) les études qui présentent des modèles de consommation et de performances pour valider un nouveau système de stockage ou une optimisation et (B) les études dont le thème principal est l'ensemble des modèles présentés. Dans le premier cas, les modèles sont relativement spécifiques au système proposé et ne sont en général pas utilisables dans un autre contexte. Dans le second cas, il s'agit de modèles généralistes dont le but est d'être réutilisés par la communauté scientifique. Ils sont généralement implémentés dans des simulateurs, qui seront exposés dans la section suivante.

a) Modèles spécifiques au contexte d'une étude

Plusieurs études (Kgil et Mudge, 2006; Kgil et coll., 2008; Chen et coll., 2006; Ou et Härder, 2011) proposent des solutions qui consistent à introduire la mémoire flash dans la hiérarchie mémoire entre la mémoire principale et un système de stockage secondaire de type disque dur, sous la forme d'un système de cache. Les bénéfices apportés sont multiples :

— Il est possible de réduire la quantité totale de mémoire vive et donc les coûts du système (le prix au bit de la flash étant largement inférieur à celui de la RAM), ainsi que la consommation au repos due au rafraîchissement de la mémoire RAM ;
— Les accès disque sont réduits et les performances augmentées. La consommation relative à ces accès est également réduite ;

FlashCache (Kgil et Mudge, 2006; Kgil et coll., 2008) est un système qui combine de la mémoire flash NAND et une quantité réduite de RAM. Son but est de remplacer la mémoire principale dans un système informatique de type serveur, pour minimiser en particulier la consommation au repos due au rafraîchissement de la RAM. Un modèle de performance d'un système implémentant FlashCache est proposé par les auteurs pour évaluer l'efficacité de la solution proposée. Ce modèle prend en entrée des valeurs de latences et de consommation pour les opérations flash de base, ainsi qu'une trace représentant la charge de travail appliquée au modèle. Le modèle implémente également les algorithmes de fonctionnement de FlashCache. Le modèle permet de calculer la puissance moyenne constatée pendant l'application de la charge de travail, ainsi que diverses métriques de performances comme la bande passante du serveur modélisé et la durée de vie (endurance) du système de stockage à base de mémoire flash. Dans Ou et Härder (2011), les auteurs proposent un système relativement similaire. *SmartSaver* (Chen et coll., 2006) propose l'introduction d'un cache de mémoire flash pour minimiser les accès au disque dur dans un système informatique et profiter du système de DPM (*Dynamic Power Management*) du disque pour économiser de l'énergie. Le DPM est une politique qui permet à un système de se mettre dans un état de basse consommation (veille) lorsqu'il n'est pas soumis à une charge de travail. Pour évaluer l'efficacité de SmartSaver, les auteurs utilisent un modèle de consommation pour le disque dur et la mémoire flash. Le modèle de consommation de la mémoire flash est très simple, il prend en entrée deux valeurs de puissances qui correspondent aux puissances consommées par la flash en activité et au repos. Une trace représentant la charge de travail est également prise en entrée par le modèle, qui calcule, entre autres, l'énergie consommée par le système.

Dans le domaine des systèmes embarqués, les disques durs sont très peu présents et la mémoire flash fait généralement office de stockage secondaire. Dans ce contexte, plusieurs études (Park et coll., 2004; Tseng et coll., 2006; Li et coll., 2008) proposent des couches de gestion flash qui peuvent être vues comme des FTL simples ciblant la consommation via la réduction de l'utilisation de la mémoire principale. Des modèles de performance et de consommation sont utilisés pour valider l'efficacité

des solutions. Utilisés dans des simulateurs implémentant les algorithmes de gestion proposés dans les études respectives, ces modèles prennent en entrée des valeurs d'énergie et de latence pour les opérations flash de base, ainsi que pour les opérations en mémoire principale. De plus, les auteurs de Park et coll. (2004) présentent également une modélisation simple de la consommation du processeur, prenant en entrée les valeurs de puissance constatée lorsque le processeur est en activité et au repos. Ces modèles prennent également en entrée des traces représentant la charge d'E/S, et donnent en sortie des estimations concernant l'énergie consommée et le temps d'exécution.

Dans Lee et Kim (2010), les auteurs présentent un modèle de consommation et de performances pour évaluer une proposition d'algorithme de DVFS (*Dynamic Voltage and Frequency Scaling*) pour contrôleur de SSD. Une politique de DVFS définit pour un dispositif de calcul des couples tension / fréquence sur lesquels le dispositif peut s'ajuster de manière dynamique pour pouvoir fonctionner à pleine puissance en période de charge, et passer dans un état de basse consommation en période creuse. Un modèle de performances et de consommation est construit pour évaluer l'efficacité de l'algorithme proposé. Ce modèle se base sur des valeurs de latence et de consommation des différents paliers de l'algorithme DVFS proposé.

b) Modèles généralistes

Les modèles généralistes sont destinés à être utilisés par la communauté scientifique et industrielle. Ils sont généralement implémentés et distribués au sein de simulateurs pour une utilisation plus aisée. Ce sont pour la majorité d'entre eux des modèles concernant des SSD, avec pour la plupart la possibilité de modéliser des architectures multi-puces complexes.

Dans El Maghraoui et coll. (2010), les auteurs proposent un modèle de performance pour SSD très simple, indépendant de la couche de gestion utilisée et des spécificités de la mémoire flash NAND. Le modèle concerne les accès au niveau bloc d'un SSD, et détermine le débit des opérations de lecture / écriture séquentielles et aléatoires. Pour chacune des opérations le débit est modélisé de manière suivante :

$$D_{op}(n) = \frac{n}{A_{op} + n * B_{op}} \tag{2.6}$$

$D_{op}(n)$ est le débit d'une opération op classée comme *lecture séquentielle*, *lecture aléatoire*, *écriture séquentielle* ou *écriture aléatoire*, de taille n octets. A_{op} et B_{op} représentent des coefficients déterminés par des expériences et une technique de régression linéaire. On peut voir ces coefficients comme un profil de performances pour un système de stockage donné recevant un certain type d'opération. La validation face à un SSD montre que l'erreur d'un tel modèle reste inférieure à 10%.

Comme dit précédemment, bon nombre de modèles généralistes sont implémentés dans des simulateurs. Une partie de ces simulateurs implémentent des modèles relativement complexes de couche de gestion, qui seront vus dans la section suivante traitant des simulateurs. Concernant les modèles de performance et de consommation relatifs à ces simulateurs, on retrouve les modèles utilisant en entrée des valeurs simples pour les latences / consommation des opérations flash de base (Caulfield et coll., 2009; Kim et coll., 2009b). Certains modèles prennent en entrée des paramètres plus détaillés : par exemple les auteurs de Hu et coll. (2011) détaillent les temps et consommation des différentes phases constituant les cycles d'envoi des commandes, adresses, et données appliqués aux puces flash du SSD. Tous ces modèles prennent en entrée de nombreux paramètres concernant l'architecture du SSD simulé comme la taille d'une page, le nombre de pages par bloc, le nombre de puces et leur organisation dans des canaux, etc. Le modèle fonctionnel de couche de gestion implémenté par le simulateur est utilisé pour transformer une trace d'E/S en entrée en accès flash, qui sont alors eux mêmes utilisés pour calculer les performances et la consommation.

3.4 Modèles niveau système d'exploitation

Ces modèles concernent les systèmes embarqués et plus particulièrement Linux gérant une puce de mémoire flash embarquée, qui est la plate-forme qui nous intéresse dans le cadre de ce travail de thèse.

Dans Goyal et Mahapatra (2005) et Choudhuri (2003), les auteurs proposent une modélisation simple et haut niveau de la consommation de systèmes de fichiers gérant une puce flash brute, en particulier le FFS JFFS2 sous Linux. Les auteurs proposent un modèle au sein duquel une équation de calcul de consommation (énergie) est associée à chacun des appels systèmes Linux relatifs au système de fichiers (*read()*, *write()*, *open()*, etc.). Ces appels systèmes sont divisés en deux groupes : (A) ceux qui font intervenir une quantité de données transférées (majoritairement *read()* et *write()*) ainsi que (B) le reste des appels systèmes relatifs au système de fichiers, par exemple *rename()*, *mkdir()*, etc. La première partie voit sa consommation modélisée via une équation linéaire, par exemple :

$$E_{read()}(n \text{ octets}) = A * n + B \tag{2.7}$$

Les appels système qui ne font pas intervenir de quantité de données (par exemple *open()*, *chown()*, etc.) voient leur consommation modélisée comme des constantes, par exemple :

$$E_{open()} = A \tag{2.8}$$

Les inconnues dans les équations sont déterminées via des expériences et une méthode de régression linéaire. Ces paramètres sont déterminés pour le CPU et pour la mémoire flash. On peut donc voir les termes A et B dans les équations ci-dessus comme un profil de consommation relatif à un couple (plate-forme matérielle, système de fichiers) donné. A noter que rien n'est fait concernant la RAM, qui est utilisée de manière importante lors des accès aux fichiers.

3.5 Conclusion concernant les modèles de performances et de consommation pour systèmes de stockage à base de mémoire flash

Pour conclure cet état de l'art sur les modèles de performances et de consommation pour système de stockage à base de mémoire flash, on peut noter plusieurs choses.

Premièrement, on peut voir qu'il existe différents niveaux de granularité de modélisation, allant du niveau micro-architectural au niveau système d'exploitation. La figure 2.8 page 68 tente de situer les différentes études présentées dans ce chapitre suivant le niveau de granularité de la modélisation présentée. On peut noter le nombre très important de modèles présentés au niveau couche de gestion. Cela s'explique majoritairement par le fait que, comme énoncé précédemment, la couche de gestion ait un impact très important sur les performances et la consommation, et il est important de la prendre en compte dans la modélisation de ces métriques.

On retrouve également des modèles à niveau de granularité de modélisation fin : le niveau micro-architectural, et dans une moindre mesure le niveau puce. Alors que ces modèles (par exemple le travail présenté dans Mohan (2010)) permettent d'estimer de manière relativement détaillée les métriques de consommation et de performances, ce niveau de granularité de modélisation ne se prête pas à des estimations au niveau du système informatique, cadre de travail de cette thèse (domaine des FFS). Premièrement la couche de gestion n'est pas prise en compte. Les entrées (charges d'E/S) de ce type de modèle se situent à des niveaux très éloignés de la charge applicative, ou du niveau bloc (entrée des périphériques de type FTL comme les SSD). Finalement, même s'il est possible de rajouter sur ces modèles à granularité fine des modèles représentant les couches de gestion, on peut imaginer que le temps de calcul nécessaire pour exécuter une charge de travail applicative complexe est potentiellement très important, au vu de la finesse de la modélisation à ce niveau.

Enfin, à large niveau de granularité de modélisation, on retrouve des modèles comme celui présenté dans Goyal et Mahapatra (2005). Ils présentent l'avantage d'être simples et de se situer dans le cadre du

travail de cette thèse. Néanmoins on peut considérer que leur simplicité est également un point faible. En effet ces modèles effectuent une abstraction de la couche de gestion, masquant les complexités introduites par des notions tels que les mécanismes de ramasse-miettes, l'état de la flash (la quantité d'espace libre / valide / invalide), et d'autres concepts spécifiques aux différents modèles de FFS. Comme on le verra dans la phase d'exploration de ce travail de thèse au chapitre suivant, ces éléments ont un impact non négligeable sur les performances, et l'on est en droit de questionner la précision de ce type de modèle haut niveau dans le cadre de charges de travail complexes et à large échelle.

Par opposition aux modèles généralistes, on peut noter le nombre important d'études qui utilisent des modèles, en particulier de consommation, pour valider des propositions d'optimisations ou de nouveaux systèmes de stockage à base de mémoire flash. Cela peut s'expliquer par le fait qu'il est parfois difficile de réaliser des mesures de consommation sans qu'il y ait un coût important en temps ou financier. Il est également parfois difficile de réaliser des mesures de consommation précises d'un sous composant d'un système de stockage (ou d'un système informatique complet), en fonction des points de mesures disponibles dans le système.

Finalement, on peut noter qu'il n'existe aucun modèle satisfaisant dans le contexte du travail de cette thèse, c'est à dire la consommation et les performances des systèmes de fichiers dédiés aux mémoires flash sous Linux embarqué.

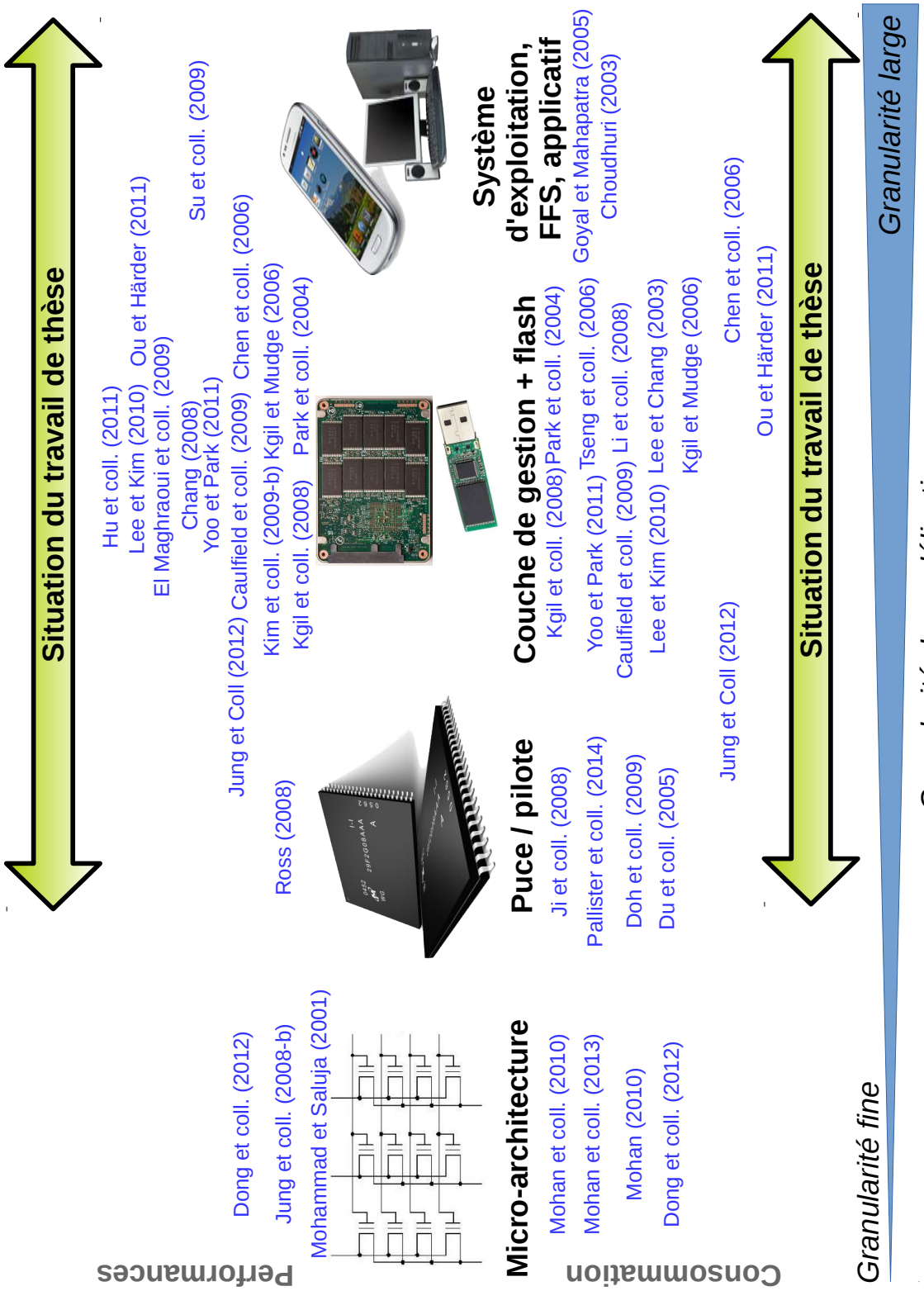


Figure 2.8 – Classification selon la granularité de modélisation de différents articles présentant des modèles de performance et de consommation pour systèmes de stockage à base de mémoire flash

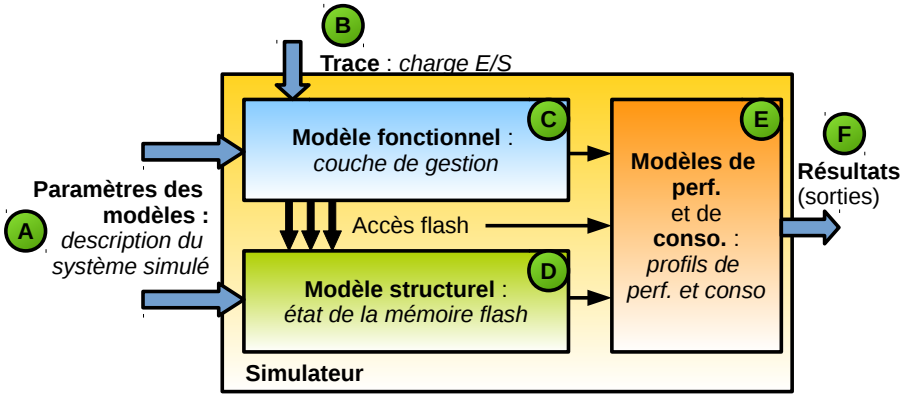


Figure 2.9 – Fonctionnement général d'un simulateur pour système de stockage à base de mémoire flash

4 Simulateurs de systèmes de stockage à base de mémoire flash

Tout comme pour les modèles, certains simulateurs sont développés spécifiquement dans le cadre d'une étude pour évaluer ou valider l'efficacité d'une solution, alors que d'autres simulateurs généralistes sont développés pour être utilisés par la communauté scientifique. Ici on s'intéresse à la deuxième catégorie, les simulateurs généralistes. Les simulateurs pour systèmes de stockage sont utilisés pour implémenter les modèles de performances et de consommation, permettant de jouer ces modèles de manière aisée. De plus, on a vu que la couche de gestion définissait les accès réalisés en flash à partir de la charge d'E/S applicative. De nombreux simulateurs prennent en compte cette couche de gestion (FTL pour la plupart des simulateurs) en intégrant des algorithmes décrivant le fonctionnement de la couche de gestion en question.

Le fonctionnement général d'un simulateur de système de stockage à base de mémoire flash est illustré sur la figure 2.9. Bien entendu, le fonctionnement de l'intégralité des simulateurs ne peut être résumé aussi simplement, et certains simulateurs fonctionnent de manière assez différente de celle présentée ici. Néanmoins, il s'agit là d'une abstraction du fonctionnement de la majeure partie des simulateurs présentés dans cet état de l'art. Les simulateurs prennent en entrée une description du système à simuler (A sur la figure 2.9), qui correspond aux paramètres des différents modèles implémentés dans le simulateur. En entrée on retrouve également une trace (B) représentant la charge d'E/S appliquée au système de stockage à simuler. Cette trace est traitée par une implémentation d'un modèle fonctionnel (C) représentant la couche de gestion flash. Cette couche de gestion calcule les accès flash réalisés, et met à jour l'état du système simulé. Il peut s'agir de l'état de la flash (état valide / invalide / libre des pages, D sur la figure 2.9), mais également d'autres composants comme par exemple la présence de données dans un buffer de DRAM dans le cas de la simulation d'un SSD. Les modèles de performances et de consommation (E) sont alors utilisés pour calculer des valeurs concernant les métriques qui sont les sorties du simulateur (F). Cela est fait en fonction de la charge d'E/S, d'éventuels évènements internes calculés au niveau de la couche de gestion, et de l'état du système.

On présente ici plusieurs propositions de simulateurs de systèmes de stockage à base de mémoire flash que l'on peut trouver dans la littérature. On s'attache à déterminer pour chacun des simulateurs les caractéristiques suivantes :

— Le *contexte d'utilisation* du simulateur. Il s'agit de la réponse à des questions générales telles que le but recherché lors de l'utilisation du simulateur, le type de système simulé, etc ;
— La *granularité* des modèles implémentés dans le simulateur, telle que décrite dans la section précédente ;
— Les différents paramètres passés en entrée, et les différentes métriques disponibles en sortie. La plupart des simulateurs ciblent les performances, et certains proposent des métriques de consommation en sortie. En entrée les simulateurs prennent une description du système à

simuler et de la charge de travail, et en sortie généralement on retrouve diverses statistiques plus ou moins détaillées sur les performances et éventuellement la consommation du système traitant la charge ;
— Les caractéristiques de la charge de travail passée en entrée ;
— La présence ou l'absence du support d'architecture multi-puces avancées ou encore de commandes flash avancées telles que décrites dans le premier chapitre.

On présente ici un total de 10 simulateurs (Microsoft Research, 2009; Gupta et coll., 2009; Kim et coll., 2009b; Hu et coll., 2011; Jung et coll., 2012; MTD Contributors, 2008; Dayan et coll., 2013; Su et coll., 2009; El Maghraoui et coll., 2010; Yoo et Park, 2011). Alors que la plupart sont utilisés pour l'exploration de l'espace de conception en variant les différents paramètres relatifs à l'architecture de stockage et à la couche de gestion simulée, certains proposent également des infrastructures pour développer de manière plus ou moins aisée de nouvelles couches de gestion (FTL) dans le simulateur.

Add-on Microsoft pour le simulateur DiskSim Il s'agit là de l'un des premiers simulateurs pour système de stockage à base de SSD. Le système simulé est ici un SSD. Le simulateur (Microsoft Research, 2009) est réalisé en adaptant le populaire simulateur de systèmes de stockage à base de disque dur DiskSim (Bucy et coll., 2008), un simulateur à évènements discrets écrit en language C. La contribution de Microsoft Research (2009) est donc d'ajouter à DiskSim la possibilité de simuler des SSD.

L'utilisation de DiskSim comme infrastructure pour ce simulateur de SSD lui permet de profiter de plusieurs avantages. Premièrement, cela permet de profiter de tous les modèles de mécanismes satellites implémentés par DiskSim : bus, caches, files d'attentes, etc. Cela permet également de ne pas avoir à ré-implémenter des mécanismes fondamentaux des simulateurs à évènements discrets tels que la gestion du temps et des évènements. Enfin, DiskSim propose un générateur synthétique de traces niveau bloc qu'il est possible d'utiliser en simulant un SSD.

Ce simulateur ne propose qu'un seul modèle de FTL idéale (à base de page-mapping) relativement peu configurable, et ne supporte qu'un seul type d'opération avancée : la lecture et l'écriture en mode cache. Il propose néanmoins la simulation d'architectures de stockage complexes, contenant plusieurs puces et canaux. De plus, il est possible de simuler plusieurs SSD. On peut donc conclure que ce simulateur est principalement dédié à l'exploration de l'espace de conception d'un système de stockage à base de SSD. Les sources de cet outil sont disponibles sur internet [1], sous licence *Microsoft Research License Agreement* (utilisation non commerciale autorisée). L'outil n'a pas été mis à jour depuis 2009 et est relativement peu utilisé. Ce simulateur cible les performances. Il prend en entrée des valeurs de latences non pas pour les opérations de bases flash, mais plutôt des sous-opérations les constituant : les temps de transferts entre la matrice de transistor NAND et le *page buffer*, et les temps de transfert sur le bus d'E/S.

FlashSim version C *FlashSim* (Gupta et coll., 2009) est également une extension du simulateur DiskSim (donc toujours en C). Il existe une autre version de FlashSim en C++ qui sera présentée dans la sous-section suivante. FlashSim version C à été proposé en 2009 dans un article proposant un système de FTL nommé DFTL (Gupta et coll. (2009), *Demand-based Flash Translation Layer*). FlashSim est un simulateur utilisé dans de nombreuses études proposant de nouveaux systèmes de FTL, pour les raisons suivantes :

— L'infrastructure de développement isole de manière relativement efficace, et identifie clairement l'endroit du code de l'outil au niveau duquel il faut intervenir pour implémenter une proposition d'algorithme de couche de gestion ;
— Dans cette version de FlashSim sont déja implémentés quelques algorithmes de FTL reconnus, il est donc aisé pour l'utilisateur de comparer sa proposition de couche de gestion avec l'état

1. L'add-on Microsoft pour DiskSim est disponible ici : `http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/` [accédé le 6 octobre 2014]

de l'art.

Basée sur DiskSim, cette version de FlashSim profite également des avantages précédemment cités. Les paramètres en entrée du simulateur sont relativement simples, il s'agit de paramètres architecturaux / de latences concernant la mémoire flash à modéliser. FlashSim ne supporte la modélisation que d'une seule puce, et ne supporte pas les commandes avancées.

Les sources de cet outil sont disponibles sur internet [2]. Aucune licence n'est indiquée.

FlashSim version C++ Toujours en 2009 les auteurs de FlashSim ont proposé une nouvelle version du simulateur, cette fois en C++ (Kim et coll., 2009b). Cette version apporte diverses améliorations. Premièrement le couplage avec DiskSim a quasiment disparu, FlashSim est maintenant un outil à part entière. Il est néanmoins toujours possible de prendre une trace générée par DiskSim en entrée, pour profiter du générateur synthétique et des modèles de composants plus haut niveau (caches, files, etc.) fournis par le simulateur pour disques durs.

FlashSim version C++ contient une description du système de stockage orientée objet. D'une part cela permet la simulation d'architecture hiérarchiques de stockage multi puces / canaux complexes, et d'autre part cela permet le développement aisé de nouveaux algorithmes de FTL au sein du simulateur via des mécanismes de la programmation objets comme l'héritage et l'isolement des traitements et données dans des classes et objets. Cette version de FlashSim cible les performances. Même si des résultats sur la consommation sont présentés dans l'article présentant l'outil (Kim et coll., 2009b), il n'est pas possible de retrouver référence à des métriques de consommation dans le code. Il est probable que les auteurs aient effectué une analyse de la consommation à partir des résultats de performances obtenus via le simulateur.

Un désavantage de ce simulateur est que contrairement à la version en C, il n'est pas fourni avec des modèles de FTL pré-implémentés. Les sources de cet outil sont disponibles sur internet [3], sous licence GPL. La dernière version date de 2010.

SSDSim Les simulateurs présentés précédemment ne proposent pas de modèle complet pour l'intégralité des commandes flash avancées. On rappelle que ces commandes sont le mode cache, le copy-back, les opérations dé / LUN entrelacées, et les opérations multi-canaux. *SSDSim* (Hu et coll., 2011) propose un support complet pour ces opérations avancées, ainsi que le support pour une architecture de stockage complexe. SSDSim est livré avec des modèles correspondant à plusieurs algorithmes de FTL populaires.

SSDSim est écrit en C, et permet de décrire de manière détaillée les performances, sous la forme de latences des opérations des puces de mémoire flash simulées. Ils prend également en entrée certains paramètres relatifs à la consommation mais ils ne sont pas utilisés dans le code de l'outil, il cible donc les performances uniquement.

Les sources de cet outil datant de 2012 sont disponibles sur internet [4]. Le principal défaut de ce simulateur est le manque de documentation qui rend son utilisation et la compréhension de son fonctionnement interne difficile. Aucune licence particulière n'est indiquée, mais l'article présentant SSDSim indique que le code source est disponible pour utilisation publique.

NandFlashSim Les quatre simulateurs présentés ci-dessus prennent en entrée une trace niveau bloc car ils simulent des systèmes à base de FTL, à un niveau de granularité *couche de gestion et flash*. *NandFlashSim* (Jung et coll., 2012) simule quant à lui seulement l'architecture (complexe) de stockage, correspondant à un ensemble de puces, éventuellement organisées en canaux. Il prend donc en entrée

2. Les sources de FlashSim version C sont disponibles ici : `http://csl.cse.psu.edu/?q=node/322` [accédé le 6 octobre 2014]

3. FlashSim version C++ est disponible ici : `http://csl.cse.psu.edu/?q=node/321` [accédé le 6 octobre 2014]

4. SSDSim est disponible ici : `http://cse.unl.edu/~tian/software/SSDsim_v2_1.rar` [accédé le 6 octobre 2014]

une série de commandes flash de type legacy ou avancées. NandFlashSim ne propose donc pas de modèle pour la couche de gestion. Il est distribué sous la forme d'une librairie avec des interfaces bien définies, destinée à être compilée avec un programme (non livré avec NandFlashSim) représentant la couche de gestion et faisant appel à la librairie.

Écrit en C++, cette librairie est un simulateur au cycle près (*cycle-accurate*). Il est possible de décrire en détail les latences en cycles d'horloge pour les différentes phases des opérations flash de base et avancées : envoi des commandes, adresses, envoi / réception des données, etc. Des statistiques détaillées sur les performances et la consommation du système simulé sont disponibles en sortie après simulation.

NandFlashSim date de 2012. Ses sources sont disponibles sur internet [5]. NandFlashSim est distribué sous licence LGPL.

NandSim *NandSim* (MTD Contributors, 2008) est un module intégré au noyau Linux qui permet de reproduire de manière logicielle la présence d'une puce de mémoire flash NAND embarquée. Il ne s'agit pas d'un simulateur à proprement parler mais d'un *émulateur*. NandSim a été créé en 2004 et est régulièrement mis à jour par les contributeurs du noyau Linux. NandSim est donc disponible de base avec Linux, et ses sources distribuées avec les sources de Linux [6]. NandSIm est donc sous licence GPL. Le but premier de NandSim est de permettre le prototypage et le déboguage de systèmes de type FFS sous Linux embarqué. NandSim crée un périphérique virtuel utilisable sous Linux, on peut ainsi formater la puce de mémoire flash émulée avec un FFS et l'utiliser pour stocker des données. NandSim stocke le contenu des pages flash virtuelles en RAM. Il permet d'émuler la présence et l'apparition de blocs usés au fil de l'utilisation de la puce virtuelle, et de tester la tolérance des FFS face à ces difficultées. L'objectif principal n'est donc pas l'étude des performances. Néanmoins, il est possible de simuler avec NandSim des délais relatifs aux opérations flash, permettant d'effectuer des mesures de performances au niveau de l'applicatif (par exemple des temps d'exécutions). NandSim prend en entrée des paramètres architecturaux (la taille de flash à simuler, la taille d'une page, etc.), des valeurs de latences, et des informations sur les blocs usés.

EagleTree *EagleTree* (Dayan et coll., 2013) est un simulateur de SSD permettant de simuler une architecture complexe de puces de mémoire flash, la couche de gestion de type FTL, mais également l'applicatif accédant au système de stockage, en particulier la présence de l'ordonnanceur des E/S du système d'exploitation et la présence de plusieurs programmes (*threads*) accédant au stockage. On peut donc le classifier au niveau de granularité du système d'exploitation. L'objectif de ce simulateur est de permettre une exploration de l'espace de conception des SSD et de l'impact sur les performances des interactions entre l'applicatif et la couche de gestion.

EagleTree date de 2011 et est toujours régulièrement mis à jour. L'outil est écrit en C++ et ses sources sont disponibles sur internet [7], sous licence GPL.

Autres simulateurs Le modèle de performance présenté dans (Su et coll., 2009) est implémenté dans un simulateur nommé Flash-DBSim. Ce simulateur permet de décrire un système Linux contenant une puce flash embarquée gérée via la couche FTL du pilote NAND générique MTD de Linux embarqué. Comme énnoncé précédemment cette couche est déclarée obsolète par les développeurs de MTD et son utilisation est déconseillée (MTD Contributors, 2009). Flash-DBSim date de 2008 et à été mis à jour pour la dernière fois en 2009. Les sources de cet outil sont disponibles sur internet [8] (aucune licence

5. NandFlashSim est disponible ici : `http://www.utdallas.edu/~jung/nfs/pmwiki.php` [accédé le 6 octobre 2014]

6. Les sources de NandSim sont accessibles par exemple ici : `http://lxr.free-electrons.com/source/drivers/mtd/nand/nandsim.c` [accédé le 6 octobre 2014]

7. EagleTree est disponible ici : `https://github.com/ClydeProjects/EagleTree/` [accédé le 6 octobre 2014]

8. Flash-DBSim est disponible à l'adresse suivante : `http://kdelab.ustc.edu.cn/flash-dbsim/index_en.html` [accédé le 6 octobre 2014]

particulière n'est indiquée).

Dans El Maghraoui et coll. (2010), les auteurs proposent un modèle analytique de performances haut niveau (présenté en section précédente) et indiquent l'avoir implémenté dans un simulateur. il s'agit d'un module pour le noyau Linux intégré dans le système d'exploitation. Un périphérique virtuel est créé à la manière de NandSim, représentant cette fois un SSD. Les accès des applications à ce périphérique virtuel sont tracés et le modèle est appliqué pour estimer les performances. Cet outil ne semble pas être disponible sur internet. Les auteurs de Yoo et Park (2011) présentent un simulateur pour calculer la consommation d'une architecture de stockage multi-puces, en particulier le coût dû aux opérations parallèles.

Conclusion concernant les simulateurs En conclusion de cet état de l'art sur les simulateurs de systèmes de stockage flash, on peut noter un certain nombre de points. Premièrement il n'existe pas de simulateur pour systèmes de type FFS. La plupart ciblent des périphériques géré via FTL, c'est à dire des périphériques de type bloc, pour la plupart des SSD. En suivant la classification définie en section précédente, une grande partie de ces simulateurs se situe donc au niveau de granularité de modélisation *couche de gestion + flash* (voir figure 2.8 page 68). Alors que la plupart des simulateurs peuvent être utilisés pour faire de l'exploration de l'espace de conception en variant plusieurs paramètres de l'architecture de stockage et de la couche de gestion, plusieurs simulateurs proposent également une infrastructure plus ou moins évoluée pour développer des prototypes de nouvelles couches de gestion (FTL).

On a vu également que les systèmes de stockage à base de mémoire flash réalisent des opérations en arrière plan, relatives en particulier au ramasse-miettes et à la répartition de l'usure. Cela est vrai pour les SSD comme on l'a vu dans l'état de l'art sur l'exploration des performances et de la consommation. On sait que cela est également vrai pour les FFS YAFFS2, JFFS2 et UBIFS. Il est intéressant de noter qu'aucun des simulateurs ne semble proposer de support pour les opérations en arrière plan.

Aucun des simulateurs présentés ici n'est applicable dans le cas de notre travail. En effet, les auteurs de Gupta et coll. (2009); Kim et coll. (2009a); Hu et coll. (2011); Dayan et coll. (2013); Su et coll. (2009) ciblent des systèmes de type FTL et n'intègrent pas la notion de FFS. *NandSim* est un module Linux et son environnement d'exécution (le noyau) n'est pas adapté à l'estimation de performances. Enfin, *NandFlashSim* est un simulateur de type *cycle-accurate*. Or pour notre travail on souhaite réaliser un simulateur de type *à évènements discrets*. Ces concepts seront détaillés au chapitre 5 décrivant le simulateur.

On peut extraire les avantages que l'on retrouve dans divers simulateurs comme les points suivants :

— La *Précision des paramètres d'entrées des modèles de performance* : Les latences des opérations flash de base peuvent être modélisées et implémentées (A) à large granularité comme des constantes simples (Gupta et coll., 2009), avec un risque de perte en précision des résultats ; et (B) de manière très précise au cycle d'horloge près (Jung et coll., 2012) avec un risque d'augmentation du temps de développement et d'exécution du fait de la forte précision. Un bon compromis semble la décomposition des opérations flash en sous opérations simples tels que les temps de transfert entre la matrice de transistor NAND et le page buffer interne à la puce, et les temps de transfert sur le bus d'E/S ;

— Le support des *architectures avancées multi plans / puces (LUN) / canaux* ;

— Le support des opérations avancées comme le *copy-back*, *cache mode*, et les opérations liées aux architectures avancées comme les opérations multi plans / LUN / canaux ;

— Le fait de proposer une personnalisation de l'état de départ de la mémoire flash et de la couche de gestion en début de simulation ;

— La possibilité pour l'utilisateur de développer relativement aisément de nouvelles couches de gestion dans le simulateur : cela est possible si le simulateur propose des interfaces de développement bien définies, et isole suffisamment l'endroit au sein du code où l'utilisateur doit

implémenter son algorithme de gestion ;
— La possibilité pour le simulateur de prendre en entrée une trace réelle (enregistrée sur un système réel), et de proposer un simulateur de trace synthétiques ;
— Le support des opérations en arrière plan ;
— Le C semble être un langage de référence dans le domaine des simulateurs pour système de stockage à base de flash, probablement pour des raisons de performances. Quelques simulateurs sont écrits en C++.

Enfin, on peut noter qu'aucun simulateur ne s'impose vraiment comme une référence dans le domaine des systèmes de stockage à base de mémoire flash, comme peut l'être DiskSim dans le domaine des disques dur.

Le tableau 2.3 page 75 présente un résumé des informations principales relatives à chacun des simulateurs étudiés dans ce chapitre.

Nom	Granularité	Cible la consommation ?	Complexité des modèles	Trace en entrée	Arch. complexe supportée ?	Comm. avancées supportées ?	Type	Langage	Ref.
NandSim	Puce	Non	Simple	NA	Non	Non	Émulateur	C	MTD Contributors (2008)
Extension Microsoft pour DiskSim	Gestion + flash (FTL)	Non	Moyenne	Bloc	Oui	Partiel	Evts. discrets	C	Microsoft Research (2009)
FlashSim version C	Gestion + flash (FTL)	Non	Simple	Bloc	Non	Non	Evts. discrets	C	Gupta et coll. (2009)
FlashSim version C++	Gestion + flash (FTL)	Non*	Simple	Bloc	Oui	Partiel	Evts. discrets	C++	Kim et coll. (2009b)
SSDSim	Gestion + flash (FTL)	Non*	Moyenne	Bloc	Oui	Oui	Evts. discrets	C	Hu et coll. (2011)
NandFlashSim	Puce	Oui	Élevée	Série de commandes flash	Oui	Oui	Cycle-accurate	C++	Jung et coll. (2012)
EagleTree	Système d'exploitation (Gestion FTL)	Non	Moyenne	Description haut niveau d'une charge applicative	Oui	Oui	Evts. discrets	C++	Dayan et coll. (2013)
El Maghraoui et coll. (2010)	Système d'exploitation	Non	Simple	Quantités de données à écrire / lire	NA	NA	Émulateur	C	El Maghraoui et coll. (2010)
FlashDBSim	Gestion + flash (FTL)	Non	Simple	Bloc	Non	Non	Evts. discrets	C++	Su et coll. (2009)
Yoo et Park (2011)	Gestion + flash (FTL)	Oui	Simple	Bloc	Oui	Oui	Evts. discrets	C/C++	Yoo et Park (2011)

Table 2.3 – Résumé des différents simulateurs de performances et consommation pour systèmes de stockage à base de mémoire flash étudiés dans cet état de l'art.
*Les articles concernant ces outils présentent des résultats concernant la consommation mais à partir du code des outils il n'est pas possible de déterminer comment ces résultats ont étés obtenus.

5 Conclusion et positionnement du travail de thèse

5.1 Conclusion du chapitre

Dans ce chapitre on a premièrement vu les différentes techniques de benchmarking pour systèmes de stockage. Ces techniques définissent la charge d'E/S appliquée à un système en fonctionnement. Les caractéristiques de cette charge impactent la consommation et les performances du système de stockage. Les macro-benchmarks représentent une charge provenant d'un contexte applicatif donné. Ce type de benchmark est utile pour comparer des systèmes de stockages entre eux, et explorer le comportement d'un système à haut niveau, dans un contexte représentatif de son environnement de fonctionnement standard. On constate qu'il n'existe pas de macro-benchmarks ciblant le stockage dans le domaine de l'embarqué. Pour ce qui est des micro-benchmarks, ils sont utilisés pour effectuer des analyses locales de l'impact d'un élément particulier du système de stockage sur les performances et la consommation. On différencie les benchmarks développés pour être utilisés par la communauté scientifique et industrielle, des benchmarks ad-hoc, développés dans le cadre et pour les besoins d'une étude particulière.

Des études donnant des résultats d'exploration des performances et de la consommation des systèmes de stockage à base de mémoire flash ont été présentées. On en retient principalement le fait qu'il n'existe pas d'étude détaillée sur les systèmes à base de FFS. On peut néanmoins noter que ces études mettent en lumière différents éléments impactant la consommation d'un système de stockage en général : ce sont les caractéristiques architecturales du système, la couche de gestion, la charge en entrée et l'état du système de stockage. On peut noter le rôle central de la couche de gestion.

Des modèles présentés dans de multiples études s'attachent à décrire l'impact sur les performances et la consommation de ces éléments. On peut retenir plusieurs granularités de modélisation : du niveau micro-architectural jusqu'au niveau système d'exploitation. Chaque niveau de granularité possède des avantages et les inconvénients. On voit également qu'un modèle peut être développé pour être utilisé par la communauté, mais également pour les besoins d'une étude : on retrouve une séparation similaire à la classification généraliste / ad-hoc vue au niveau des benchmarks. Il n'existe pas de modèle détaillé concernant les performances et la consommation des systèmes à base de FFS, même si certains modèles simples et haut niveau sont disponibles.

Les modèles sont implémentés dans des simulateurs pour être utilisés. On retrouve différents simulateurs généralistes, opérant à différents niveaux de granularité de modélisation. Ces simulateurs ciblent en majorité des systèmes à base de FTL, en particulier des systèmes de stockage contenant des SSD. Il n'existe pas de simulateur ciblant les FFS. Les simulateurs se définissent par de nombreuses caractéristiques, en particulier la granularité des modèles implémentés, le support de résultats concernant la consommation, le support des commandes flash avancées / des architectures de stockage avancés (multi puces / canaux). On différencie d'une part les simulateurs permettant uniquement de réaliser une exploration de l'espace de conception des systèmes de stockage à base de mémoire flash par l'obtention d'estimation sur les performances et la consommation. On peut noter d'autre part les simulateurs permettant en plus de développer de nouveaux systèmes de couches de gestion.

5.2 Positionnement et approche

Cet état de l'art montre un manque de travaux concernant la modélisation des performances et la consommation des systèmes de fichiers dédiés aux mémoires flash : seulement deux études (Goyal et Mahapatra, 2005; Choudhuri, 2003) ciblent ce sujet, et les auteurs ne s'intéressent qu'à la consommation énergétique. De plus, la granularité des modèles est très large et potentiellement imprécise. Dans ce travail de thèse on propose donc d'effectuer une étude en trois phases, correspondant aux trois chapitres suivants de ce document :

1. Premièrement, on réalise un travail d'exploration de l'impact sur les performances et la consom-

mation dû à un système de stockage à base de FFS dans un système informatique de type Linux embarqué. Ce travail est réalisé en s'appuyant sur des mesures de performances et de consommation sur une plate-forme matérielles réelle. Durant cette phase d'exploration, on utilise des micro-benchmarks ad-hoc appliqués sur un système à base de FFS pour en étudier les performances et la consommation. Plus particulièrement, l'objectif de cette phase est d'identifier les différents éléments du système de stockage qui ont un impact significatif sur les métriques concernées. Dans le cadre de ce travail on présente également une infrastructure logicielle de trace développée, permettant l'exploration des comportements des systèmes à base de FFS. Cette suite d'outils de trace cible l'intérieur de la couche de gestion du stockage flash sous Linux (FFS, VFS, pilote NAND), et a été développée pour les raisons suivantes :

(a) Il est nécessaire d'explorer le fonctionnement interne de la couche de gestion pour pouvoir modéliser finement les performances et la consommation relatives au stockage flash ;

(b) Les techniques d'exploration internes à la couche de gestion sont peu présentes dans les études listées dans ce chapitre. Seuls quelques articles (Liu et coll., 2010; Homma, 2009) expliquent avoir tracés des évènements au sein de la couche de gestion. Il s'agit pour Homma (2009) d'instrumentation de code, et les auteurs de Liu et coll. (2010) ne donnent pas d'information sur le processus de trace. Ces outils ne sont donc pas réutilisables dans un autre contexte.

Cette phase d'exploration est décrite au chapitre suivant, le chapitre 3 ;

2. Par la suite, on s'attache à décrire l'impact des différents éléments identifiés dans la phase précédente sur les performances et la consommation du système de stockage à base de FFS. Cela est fait sous forme de modèles de performances et de consommation pour les différents composants matériels constituant l'architecture du système de stockage, et sous la forme de modèles fonctionnels représentant les algorithmes de la couche de gestion (FFS). L'ensemble de ces modèles se situe au niveau de granularité du système d'exploitation. Les opérations sur lesquelles on se concentre sont les opérations de lecture et d'écriture dans les fichiers. Des modèles spécifiques ont étés développés dans le cadre de cette thèse pour les raisons suivantes :

   (a) Il n'existe aucun modèle décrivant de manière fine le comportement de la couche de gestion flash sous Linux :

   (b) Les seuls travaux en relation sont *choudhuri2003* et *goyal2005*. Il s'agit de modèles très haut niveau, ne prenant pas en compte le fonctionnement interne de la couche de gestion, et dont la précision peut être discutée.

   La méthodologie de modélisation est décrite au chapitre 4 ;

3. Ces modèles sont implémentés dans un simulateur pour système de stockage à base de mémoire flash. Ce simulateur permet d'estimer les performances et la consommation d'un système de stockage de type FFS, mais également de type FTL (en particulier SSD). Il supporte les commandes et architectures avancées. Il est construit de manière à faciliter le prototypage de nouveaux systèmes de couches de gestion. Il est notamment possible de spécifier le comportement de divers mécanismes s'exécutant en arrière plan, comme le ramasse-miettes. Les différents modèles implémentés dans le simulateur sont validés à une granularité fine, en comparant les estimations avec des mesures réalisées lors de l'application de micro-benchmarks sur des plate-formes réelles. On a choisi de développer un simulateur dédié, et de ne pas ré-utiliser l'existant pour les raisons suivantes :

   (a) Il n'existe pas de simulateurs ciblant les FFS. Les outils présentés dans ce chapitre ciblent les FTL, et sont difficilement adaptables au contexte de ce travail de thèse ;

   (b) NandFlashSim (Jung et coll., 2012) fait exception, car il ne simule que la partie flash, et peut donc en théorie être intégré dans un outil simulant un FFS. Néanmoins, il s'agit d'un

simulateur *cycle-accurate*. Dans le cadre de ce travail on souhaite développer un simulateur à évènements discrets, comme c'est le cas d'une grande partie des simulateurs de stockage.

Le simulateur et la validation des modèles sont présentés au chapitre 5.

CHAPITRE 3

Exploration des métriques de performances et de la consommation des systèmes de stockage à base de FFS

Ce chapitre présente la première étape du processus de modélisation des performances et de la consommation adopté dans ce travail de thèse. On se concentre ici sur les systèmes de stockage à base de FFS sous Linux. L'objectif du travail présenté dans ce chapitre est de mettre en évidence les différents éléments du système de stockage à base de mémoire flash qui impactent de manière significative les performances et la consommation de ce système. Cela est réalisé en deux parties. Premièrement, il est nécessaire d'étudier de manière détaillée la façon dont une requête applicative (par exemple un appel système read()*) est manipulée par les différentes couches de gestion du stockage (VFS et le FFS). Il s'agit d'exploration fonctionnelle, réalisée en étudiant le code du système d'exploitation et des FFS (sous Linux le code des FFS est dans l'OS). On se concentre en particulier sur JFFS2. Deuxièmement, on réalise des mesures de performances et de consommation pour vérifier l'impact effectif de divers éléments des systèmes à base de FFS. Ces mesures ont été réalisées dans le cadre de l'étude des opérations de lecture et d'écriture de données dans des fichiers. La méthodologie de mesure, utilisée notamment pendant la phase d'exploration, est nécessaire pour plusieurs raisons :*

- *L'identification et la caractérisation des éléments et mécanismes à cibler dans la phase de modélisation ;*
- *L'extraction de paramètres : lors de son utilisation, le modèle prend, en général, un ensemble de paramètres en entrée représentant, entre autres, un profil de performance et de consommation pour le système de stockage représenté dans un environnement donné. Afin d'obtenir les valeurs de ces paramètres, des mesures sur des plates-formes réelles sont réalisées ;*
- *La validation des modèles en comparant leur estimations avec des résultats de mesures en environnement réel ;*

Dans ce chapitre, on présente en premier lieu les résultats de l'exploration fonctionnelle : on explique dans les détails la manière dont une requête applicative est traitée par le système de stockage. Dans la section suivante, on présente une suite d'outils d'exploration développée dans le cadre du travail présenté dans cette thèse. Ces outils permettent de réaliser des mesures concernant les performances et le fonctionnement des systèmes de stockage de type FFS sous Linux embarqué. Ils ont été utilisés lors des différentes phases du travail de modélisation présenté ici, pour répondre aux objectifs listés ci-dessus. Enfin, en troisième partie on présente les résultats des mesures de performance et de consommation permettant d'isoler les éléments du système de stockage impactant ces métriques.

Sommaire

Vue globale de la méthodologie d'exploration et des outils utilisés La phase d'exploration présentée dans ce chapitre peut être déclinée en plusieurs sous-parties, représentées sur la figure 3.1. On souhaite comprendre (A) le fonctionnement des algorithmes de gestion (exploration fonctionnelle de VFS et FFS), et (B) identifier les éléments d'un système de stockage à base de FFS qui impactent les performances et la consommation des accès au stockage sur puce flash embarquée sous Linux. On se place donc ici à la granularité du système d'exploitation. Plus particulièrement, on se concentre sur les appels systèmes Linux *read()* et *write()* qui correspondent respectivement aux opérations de lecture et d'écriture de données dans des fichiers. Dans ce chapitre, si l'on souhaite identifier les éléments impactant les métriques qui nous intéressent, on ne quantifie pas cet impact dans les détails. Cela sera réalisé en phase de modélisation, présentée dans le chapitre suivant.

L'exploration fonctionnelle est réalisée en deux parties : premièrement, une étude *hors-ligne* est effectuée. Il s'agit d'étudier la littérature décrivant le fonctionnement des couches de gestion du stockage flash : livres et articles concernant Linux (Bovet et Cesati, 2005) et les FFS (Woodhouse, 2001; Manning, 2010; Hunter, 2008). De plus, une étude du code source des FFS, du sous-ensemble du code du VFS de Linux qui nous intéresse, et du pilote NAND MTD est effectuée. Cela est fait dans le but de comprendre dans le détail le fonctionnement de l'implémentation de la couche de gestion. Dans ce chapitre on se concentre plus particulièrement sur la gestion via le FFS JFFS2 (Woodhouse, 2001). Les codes sources des FFS, du pilote NAND et de VFS sont intégrés dans les sources de Linux. Cela est vrai nativement pour JFFS2 et UBIFS, et via un patch pour YAFFS2. L'étude des sources du noyau Linux peut s'effectuer de multiples manières. Il est possible de télécharger le code de Linux depuis le site officiel[1] et de l'étudier via un simple éditeur de texte. On peut également utiliser des outils d'exploration du code comme par exemple *cscope* (Steffen et coll., 1985), ou des outils en ligne comme *Linux cross reference* (Gleditsch et Gjermshus, 2006).

L'exploration fonctionnelle est également réalisée à l'exécution (*en ligne*), sur des plates-formes matérielles présentées ci-dessous (section *Plates-formes matérielles utilisées*). Par *en ligne* on entend l'exploration du fonctionnement des mécanismes de gestion du stockage alors que le système est en cours d'exécution, et que des E/S sont actuellement traitées. Plusieurs outils ont été utilisés. Premièrement, on a utilisé des outils de traces, développés dans le cadre de cette thèse, et qui seront présentés dans la deuxième section de chapitre. Deuxièmement, on a utilisé *gdb-server* (Stallman et Pesch, 1991)

1. Le code de toutes les versions de Linux est disponible sur `https://www.kernel.org/`

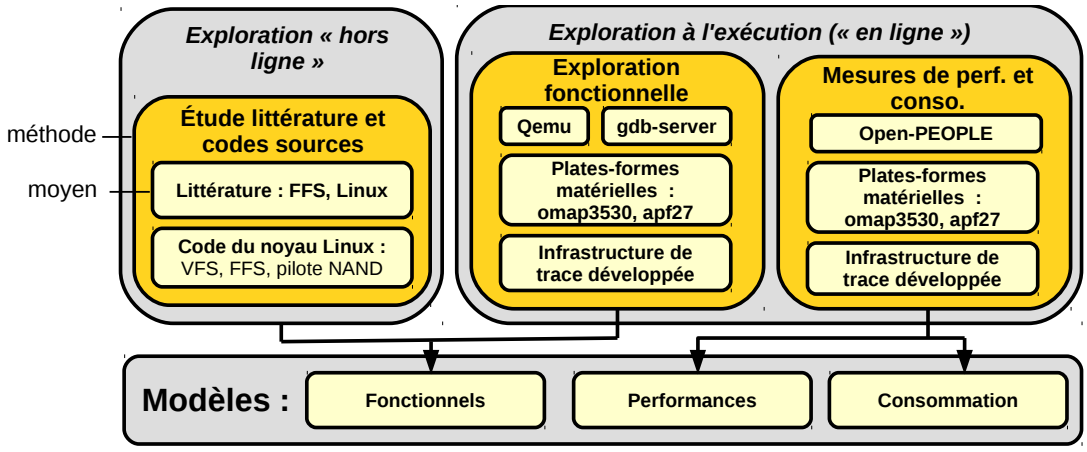


FIGURE 3.1 – Différentes méthodes d'exploration misent en œuvre, et les types de modèles construits avec les résultats obtenus via chacune des méthodes.

Figure 3.2 – Photographies des cartes *Mistral Omap3evm* (gauche) et *Armadeus APF27* (droite) utilisées dans le cadre de ce travail de thèse.

couplé à l'émulateur *qemu* (Bellard, 2005). Gdb-server est une implémentation du débogueur GNU GDB qui permet de déboguer à distance une application. Cet outil est très utilisé dans le domaine de l'embarqué, dans des contextes de développement croisé avec une machine hôte (PC standard), et une plate-forme d'exécution cible, le système embarqué. Gdb-server n'est pas seulement utile pour le déboguage, mais également pour l'exploration : il permet de placer des points d'arrêts dans le code du noyau, et d'observer à l'exécution les valeurs de différentes variables. Cela est particulièrement utile dans un environnement tel que VFS, qui comporte de nombreux pointeurs de fonction. En effet, la fonction pointée pouvant être déterminée à l'exécution, une exploration hors-ligne est parfois insuffisante pour obtenir cette valeur.

Pour ce qui est des mesures de performances, elles ont été réalisées via une infrastructure de trace dédiée développée dans le cadre du travail de thèse, et présentée en deuxième section de ce chapitre. Les mesures de consommation ont, quant à elles, été réalisées via la plate forme de mesure de consommation OPEN-PEOPLE (Senn et coll., 2012; Benmoussa et coll., 2014) présentée dans la section concernant l'exploration de la consommation.

Pour ce qui est des versions de Linux utilisées dans le cadre de ce travail de thèse, il s'agit des versions compatibles avec les plates-formes matérielles utilisées (présentées ci-dessous) : Linux 2.6.37 et 2.6.38.

Plates-formes matérielles utilisées Deux cartes embarquées ont été considérées en phase d'exploration (et plus généralement dans le cadre du travail de thèse global) : la carte *Mistral Omap3evm* et la carte *Armadeus APF27*. Une photographie de chacune des plate-formes est présentée sur la figure 3.2.

La carte *Omap3evm* (Mistral Solutions, 2013) comporte un processeur *Texas Instruments OMAP3530*, basé sur un cœur *ARM Cortex A8*, cadencé à 720 Mhz. La carte embarque également une puce mémoire *Micron* contenant 256 Mo de mémoire DRAM, et 256 Mo de mémoire flash NAND SLC (Micron Inc., 2009). On exécute sur la carte la version de Linux 2.6.37 (datant de 2011). Dans la suite de ce document on nomme cette carte "carte Omap". La puce flash intégrée à cette carte contient des blocs de 64 pages de 2 Ko chacune. La plupart des mesures de consommation présentées ici sont réalisées sur la carte Omap, qui présente la particularité suivante : les mémoires flash et RAM sont situées dans la même puce. Il existe un point de mesure de consommation situé sur le rail d'alimentation de cette puce, les mesures réalisées sur ce point comprendront donc la consommation de la mémoire flash et de la mémoire RAM. Un autre point de mesure utilisé est situé sur le rail d'alimentation du CPU.

La carte *Armadeus APF27* contient un processeur *Freescale i.MX27*, basé sur un cœur *ARM926EJ-S*, cadencé à 400 Mhz. La carte embarque également 128 Mo de mémoire RAM et 256 Mo de mémoire flash NAND SLC *Micron* (Micron Inc., 2007). La carte supporte de manière officielle les versions de Linux 2.6.29 (2009) et 2.6.38 (2011). La puce flash intégrée à cette carte contient, tout comme celle de la carte Omap, des blocs de 64 pages de 2048 octets chacune.

Il faut noter que la plate-forme matérielle principale utilisée dans ce travail de thèse est la carte

Omap. La carte Armadeus est utilisée dans le cadre de travaux présentés en annexe.

1 Gestion du stockage à base de flash NAND sous Linux embarqué

Dans cette section on présente les résultats de l'exploration fonctionnelle. On décrit ici la manière dont les appels système *read()* et *write()* sont traités, partant de l'application d'origine de la requête, et traversant toutes les couches logicielles composant la couche de gestion (VFS, FFS, pilote NAND) pour au final déboucher (éventuellement) sur des accès flash. A noter que tous les appels à *read()* et *write()* ne se terminent pas forcément sur un accès à la flash. On rappelle également qu'on se concentre sur les opérations relatives aux tranferts de données, et que dans le cadre de ce travail on ne s'intéresse pas aux opérations concernant les méta-données (par exemple la résolution du chemin d'accès à un fichier), ou encore des opérations comme l'utilisation de verrous, la vérification de droits d'accès, etc. Au final, la description des différents algorithmes présentés dans cette section est très simplifiée par rapport au code du système d'exploitation étudié.

1.1 Écriture de données dans un fichier : appel système *write()*

Sous Linux, on peut depuis un programme écrire des données dans un fichier ouvert en écriture via l'appel système *write()*. A noter que les appels à *write()* peuvent être encapsulés dans des librairies, en fonction du langage de programmation utilisé pour construire le programme. *write()* prend en paramètre un descripteur représentant le fichier cible, un buffer contenant les données à écrire, et un entier représentant le nombre d'octets à écrire. De manière implicite est également indiquée l'adresse logique dans le fichier à partir de laquelle l'écriture s'effectue, on l'appelle l'*offset*. La première écriture sur un fichier ouvert de manière standard est réalisée à l'offset 0, et chaque écriture / lecture incrémente l'offset d'une valeur égale au nombre d'octets écrits ou lus. Ouvrir un fichier avec le flag `O_APPEND` permet de positionner directement l'offset à la fin du fichier. L'appel système *lseek()* permet de manipuler l'offset directement.

a) *write()* : couche VFS

La figure 3.3 représente le traitement d'un appel système *write()* par la couche VFS sous Linux. Un appel à *write()* (A sur la figure 3.3) déclenche l'appel d'une série de fonctions dans le noyau qui débouche sur un appel à *vfs_write()* (B). Il s'agit là du point d'entrée dans VFS pour la fonction d'écriture de données dans un fichier. *vfs_write()* appelle elle-même une série de fonctions qui débouche sur un appel à *generic_perform_write()* (C). Il s'agit là du cœur du traitement de l'écriture. On rappelle qu'un fichier est vu par Linux comme un ensemble de page contiguës de taille 4096 octets (voir figure 1.18 au chapitre 1, page 34). *generic_perform_write()* itère sur chacune des pages Linux touchées par la requête d'écriture. Les pages touchées sont déterminées en fonction de la taille des données à écrire et de l'adresse cible dans le fichier, un exemple est illustré sur la figure 3.4. Pour chaque page Linux touchée, deux fonctions sont successivement appelées. Il s'agit des fonctions *write_begin()* et *write_end()*, qui sont implémentées au niveau du système de fichiers. Plus précisémment, ces deux fonctions sont au niveau de VFS des pointeurs de fonction. Ils pointent vers l'implémentation des fonctions *write_begin()* et *write_end()* du système de fichiers concret contenant le fichier accédé. Cela permet à VFS de jouer son rôle de couche d'abstraction pour les différents systèmes de fichiers supportés par Linux. Une fois que l'itération effectuée dans *generic_perform_write()* est terminée, l'opération d'écriture retourne (E).

Pour les systèmes de fichiers ne supportant pas la fonctionnalité de *write-back* relative au page cache (par exemple JFFS2 ou YAFFS2), l'écriture effective sur le média de stockage est réalisée pendant l'exécution de la fonction *write_end()*. Le processus à l'origine de l'écriture bloque sur l'appel à *write()* qui a déclenché l'appel à *write_end()* : les écritures sont synchrones (même si ces systèmes de fichiers ne supportent pas l'utilisation des flags d'ouverture de fichiers `O_SYNC` ou `O_DIRECT`). Pour un système

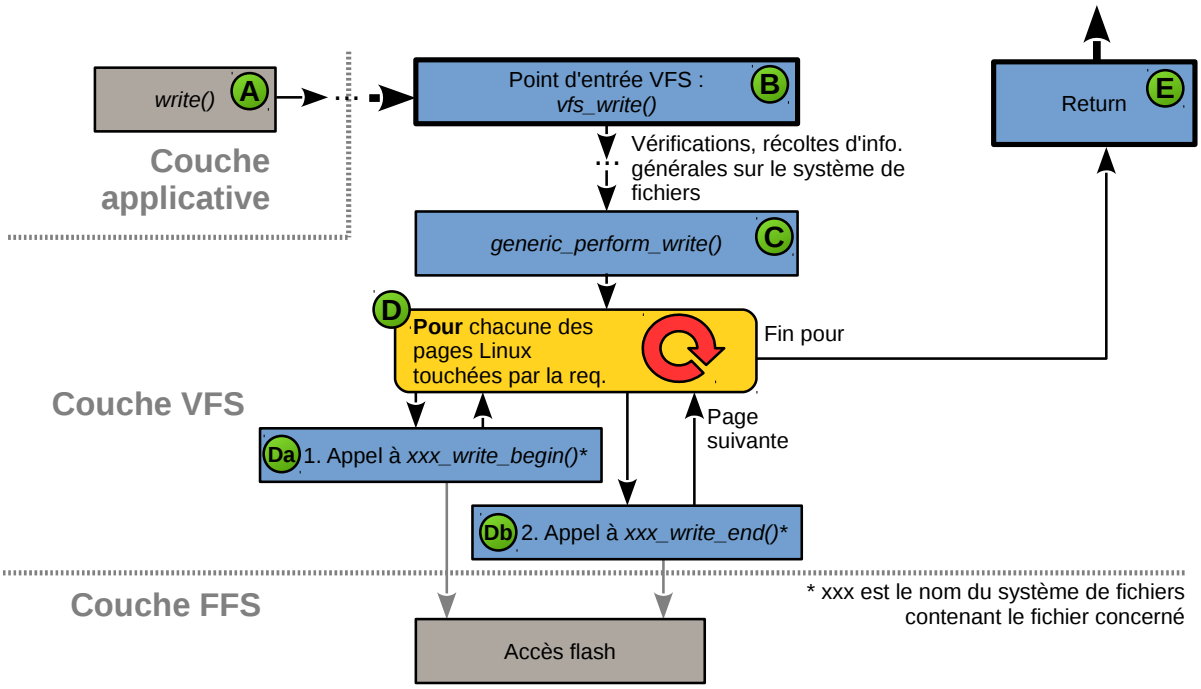


Figure 3.3 – Séquence simplifiée d'opérations représentant le traitement d'un appel système *write()* par la couche VFS

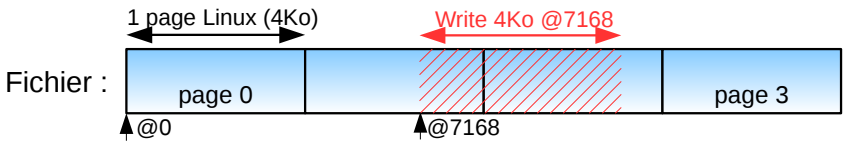


Figure 3.4 – Exemple d'écriture de taille 4 Ko dans un fichier de 12 Ko, à l'adresse 7168. On voit que les pages Linux touchées sont les deuxième et troisième pages du fichier.

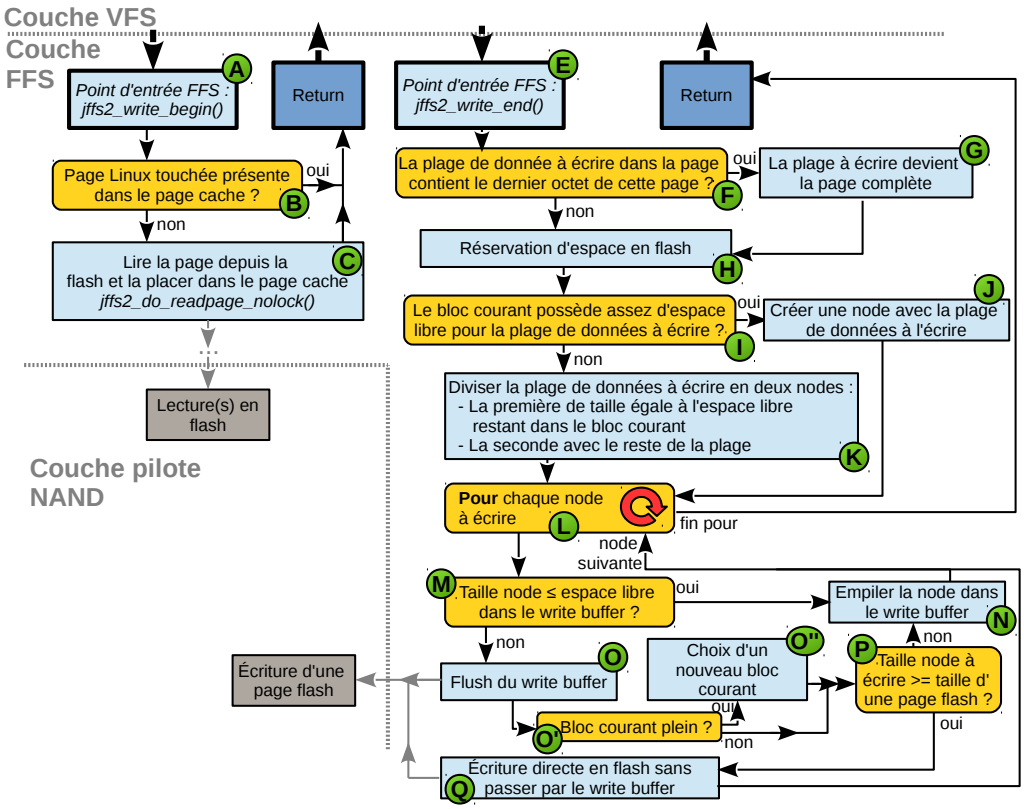


FIGURE 3.5 – Séquence simplifiée d'opérations représentant le traitement d'une opération d'écriture par JFFS2, c'est à dire les fonctions *jffs2_write_begin()* et *jffs2_write_end()*.

de fichiers comme UBIFS qui supporte le mécanisme de write-back, l'écriture en flash n'est pas réalisée durant *write_end()*, les données sont uniquement mises à jour dans le page cache : il s'agit de données dites *dirty*, qui ne sont pas synchronisées avec le média de stockage. L'écriture effective est laissée au soin d'un thread noyau, le démon *pdflush*. Ce thread se réveille à de multiples occasions, notamment lorsque la quantité de données dirty dépasse un certain seuil.

b) *write()* : couche FFS

On présente ici le traitement par le FFS JFFS2 des écritures, c'est à dire les grandes lignes des implémentations des fonctions *write_begin()* et *write_end()* introduites ci-dessus. Elles sont, dans le cadre des opérations d'écriture, les interfaces entre le système de fichiers virtuel et le système de fichiers concret. Dans notre cas il s'agit des fonctions *jffs2_write_begin()* et *jffs2_write_end()*. Cela est illustré sur la figure 3.5.

jffs2_write_begin() Le but général d'une fonction de type *write_begin()* est de préparer l'écriture à venir. Le système de fichiers y effectue, en général, de l'allocation mémoire et diverses opérations très dépendantes du type de système de fichiers. De plus, la page Linux cible de l'écriture doit être chargée dans le page cache par *write_begin()* (si elle ne s'y trouve pas déjà). C'est ce que fait *jffs2_write_begin()* (A sur la figure 3.5) : si la page Linux touchée n'est pas présente dans le page cache (B), elle est lue depuis le média flash (C). Les opérations de lecture seront vues dans les sous-section suivante.

jffs2_write_end() Le but général d'une fonction de type *write_end()* est d'effectuer l'écriture dans la page Linux concernée (dans le page cache, en RAM), et de mettre à jour les méta-données du système

de fichiers pour prendre en compte l'écriture. Pour un système de fichiers classique (pour disque dur) comme par exemple *ext4*, l'écriture effective sur le disque dur n'est pas effectuée via *write_begin()* car elle est trop coûteuse en temps : la mise à jour est effectuée uniquement en RAM dans le page cache. L'écriture effective sera réalisée de manière asynchrone lors d'opérations de *flush* du page cache : il s'agit du mécanisme de *write-back*, qui permet de profiter du principe de localité temporelle pour absorber les mises à jour fréquentes.

JFFS2 quant à lui ne fait pas usage de ce mécanisme : l'écriture effective en flash est réalisée pendant l'exécution de *jffs2_write_end()* (mis à part un tamponnage dans un buffer que nous allons présenter dans les lignes suivantes). *jffs2_write_end* (E sur la figure 3.5) est appelée prenant en paramètre la page Linux touchée par l'écriture, et une plage de données à écrire dans cette page. Cette plage concerne soit la page complète, soit un sous-ensemble de la page. La fonction commence par vérifier si dans la plage d'adresses logiques concernées par l'écriture est contenu le dernier octet de la page Linux touchée (F). Si c'est le cas, JFFS2 décide de ré-écrire la page complète (G). Cette décision est effectuée pour limiter la fragmentation des données logiques sur la flash et sera présentée dans la section concernant la lecture. Ensuite, une réservation d'espace flash est effectuée pour préparer l'écriture à venir (H).

Comme dit précédemment, JFFS2 empaquette les écritures de données dans des paquets nommés *nodes*. En revenant sur l'algorithme de *jffs2_write_end()*, JFFS2 vérifie si la taille de la plage de données à écrire est inférieure ou égale à la taille de l'espace libre restant dans le bloc courant [2] (I). Si c'est le cas, JFFS2 crée une node contenant l'intégralité de la plage de données à écrire (J). Sinon, JFFS2 divise en deux nodes la plage de données à écrire (K) :

— La première node de taille égale à l'espace libre restant dans le bloc courant ;
— La seconde node contenant le reste de la plage de données à écrire.

Par la suite intervient le *write buffer* de JFFS2. Le write buffer est un buffer en RAM qui a la taille d'une page flash sous-jacente. Il correspond toujours à une page flash physique donnée. Comme l'unité d'écriture en flash est une page complète (généralement de 512 à 8192 octets), son but est de tamponner les écritures de taille inférieure à la taille d'une page flash pour éviter le gaspillage d'espace flash. Son fonctionnement est simple : les écritures flash sont empilées dans le write buffer qui est inscrit (éviction) en flash lorsqu'il est plein. Une écriture de taille supérieure à la taille d'une page flash est réalisée directement en flash sans passer par ce tampon.

JFFS2 itère sur chacune des nodes à écrire (une ou deux, point L) : Si la taille de la node est inférieure ou égale à la taille de l'espace libre dans le write buffer (M), la node est empilée dans le tampon et JFFS2 passe à la node suivante (N). Si ce n'est pas le cas :

1. Le write buffer est vidé en flash (O). Cela fait, si le bloc courant est plein (O') un nouveau bloc courant est choisit dans la liste des blocs prêts à être écrits (O") ;
2. Si la taille de la node à écrire est supérieure à la taille d'une page flash (P), la node est écrite directement en flash sans passer par le write buffer (Q). Sinon, la node est empilée dans le write buffer et JFFS2 passe à la node suivante (N).

Lorsque toutes les nodes ont été écrites *jffs2_write_begin()* retourne.

Le ramasse-miettes de JFFS2 Au fil des mises à jour des données dans les fichiers de JFFS2, de nouvelles nodes sont créées et en cas d'écrasement, les anciennes nodes sont marquées comme *invalides* (ou *obsolètes*). Le but du ramasse-miettes est de recycler les blocs contenant des nodes invalides. Le ramasse-miettes travaille sur des listes chainées d'éléments représentant les blocs flash sous-jacents, plus particulièrement les listes suivantes :

— La liste *free* contient des blocs libres prêts à être écrits ;
— La liste *clean* lie les blocs contenant uniquement des nodes valides ;

2. Bloc en cours d'écriture par JFFS2, voir chapitre 1 page 34

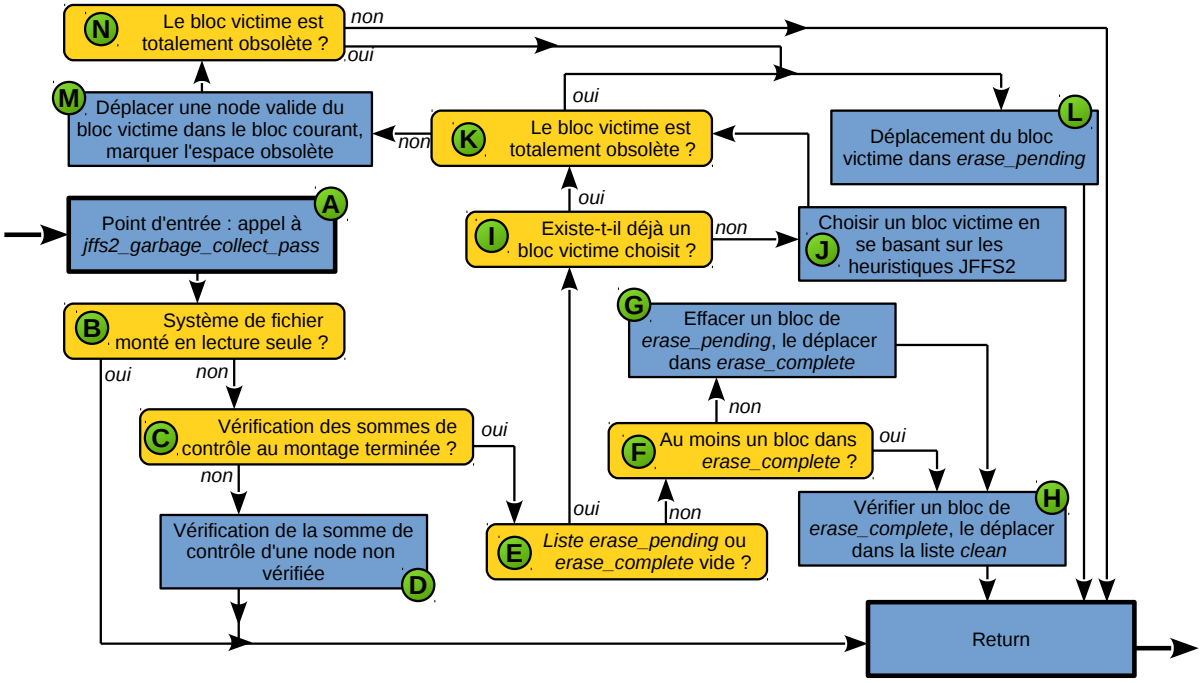


FIGURE 3.6 – Exécution d'une passe du ramasse-miettes de JFFS2

— La liste *eraseable* lie les blocs contenant uniquement des nodes invalides ;
— Les listes *dirty* et *very_dirty* contenant respectivement un nombre modéré (dirty) et important (very dirty) de nodes invalides ;
— La liste *erase_pending* contenant des blocs prêts à être effacés ;
— La liste *erase_complete* contenant des blocs effacés en attente d'être placés dans la liste *clean*.

Une passe du ramasse-miettes correspond à un appel à la fonction *jffs2_garbage_collect_pass()*. Une passe de ramasse-miettes n'aboutit pas forcément à l'effacement d'un bloc. Cette fonction est appelée à différentes occasions :

— Lorsque JFFS2 réserve de l'espace pour l'écriture de données (voir description de l'exécution de *jffs2_write_end()* plus haut) et que l'espace flash libre est critique ;
— Via un *thread* noyau tournant en arrière plan. Le thread se réveille de manière relativement régulière, et lance la fonction de ramasse-miettes si la quantité d'espace invalide est supérieure à un seuil donné.

L'exécution d'une passe du ramasse-miettes de JFFS2 est illustrée sur la figure 3.6. La fonction *jffs2_garbage_collect_pass()* (A sur la figure 3.6), point d'entrée du ramasse-miettes, commence par vérifier si le système de fichiers (partition) concerné est monté en lecture seule (B). Si c'est le cas le ramasse-miettes ne doit pas s'exécuter et la fonction retourne directement. Ensuite, JFFS2 vérifie si l'intégralité des nodes du système de fichiers ont été vérifiées via leur somme de contrôle (C). La vérification des sommes de contrôle (*checksum* en anglais) est une opération initialisée au montage et exécutée en arrière plan via le thread du ramasse-miettes. Elle consiste à vérifier l'intégrité des données contenues dans toutes les nodes du système de fichiers monté. Si cette phase n'est pas terminée, l'intégrité d'une node est vérifiée (D) et la passe se termine.

Si la vérification de l'intégrité des données est complète, JFFS2 vérifie si les listes *erase_pending* et *erase_complete* sont vides (E). Si ce n'est pas le cas :

— S'il existe au moins un bloc dans *erase_complete* (F), ce bloc est placé dans la liste *clean* (H) après avoir été vérifié. La vérification des blocs nouvellement effacés est effectuée par JFFS2 pour détecter les blocs usés. Elle consiste à lire l'intégralité du bloc pour vérifier qu'il ne contient que des '1'. La passe de ramasse-miettes retourne ;

— Si *erase_complete* est vide, alors un bloc de *erase_pending* est effacé (G), placé dans *erase_complete*, vérifié (H), puis placé dans la liste *clean*. La passe de ramasse-miettes retourne.

Si les deux listes *erase_pending* et *erase_complete* sont vides, JFFS2 vérifie si un bloc victime est actuellement disponible (I). Si ce n'est pas le cas un bloc victime est choisi (J) selon les règles suivantes (certaines règles peu importantes sont volontairement omises ici pour simplifier l'explication) :

— Il y a 40% de chance que le bloc victime soit choisi dans la liste *eraseable* (blocs complètement invalides) ;
— Il y a 46% de chance que le bloc victime soit choisi dans la liste *very_dirty* ;
— Il y a 12% de chance de choisir le bloc victime dans la liste *dirty* ;
— Enfin, il y a 2% de chance que le choix soit effectué dans la liste *clean*.

Ces chiffres sont extraits du code de JFFS2. La victime peut provenir de la liste *clean* pour des raisons de répartition de l'usure. Cela évite la stagnation de données froides pendant de longues périodes de temps dans les mêmes blocs flash. Une fois le bloc victime en main, si ce dernier est complètement invalide (K) il est placé dans *erase_pending* et la passe de ramasse-miettes retourne (L). Si ce n'est pas le cas, JFFS2 itère sur les nodes contenues dans le bloc victime jusqu'à trouver une node valide, la déplace dans le bloc courant (bloc en cours d'écriture), puis invalide la version de la node dans le bloc victime (M). Si à ce moment là le bloc victime est complètement invalide (N), il est placé dans *erase_pending* (L). Par la suite la passe de ramasse-miettes retourne.

c) *write()* : couche pilote NAND

Dans le cadre de l'écriture, il n'y a rien de particulier à noter pour ce qui est du pilote. Il est utilisé par le FFS pour effectuer des écritures de pages flash sur la puce. La fonction d'écriture de page flash est appelée par JFFS2 à chaque éviction du write buffer. Elle peut également être appelée plusieurs fois lorsque la taille de la node à écrire est supérieure à la taille d'une page, et que JFFS2 passe outre le write-buffer.

1.2 Lecture de données dans un fichier : appel système *read()*

Depuis un programme, l'appel système Linux *read()* est utilisé pour lire des données dans un fichier. *read()* prend en paramètre un buffer où sera stocké le résultat de la lecture et une taille de données à lire. L'adresse logique dans le fichier à partir de laquelle effectuer les lectures est indiquée implicitement comme pour *write()*. Elle peut être manipulée via *lseek()*.

a) *read()* : couche VFS

De manière similaire à *write()*, *read()* déclenche une série d'appels de fonctions qui débouche sur un appel à *vfs_read()* (A sur la figure 3.7), point d'entrée au niveau VFS. Cette fonction détermine les pages Linux touchées par la requête de lecture et réalise pour chacune des pages la suite d'opérations suivantes (B) :

— On vérifie tout d'abord si la page est présente dans le page cache (C) :
  — Si c'est le cas le système vérifie si le flag *read-ahead* de la page Linux en question est activé (D). Il s'agit d'un booléen présent sur chacune des pages Linux placées dans le page cache, relatif au mécanisme de préchargement read-ahead. Le fonctionnement de read-ahead est présenté en détail dans les paragraphes suivants. Si le flag est activé un appel à read-ahead est effectué, en mode dit *asynchrone* (E). Par la suite la page Linux est lue depuis le page cache (F) et on passe à la page suivante.
  — Si la page Linux demandée n'est pas dans le page cache, un appel à read-ahead en mode dit *synchrone* est effectué (G). Le système vérifie par la suite si cet appel à read-ahead a effectivement chargé la page Linux demandée dans le page cache (H, cela peut ne pas être le cas par exemple quand read-ahead est désactivé). Si la page Linux est trouvée dans le

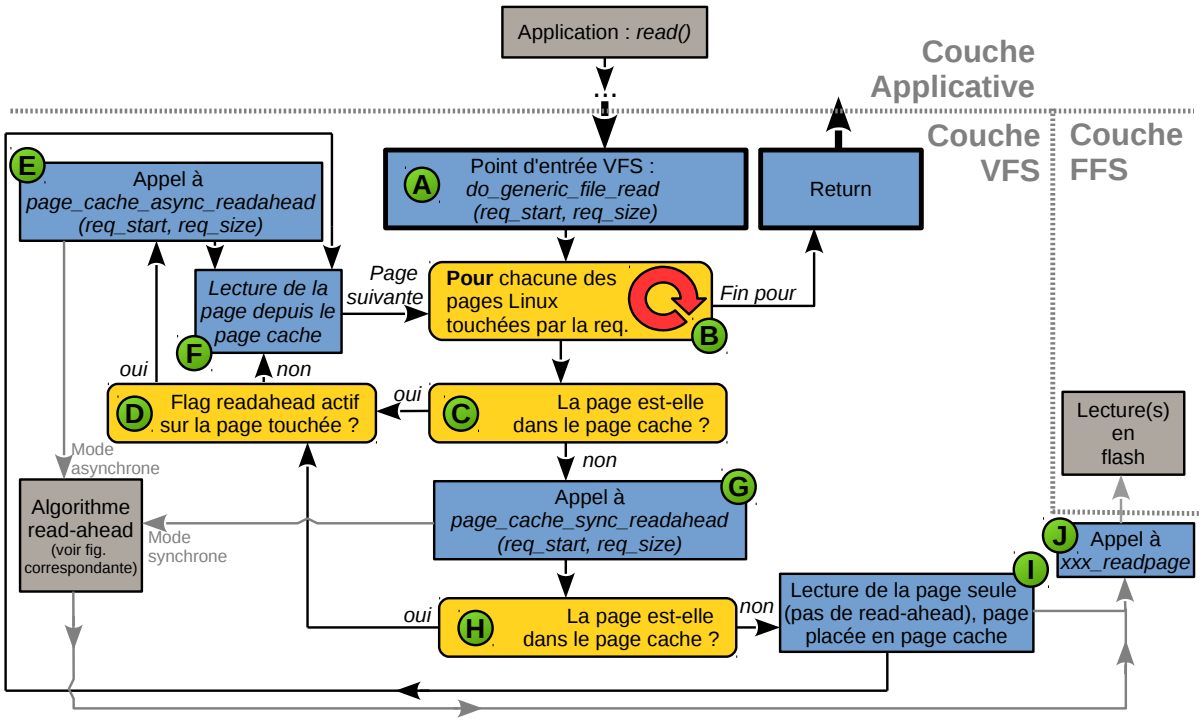


FIGURE 3.7 – Traitement par vfs d'un appel système *read()*

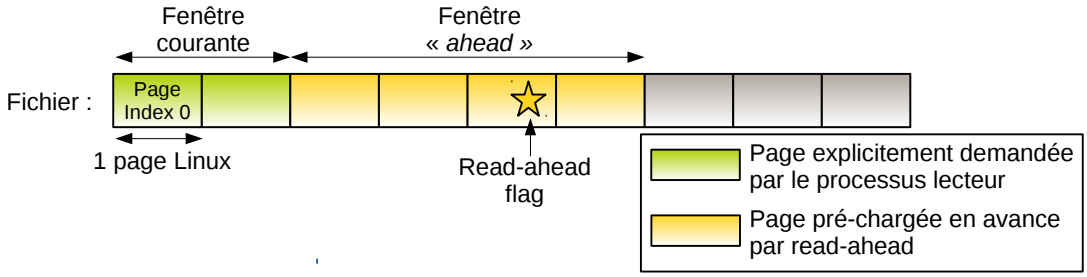


FIGURE 3.8 – Fonctionnement basique de l'algorithme read-ahead

page cache, le système exécute l'ensemble des étapes présentées dans le point ci-dessus et passe à la page suivante. Si la page n'est pas trouvée, alors elle est lue et placée directement dans le page cache (I) sans passer par read-ahead en utilisant l'interface de lecture entre VFS et le FFS : la fonction *readpage()* (J). Cette fonction est l'équivalent de *write_begin()* et *write_end()* dans le cas de la lecture. La page Linux est par la suite lue depuis le page cache et on passe à la page suivante.

Quand toutes les pages ont été lues l'opération de lecture prend fin.

Read-ahead On rappelle que read-ahead est un mécanisme de préchargement de données initialement développé pour des systèmes à base de disques durs. Il est néanmoins activé par défaut pour les FFS JFFS2 et YAFFS2. Le principe de base est le suivant : dans le cas ou le noyau détecte un accès séquentiel fait par un processus lisant des données dans un fichier, read-ahead décide de précharger (i.e. charger dans le page cache) les pages du fichier dont il estime qu'elles seront demandées par le processus dans un futur proche. D'importantes quantités de données sont préchargées en accès séquentiel. Si le motif d'accès est, au contraire, détecté comme aléatoire, read-ahead précharge des quantités de données moins importantes. Comme on l'a vu ci-dessus, un appel à read-ahead est réalisé par la fonction *do_generic_page_read()*, dans le cadre d'une requête de lecture touchant un certain

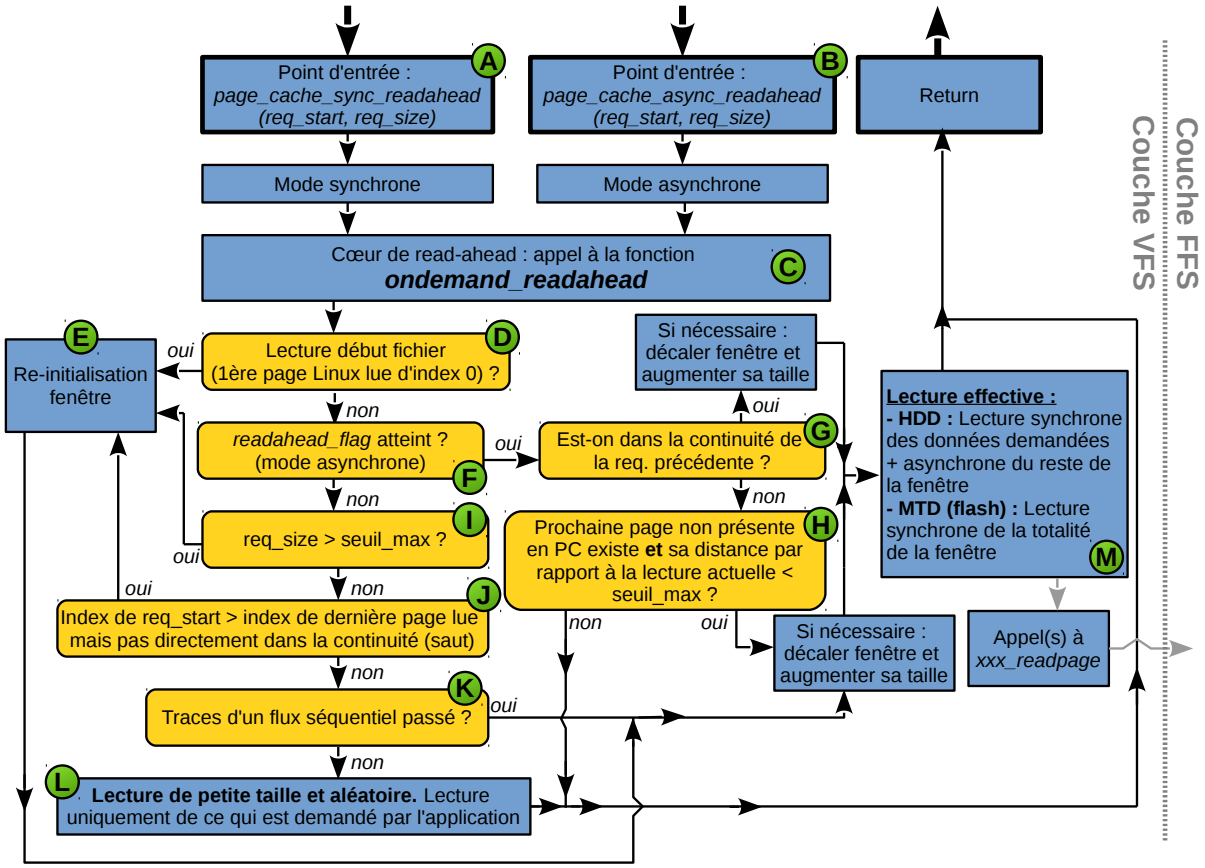


Figure 3.9 – Illustration du comportement fonctionnel de *read-ahead*

nombre de pages du fichier. Le fonctionnement basique de read-ahead est présenté sur la figure 3.8. A chaque appel, read-ahead détermine une fenêtre *courante* et une fenêtre dite *ahead*, qui sont des ensembles de pages Linux contiguës dans le fichier concerné, non présentes dans le page cache à l'appel de read-ahead. La fenêtre courante représente les données explicitement demandées par le processus lecteur. La fenêtre *ahead* représente les données qui vont être préchargées par read-ahead. Une passe de l'algorithme read-ahead peut être appelée de deux manières, correspondant à deux modes de fonctionnement :

— Le mode synchrone : lors d'un défaut de cache (*page cache miss*) dans *do_generic_page_read()* (G sur la figure 3.7) ;

— Le mode asynchrone : lors d'un succès de cache (*page cache hit*) sur une page possédant le marqueur (*flag*) *read-ahead* activé (E sur la figure 3.7). Comme la page Linux courante est déjà présente dans le page cache, une passe asynchrone de read-ahead ne définit qu'une fenêtre *ahead*.

Une passe de read-ahead se termine par la lecture en mode synchrone de la fenêtre courante, et en asynchrone pour la fenêtre *ahead*. En d'autres termes, quand la fonction correspondant à une passe de read-ahead retourne, les données explicitement demandées par le processus sont disponibles dans le page cache, et en arrière plan les données à précharger sont en cours de transfert depuis le media de stockage vers le page cache. A chaque passe read-ahead pose un marqueur (*flag read-ahead*) sur l'une des pages de la fenêtre ahead. Lorsque cette page sera demandée par le processus, read-ahead sera de nouveau appelé en mode asynchrone.

Une passe de read-ahead correspond à une série d'opérations illustrées sur la figure 3.9.

Les deux points d'entrée de read-ahead sont les fonctions *page_cache_sync_readahead()* (mode synchrone, A sur la figure 3.9) et *page_cache_async_readahead()* (mode asynchrone, B). Elles appellent

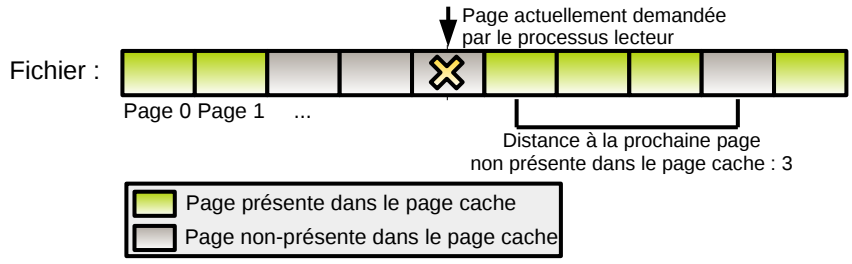


FIGURE 3.10 – Calcul de la distance à la prochaine page non-présente dans le page cache.

toutes deux la fonction qui est le cœur de l'algorithme read-ahead, *ondemand_readahead()* (C). Le but de cette fonction est de déterminer la quantité de données à précharger, c'est à dire la taille de la fenêtre *ahead*, et de lancer les transferts en faisant appel à la fonction *readpage()* du système de fichiers sous-jacent. La taille de la fenêtre est déterminée en fonction de la séquentialité des accès passés au fichier. Read-ahead commence par vérifier si la page Linux courante demandée par le processus est la page d'indice 0 (D), c'est à dire la première page du fichier. Si c'est le cas, le système suppose le début d'un accès séquentiel et la fenêtre ahead est (ré)initialisée (E). La taille de la fenêtre à l'initialisation est définie au niveau du système de fichiers. Une taille maximale est également définie, il s'agit de la quantité maximale de données préchargées lors d'une passe de read-ahead. Par la suite la lecture est lancée (M) et la passe read-ahead retourne.

Si la page accédée n'est pas la première page du fichier, read-ahead vérifie si le marqueur (*flag*) est atteint, autrement dit si l'on est en mode asynchrone (F) :

- Si c'est le cas, le système vérifie que les données actuellement demandées par le processus sont dans la continuité (séquentielles) des données précédemment demandées lors de la dernière passe de read-ahead (G) :
  - Si c'est le cas la taille de la fenêtre est augmentée (jusqu'à un seuil maximum), le transfert lancé (M), et read-ahead retourne. On est là sur un fonctionnement standard de read-ahead sous accès séquentiel ;
  - Si le marqueur est atteint sans que l'on soit dans la continuité de la requête précédente, alors le système vérifie la condition suivante : partant de la page Linux actuellement demandée par le processus, *quelle est la distance jusqu'à la prochaine page du fichier non présente dans le page cache* (H). Cela est illustré sur la figure 3.10. Si cette distance est inférieure à un certain seuil, read-ahead se comporte comme si l'accès était considéré comme séquentiel : la taille de la fenêtre est augmentée et cette dernière est décalée sur la première page du fichier non présente dans le page cache. Le transfert est lancé (M) et read-ahead retourne. Si la condition n'est pas vérifiée, read-ahead retourne directement sans rien lire.
- Si on est en mode synchrone (marqueur non-atteint), read-ahead vérifie deux conditions :
  - La taille de la requête *read()* actuelle est-elle supérieure à un certain seuil (I) ?
  - La requête actuelle est-elle dans la continuité de la précédente (J) ?
  - Est-il possible, à partir des caractéristiques de la requête actuelle et de l'historique de l'accès au fichier, de déterminer le fait qu'un flux séquentiel a précédemment eu lieu ?

  Si l'une de ces conditions est vérifiée alors l'accès est considéré comme séquentiel, read-ahead augmente la taille de la fenêtre et lance le transfert (M). Si aucune condition n'est vérifiée alors l'accès est considéré aléatoire et on lit uniquement les pages Linux demandées par le processus (L).

Dans la partie concernant l'exploration des performances, on montrera qu'il n'existe pas de mécanisme d'E/S asynchrone lors du traitement des accès à la mémoire flash. Cela a pour conséquence le fait que l'intégralité des fenêtres courante et *ahead* est lue avant que l'opération *read()* initiale ne retourne.

1.3 *read()* : couche FFS

On a vu que l'interface entre VFS et le FFS, dans le cas de la lecture, est la fonction *xxx_readpage()*, *jffs2_readpage()* dans le cas du FFS JFFS2. Cette fonction implémentée par le FFS charge une page Linux entière dans le page cache, et ce, même si le processus lecteur demande seulement un sous-ensemble d'une page.

Au sein de JFFS2, cette fonction de lecture de page effectue les opérations suivantes :

1. JFFS2 détermine l'ensemble de nodes constituant les données de la page Linux demandée ;
2. Les pages flash contenant ces nodes sont lues depuis la puce NAND par l'intermédiaire du pilote MTD ;
3. Les données composant la ou les page(s) Linux sont reconstruites et cette dernière est placée dans le page cache.

1.4 *read()* : couche pilote NAND

Le FFS utilise le pilote NAND MTD pour la lecture de pages flash. Comme énnoncé précédemment, MTD implémente un buffer en RAM utilisé pour les opérations de lecture. Ce tampon a la taille d'une page flash sous-jacente. A chaque fois qu'une page flash est lue, les données de cette page sont placées dans le buffer. Un futur accès en lecture à cette page sera alors servi depuis le buffer sans déclencher d'accès à la mémoire flash. Une page flash reste dans le tampon tant qu'elle n'est pas écrasée par une lecture d'une autre page, ou tant qu'elle n'est pas effacée / écrite en flash.

1.5 Exploration fonctionnelle - conclusion

Après avoir présenté les différents algorithmes et mécanismes fonctionnels qui entrent en considération dans le cadre de la lecture et de l'écriture de données dans les systèmes de stockages gérés par FFS, on peut identifier les éléments qui vont à priori impacter les performances et la consommation :

— Au niveau VFS, le *page cache* tamponne toutes les données écrites et lues. En cas d'accès en lecture sur des données présentes dans le page cache, les données sont lues depuis la RAM et il n'y a pas d'accès flash. La charge applicative en lecture est modifiée par l'algorithme *read-ahead*. En écriture, la charge est modifiée par l'utilisation potentielle du mécanisme de *write-back* ;

— Les *algorithmes de gestion des opérations d'écriture et de lecture* au niveau du FFS vont déterminer le nombre d'opérations de lecture et écriture de pages flash et effacements de blocs en fonction de la charge appliquée par la couche VFS sur la couche FFS. Des mécanismes de *tampon* comme le *write buffer* de JFFS2 absorbent les écritures de petites tailles ;

— Le pilote NAND MTD est utilisé par le FFS pour accéder à la flash. Un *buffer* de lecture présent au niveau pilote peut réduire le nombre de lectures de pages flash en cas d'accès répétés à la même page en lecture.

2 Une suite d'outils de trace ciblant l'exploration et la mesure de performances des FFS sous Linux

Dans le cadre de ce travail de thèse, plusieurs outils d'exploration ont été développés. Ces outils s'utilisent dans le cadre des systèmes de stockage à base de FFS sous Linux embarqué. Ils permettent de réaliser des mesures de performances et d'occurences d'évènements pour l'exploration, la modélisation, et également le benchmarking de ce type de systèmes de stockage. Ces outils ont été utilisés de manière importante pendant les phases d'exploration des performances, de modélisation, et de validation.

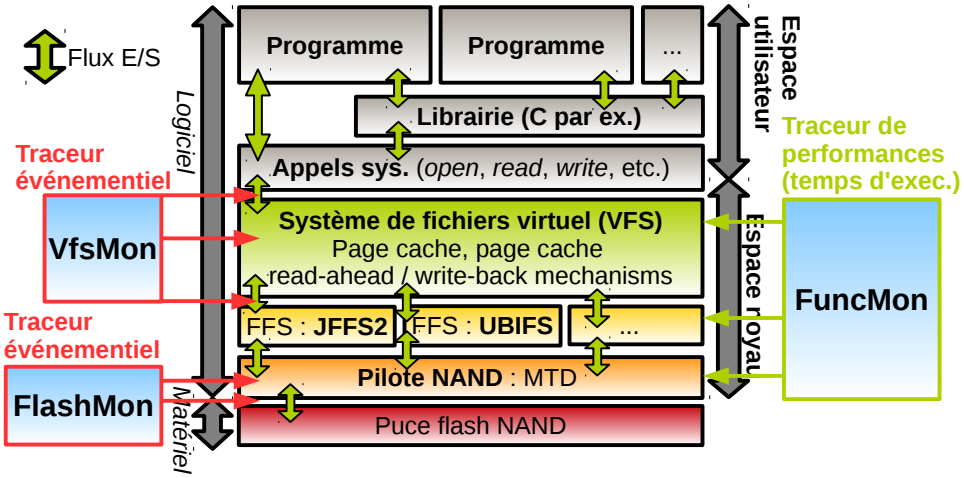


FIGURE 3.11 – Points de trace relatifs aux différents outils d'exploration développés

Ces outils permettent de récolter des informations sur les performances et le déroulement des différentes opérations relatives aux E/S en stockage flash de type FFS, plus particulièrement sur les opérations de lecture et d'écriture de données dans les fichiers. L'endroit où chaque outil opère dans le noyau est illustré sur la figure 3.11. Les outils sont au nombre trois :

1. *VFSMon* (*VFS Monitor*) trace des informations relatives aux occurrences de divers évènements liés à la gestion du stockage, et les paramètres associés, et ce, au niveau des couches de gestion VFS et FFS ;
2. *FlashMon* Boukhobza et coll. (2011); Olivier et coll. (2014a) (*Flash monitor*) trace des informations concernant les occurrences d'évènements au niveau du pilote NAND, c'est à dire un niveau très proche du matériel ;
3. *FuncMon* (*Function execution time Monitor*) trace les temps d'exécution de diverses opérations relatives à la gestion du stockage. Il s'agit d'un traceur de performances opérant à tous les niveaux de la pile logicielle de gestion du stockage flash.

On a vu au chapitre précédent pourquoi les techniques de trace présentées dans les différentes études concernant les FFS ne sont pas réutilisables. Les outils présentés ici utilisent un mécanisme de sondes fourni par Linux, se nommant *Kprobes* (*Kernel probes*, Keniston et coll. (2014)). Outre les Kprobes, il existe de multiples outils permettant de réaliser à la volée l'exploration d'évènements et de temps d'exécution au niveau du noyau. On peut par exemple citer *SystemTap* (Eigler et coll., 2005), *OProfile* (Levon, 2004), ou encore *ftrace* (Rostedt, 2008). SystemTap et OProfile ont le désavantage de nécessiter l'installation de dépendances (librairies, autres programmes, etc.), ce qui peut rendre leur installation assez fastidieuse, en particulier dans un environnement de développement de type embarqué (compilation croisée). Ftrace est, quant à lui, supporté nativement par le noyau. Néanmoins, cet outil est relativement mal supporté sur la branche ARM de Linux, pour les versions compatibles avec les plates-formes matérielles utilisées dans le cadre de notre travail. Les Kprobes sont un mécanisme intégré nativement au noyau, simple d'utilisation et présentant un impact léger sur les performances (Keniston et coll., 2014).

2.1 Flashmon

L'objectif principal de Flashmon est de tracer, au niveau du pilote NAND MTD, les accès à la flash via les opérations principales que sont la lecture / l'écriture de page flash, et l'effacement de blocs. La première version de Flashmon date de 2011. Une seconde version à été réalisée en 2013. Ces deux version ont étés présentées dans des articles (Boukhobza et coll., 2011; Olivier et coll., 2014a).

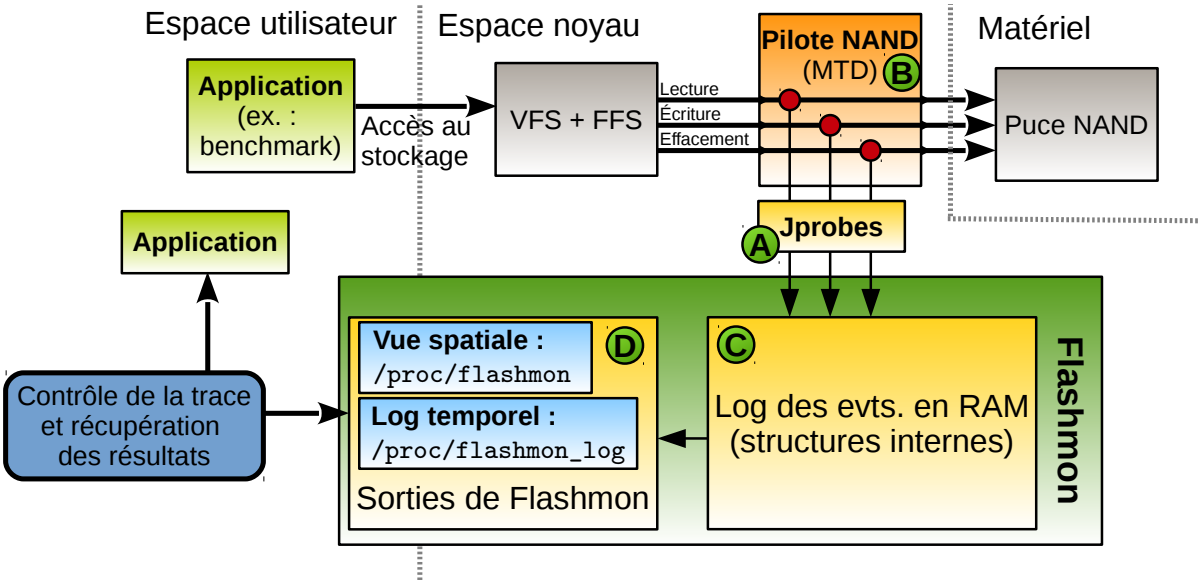


Figure 3.12 – Intégration et fonctionnement de Flashmon sous Linux

a) Flashmon : concepts principaux

Flashmon est un module pour le noyau Linux, traçant des appels de fonctions correspondant à l'exécution des opérations flash de base au niveau du pilote NAND générique MTD. Un module Linux est un programme contenant du code noyau, que l'on peut charger et décharger dynamiquement à l'exécution. Un module Linux est compilé à l'aide des sources du noyau, pour l'accès aux fichiers d'en-tête (*headers*) de ce dernier. Néanmoins, la construction d'un module ne nécessite pas la compilation du noyau lui-même. Flashmon est composé d'environ 1000 lignes de code C.

Flashmon utilise un sous-type de Kprobe appelé *Jprobe*.

Cœur de Flashmon Une Jprobe est une sonde posée sur une fonction noyau. Flashmon pose 3 sondes sur les fonctions MTD correspondant à la lecture, l'écriture, et l'effacement en flash (A et B sur la figure 3.12). A chaque sonde est associée une fonction, un *handler*, implémentée par Flashmon et exécutée à chaque fois que la fonction sondée est exécutée. Au niveau du handler, l'utilisation de Jprobes permet d'accéder aux valeurs des paramètres de la fonction sondée. Cela permet à Flashmon de tracer les évènements et paramètres suivantes : (1) quand une opération flash a lieu et (2) quelle adresse (indice de page(s) / de bloc) sont touché(es).

Les évènements tracés par Flashmon sont stockés dans un buffer en RAM (C sur la figure 3.12) pour minimiser l'intrusivité de l'outil : en effet inscrire à la volée un log des évènements tracés en flash interfèrerait avec les E/S tracées par Flashmon. Tracer au niveau MTD permet de rendre Flashmon (1) indépendant du modèle de puce flash utilisé, car MTD est un pilote générique ; et (2) indépendant du type de FFS utilisé. En effet les trois FFS principaux, JFFS2, YAFFS2 et UBIFS, font tous usage du pilote MTD pour les accès à la mémoire flash.

Flashmon trace les évènements suivants : lecture d'une page flash, écriture d'une page flash, effacement d'un bloc flash, et *cache hit* dans le buffer de lecture MTD. Pour chaque évènement les paramètres suivants sont tracés : (1) l'adresse (indice de page lue / de bloc effacé), (2) le temps d'arrivée auquel l'évènement a été tracé, (3) le nom du processus dans l'état *en cours d'exécution* sur le CPU au moment ou l'évènement a été tracé. Comme nous le verrons dans la section suivante concernant l'exploration des performances, MTD ne supporte pas les E/S asynchrones. Cela signifie que lors de la trace d'un évènement dans MTD, le processus exécuté à ce moment sur le CPU est le processus responsable de l'évènement. Le temps d'arrivée des opérations tracées est mesuré via la fonction

ktime_get() qui possède une granularité de l'ordre de la nanoseconde.

Flashmon est fourni avec un script permettant d'intégrer le module dans les sources d'un noyau existant, et créant une entrée pour Flashmon dans le menu de configuration de la construction du noyau. Il est alors possible de compiler Flashmon comme un module standard, mais également de l'intégrer comme une fonction à part entière du noyau (*built-in*). Ce faisant, Flashmon est chargé dès le démarrage du noyau, avant l'opération de montage du système de fichiers racine. Cela permet d'observer les accès flash au cours de cette opération de montage. Dans l'article présentant la version 2 de Flashmon, un cas d'étude concernant les accès flash au montage d'un système de fichiers racine est réalisé (Olivier et coll., 2014a).

b) Flashmon : fonctions sondées

Trouver au sein de MTD les fonctions à sonder n'est pas un problème dont la solution est triviale. MTD peut être vu comme une pile de couches logicielles. La partie supérieure de cette pile contient des fonctions génériques haut niveau pour les accès à la flash, appelées par les FFS. En bas de la pile on retrouve, au contraire, des fonctions très proches du matériel et dont l'implémentation varie en fonction du modèle de puce flash géré (voir la figure 1.17 page 33 au chapitre 1). Dans le cas de Flashmon, sonder des fonctions haut niveau est utile car elles sont très génériques. Néanmoins, pour plusieurs raisons cela peut mener à une erreur relativement importante pour ce qui est des paramètres tracés, en particulier le temps d'arrivée. Pour comprendre cela, il faut d'abord noter qu'un handler de Jprobe est exécuté juste avant l'appel à la fonction sondée. En d'autres termes la fonction sondée n'est exécutée qu'au moment ou le handler correspondant retourne. Flashmon détermine le temps d'arrivée d'un évènement au sein du handler associé via une fonction de type *getTime()*. Cela mène à une erreur potentiellement importante lorsque l'on trace à haut niveau dans MTD :

— Premièrement, les fonctions haut niveau de MTD correspondantes à des lectures et écritures peuvent déclencher des séries de plusieurs de ces opérations flash, en un seul appel de fonction. Même s'il est possible dans le handler de déterminer les adresses de toutes les pages touchées, le temps d'arrivée tracé à ce moment-là est assez éloigné du temps exact d'occurrence des opérations sur le matériel. Cela est vrai en particulier en ce qui concerne la dernière opération flash de la série ;
— Deuxièmement, le temps tracé au niveau du handler est éloigné du temps effectif de l'opération en matériel : la fonction tracée étant haut niveau, plusieurs appels de fonction au sein de MTD sont encore nécessaires avant l'exécution concrète de l'opération sur la puce.

Ainsi, les fonctions tracées doivent être le plus bas niveau possible, tout en restant relativement génériques pour ne pas correspondre à un seul modèle de puce. Lorsque Flashmon est lancé (on parle d'*insertion* pour un module Linux), une recherche est effectuée pour trouver la fonction optimale à sonder. Cette recherche s'effectue selon plusieurs paramètres, par exemple la version du noyau, le modèle de puce flash ou le système de fichiers utilisé. Ces paramètres sont vérifiés à la compilation et au lancement de Flashmon, et la fonction optimale à sonder est extraite d'une base de connaissances construite expérimentalement.

De plus à la compilation, Flashmon vérifie la version du noyau utilisée ce qui lui permet également de cibler les fonctions optimales en fonction de ce paramètre.

c) Flashmon : sorties et contrôle de la trace

Flashmon offre deux sorties par l'intermédiaires de fichiers virtuels, créés lors de l'insertion du module, dans le répertoire */proc* du système de fichiers racine (D sur la figure 3.12). On peut depuis l'espace utilisateur lire le contenu de ces fichiers pour récupérer les résultats de la trace. Le contenu de ces fichiers est construit dynamiquement par Flashmon à chaque demande de lecture, à partir

des résultats de trace stockés en RAM. Ces deux fichiers sont les suivants : `/proc/flashmon` et `/proc/flashmon_log`.

`/proc/flashmon` est une vue du nombre d'opérations flash reçues par chacun des blocs depuis le lancement du module. On l'appelle la vue *spatiale*. Dans ce fichier il existe une ligne par bloc de mémoire flash tracé. Chaque ligne contenant trois valeurs qui sont respectivement le nombre de lectures / écritures de pages, et le nombre d'effacements reçus par le bloc. Ce fichier est utile pour obtenir un aperçu instantané de la quantité d'opérations subies par la mémoire flash à un moment donné. Un exemple de trace obtenue via la vue spatiale est le suivant :

```
12 2 1
11 2 1
14 2 1
/* etc. */
```

Dans cette trace il y a une ligne par bloc de la mémoire flash tracée, ordonnés par adresse physique. Les première et deuxième colonnes représentent respectivement le nombre de lectures et d'écritures de pages reçues par le bloc. La troisième colonne représente le nombre d'effacements reçus par le bloc.

`/proc/flashmon_log` représente un log des évènements tracés classés par ordre temporel. Un exemple est le suivant :

```
125468.145741458 ; R ; 542 ; read_prog
125468.145814577 ; R ; 543 ; read_prog
125468.235451454 ; W ; 12 ; write_prog
125468.238185465 ; E ; 45 ; write_prog
/* etc. */
```

Chaque ligne représente un évènement tracé. La première colonne représente le temps d'arrivée de l'évènement en secondes. La seconde représente le type de l'évènement : R pour une lecture, W pour une écriture, E pour un effacement et C pour un *cache hit* dans le buffer de lecture MTD. La troisième colonne est l'adresse cible de l'évènement : l'indice des pages lues et écrites, et l'indice des blocs effacés. Enfin, la dernière colonne représente le nom du processus en cours d'éxécution sur le CPU lors de la trace de l'évènement. Il est également possible d'écrire des valeurs dans les deux fichiers virtuels pour contrôler la trace effectuée par flashmon. En écrivant des mots clefs dans les fichiers virtuels, l'utilisateur peut mettre en pause et relancer le processus de trace, mais aussi vider (*flush*) les résultats (réinitialiser le log et / ou la vue spatiale à 0).

Au lancement de Flashmon, l'utilisateur peut configurer la taille du buffer stockant la trace en RAM, ce qui permet de contrôler l'empreinte RAM du module. Il s'agit d'un buffer circulaire. On peut également spécifier au chargement de Flashmon si l'on souhaite tracer seulement une partition flash, ou l'espace correspondant à la puce toute entière.

d) Flashmon : conclusion

Flashmon s'avère très utile pour mesurer l'impact de charges de travail, concernant les nombres d'opérations flash réalisées, dans le cadre de la gestion du stockage sur flash embarquée via FFS. Flashmon a été testé et validé avec succès sur les deux plateformes matérielles utilisées dans le cadre de cette thèse. Flashmon a été testé sur une plage importante de versions de Linux, allant de Linux 2.6.29 à Linux 3.8. Une étude de l'impact de Flashmon sur le système tracé à été réalisée et montre que l'impact sur les performances est inférieur à 6%, et que l'empreinte RAM du module est contrôlable. Cette étude est présentée en annexe D page 243.

2.2 VFSMon et FuncMon

a) VFSMon

VFSMon est, tout comme Flashmon, un traceur évènementiel de type module noyau, ciblant le niveau VFS, et l'interface avec le FFS (voir figure 3.11 page 92). Cet outil, utilisant également des mécanismes de Jprobes, permet de collecter de nombreuses informations relatives au traitement des opérations de lecture et d'écriture au sein de la couche VFS, et au niveau de l'interface entre VFS et le FFS (fonctions *readpage()*, *write_begin()* et *write_end()*). Pour des raisons de généricité, nous avons choisi de ne pas récolter d'information de manière interne au FFS, pour se concentrer sur les évènements génériques se produisant indépendamment du type de FFS utilisé. On peut diviser les évènements tracés par VFSMon en 3 classes principales :

1. Les *entrées de VFS* : il s'agit de fonctions haut niveau, points d'entrées de VFS appelés via des appels systèmes. Les deux exemples principaux sont les fonctions *vfs_read()* et *vfs_write()*. VFSMon trace les appels à ces fonctions ainsi que les paramètres associés : identifiant du fichier accédé, taille et addresse (*offset*) des données accédées dans le fichier ;
2. Les *évènements internes à VFS* sont relatifs à l'utilisation du page cache et des mécanismes associés (read-ahead, write-back). VFSMon trace les succès et défauts de cache dans le page cache lors des opérations de lecture. Plusieurs paramètres relatifs à read-ahead sont tracés, permettant en particulier d'observer l'évolution de la taille de la fenêtre de préchargement. Il n'y a pas d'évènements tracés concernant directement la fonction de write-back, mais le comportement dû à ce mécanisme peut être observé de manière indirecte en traçant les appels asynchrones en écriture au FFS ;
3. Les *sorties de VFS* correspondent aux interfaces entre VFS et le FFS, c'est à dire les fonctions *readpage()*, *write_begin()* et *write_end()* présentées en section précédente.

Au chargement, on indique à VFSMon une partition à tracer. L'avantage principal de tracer une partition dédiée (autre que le système de fichiers racine) est de filtrer toutes les opérations dues au système lui-même qui viendraient perturber la trace si elle s'effectuait sur le système de fichiers racines (*rootfs*). Tout comme Flashmon, VFSMon stocke la trace dans un buffer circulaire en RAM dont la taille maximale est indiquée au chargement, et qui est entièrement alloué à ce moment là. L'avantage d'allouer l'intégralité du buffer au chargement du module est que (1) cela rend l'empreinte RAM déterministe et (2) il n'est pas nécessaire d'allouer de l'espace RAM à chaque évènement tracé. On rappelle que le temps d'exécution d'un handler de Jprobe impacte directement les performances du système tracé, il est donc important d'y effectuer un minimum de traitements.

VFSMon crée un fichier virtuel dans le répertoire */proc*. On peut lire ce fichier virtuel pour obtenir la trace, et y écrire pour mettre en pause et reprendre le processus de trace, ou encore remettre à zero la trace. Un exemple de trace obtenue via *VFSMon* est le suivant :

```
1 0.20700080;VFS_READ; 1 pages from page 0 - PC MISS
2 0.21297619; SYNC_RA; offset: 0, req_size: 1
3 0.21304926;  ONDEMAND_RA; offset: 0, req_size: 1
4 0.21311542;   RA_SUBMIT; start: 0, size:4, async_size:3
5 0.21315772;    RA_FLAGSET; offset: 0, n: 4, lookaheadc: 3
6 0.21349157;    FFS_READPAGE; page 0
7 0.22024388;    FFS_READPAGE; page 1
8 0.22620234;    FFS_READPAGE; page 2
9 0.23040542;    FFS_READPAGE; page 3
10 0.23535849;VFS_READ; 1 pages from page 1 - PC HIT
```

Dans cette trace il y a une ligne par évènement. Le premier champ correspond au temps d'arrivée, le deuxième au type d'évènement. Les champs suivants varient selon le type d'évènement. L'indentation d'un évènement correspond à son imbrication dans la pile d'appels de fonctions correspondant au code

de VFS. Dans notre exemple, un appel à *read()* déclenche un appel à *vfs_read()* (Ligne 1), touchant la première page d'un fichier. Il s'agit d'un page cache miss, read-ahead est donc appelé en mode synchrone (ligne 2). L'appel à read-ahead déclenche la lecture des 4 premières pages du fichier via des appels au FFS (L6 - L9). Finalement, un prochain appel à *read()* sur la deuxième page du fichier déclenche un page cache hit (L10) car les données ont précédemment été chargées en page cache.

b) FuncMon

FuncMon est un module de trace de temps d'exécution de fonctions noyau. Il n'est pas relatif à un niveau particulier de la hiérarchie de stockage et peut être utilisé pour mesurer le temps d'exécution de fonctions VFS, FFS et MTD. Son fonctionnement se base sur un sous-type de Kprobes nommé les *Kretprobes*. Ce sont des sondes qui, une fois posées sur une fonction noyau, permettent à l'utilisateur de spécifier deux handlers associés. Lors de l'exécution par le système d'un appel à la fonction sondée, le premier handler est exécuté avant l'appel à la fonction sondée, et le second handler est exécuté juste après le retour de la fonction sondée. Afin de déterminer le temps d'exécution de la fonction sondée, on peut via la fonction *ktime_get()* obtenir le temps système avant et après l'appel à la fonction sondée, avec une granularité de l'ordre de la nanoseconde. La différence entre ces deux valeurs représente alors le temps d'exécution de la fonction sondée.

L'utilisateur spécifie à la compilation de FuncMon le nom des fonctions noyau à sonder. Une fois de plus on peut également filtrer la trace en spécifiant une partition à sonder. Le contrôle et la récupération de la trace se fait une fois de plus via une entrée dans le répertoire */proc*. On présente ci-dessous un exemple de trace réalisée par FuncMon, configuré pour mesurer le temps d'exécution de la fonction *jffs2_readpage()* :

```
1 0.21343619;0.22009542;JFFS2_readpage;1751
2 0.22020772;0.22609003;JFFS2_readpage;1751
3 0.22617234;0.23030926;JFFS2_readpage;1751
4 0.23037695;0.23439465;JFFS2_readpage;1751
```

On peut y voir que quatre appels à *jffs2_readpage()* ont été tracés. La première colonne représente le temps d'appel à la fonction tracée, et la deuxième le moment où la fonction retourne. La troisième colonne est le nom de la fonction tracée, et la quatrième est le PID du processus courant au moment de la trace de l'appel.

c) Flashmon, VFSMon et FuncMon : interopérabilité

Les trois modules peuvent être utilisés de manière indépendante l'un de l'autre. A l'inverse les traceurs peuvent être lancés de manière concurrente dans un système. Dans ce cas, bien que les sorties soient des fichiers (virtuels) distincts, il est possible d'obtenir une unique trace contenant une fusion des trois traces. Il faut pour cela fusionner les traces et trier le fichier obtenu en fonction de la première colonne qui représente le temps d'arrivée des évènements. Cela est possible car : (1) la première colonne des trois traces contient toujours le temps d'arrivée de l'évènement, et (2) pour les 3 traceurs ce temps d'arrivée provient de la même source qui est le temps système, une horloge absolue du point de vue des traceurs. Un exemple d'une telle trace est présenté sur la figure 3.13.

La suite d'outil composée de Flashmon, VFSMon et FuncMon a été présentée dans un article accepté dans une conférence internationale (Olivier et coll., 2014c).

2.3 Exploration de la consommation : mesures avec la plate-forme *Open-PEOPLE*

Open-PEOPLE (*OPEN - Power and Energy Optimization PLatform and Estimator*, Senn et coll. (2012), Open People contributors (2012b, a); Yahia Benmoussa, Eric Senn et Jalil Boukhobza (2014)) est un projet ANR dont le but était de fournir une plate-forme permettant l'estimation et l'optimisation de


```
183202.020700080;VFS_READ; 1 pages starting from page 0 - PC MISS
183202.021297619; SYNC_RA; offset: 0, req_size: 1
183202.021304926;  ONDEMAND_RA; offset: 0, req_size: 1
183202.021311542;   RA_SUBMIT; start: 0, size:4, async_size:3
183202.021315772;    RA_FLAGSET; offset: 0, nr_to_read: 4, lookahead_size: 3
183202.021343619;183202.022009542;jffs2_readpage;1751
183202.021349157;    FFS_READPAGE; page 0
183202.021447157;R;28386;read
183202.021656849;R;28387;read
183202.021828772;R;28388;read
183202.022020772;183202.022609003;jffs2_readpage;1751
183202.022024388;    FFS_READPAGE; page 1
183202.022086388;R;28891;read
183202.022266772;R;28892;read
183202.022438311;R;28893;read
183202.022617234;183202.023030926;jffs2_readpage;1751
183202.022620234;    FFS_READPAGE; page 2
183202.022686388;R;28894;read
183202.022859311;R;28895;read
183202.023037695;183202.023439465;jffs2_readpage;1751
183202.023040542;    FFS_READPAGE; page 3
183202.023104542;R;28896;read
183202.023276619;R;28897;read
183202.023535849;VFS_READ; 1 pages starting from page 1 - PC HIT
```

Figure 3.13 – Trace résultant de la fusion des résultats des trois outils traçant un accès en lecture à un fichier.

la consommation des systèmes informatiques embarqués. L'une des contributions du projet est la *plate-forme matérielle Open-PEOPLE*. Il s'agit d'un banc contenant plusieurs cartes de développement sur lesquelles on peut effectuer des mesures de consommation. La plate-forme contient également un instrument de mesure de la consommation branché sur différents points de mesures au niveau des cartes. Il est possible d'accéder à distance (via internet) à cette plateforme pour exécuter des benchmarks sur l'une des cartes présentes, et obtenir en retour des valeurs de consommation.

La plate-forme contient, entre autres, une carte *Omap 3530*. C'est sur cette carte que sont réalisées l'intégralité des mesures de consommation présentées dans cette thèse. La carte d'acquisition utilisée par la plate-forme pour réaliser les mesures de consommation est une carte *National Instruments PXI-4472B* (National Instruments, 2014). Sur la carte Omap deux points de mesures sont considérés : le rail d'alimentation du CPU, et celui de la puce contenant la mémoire RAM et la mémoire flash. On ne s'intéresse pas au coût en consommation des tranferts sur les bus (*interconnect*). La plate-forme open-PEOPLE renvoie après l'exécution d'un benchmark des résultats sous la forme de fichiers CSV présentant les valeurs de puissances mesurées sur les différents points de mesure, en fonction du temps. Sauf mention contraire, la fréquence d'échantillonage de la carte d'acquisition est fixée à 1 kHz (1000 échantillons par seconde).

La granularité des mesures effectuées sur la plate-forme est relativement haute : la plage de temps pendant laquelle les mesures sont réalisées correspond à l'exécution du benchmark tout entier. Plus précisément, la plate-forme active la carte d'acquisition juste avant le début du benchmark, et la désactive au bout d'un temps renseigné par l'utilisateur : il est donc nécessaire de surestimer le temps d'exécution du benchmark. Du fait de ce niveau de granularité, il n'est pas possible d'obtenir facilement des valeurs de consommation pour des opérations très rapides comme la lecture de quelques Ko dans un fichier, ou encore un accès flash. Pour obtenir ce type de valeur, la technique utilisée est la suivante : on lance plusieurs fois au sein d'un benchmark l'opération dont on souhaite connaître la consommation (dans une boucle par exemple). On effectue alors une moyenne en fonction du nombre d'opérations réalisées.

La figure 3.14 représente deux exemples de résultats (fictifs) que l'on peut obtenir via Open-PEOPLE. La courbe de gauche représente la puissance sur un composant (CPU ou RAM), consommée en fonction du temps lors d'un benchmark dans lequel on effectue un nombre important de fois une même opération (par exemple un appel à *read()*) dans une boucle. On peut identifier sur cette courbe :

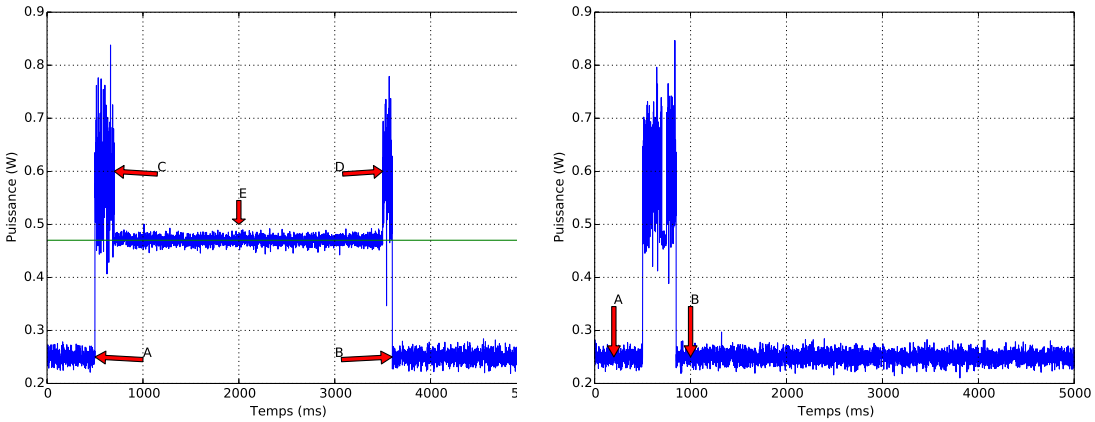


Figure 3.14 – Granularité de mesure de consommation sur Open-PEOPLE : il est nécessaire de répeter l'opération dont on souhaite mesurer l'impact un nombre relativement important de fois (courbe de gauche). En effet si l'opération est trop rapide et appelée en nombre insuffisant (courbe de droite), les résultats sont difficilement exploitables.

— La consommation à vide avant le lancement du benchmark (A), et après sa terminaison (B) ;
— Sur la figure de gauche, Le pic de consommation correspondant au chargement du programme qu'est le benchmark (C), et à sa fin (D) ;
— Sur la figure de gauche, Un plateau de consommation correspondant à la puissance consommée lors de l'opération qui nous intéresse. On peut calculer la puissance moyenne à partir de ces résultats, représentée ici par la ligne horizontale de couleur verte.

Si l'opération dont on souhaite mesurer l'impact sur la consommation est 1) trop rapide et 2) appelée en nombre insuffisant (courbe de droite sur la figure 3.14) :

1. Les temps de chargement / déchargement du programme deviennent prépondérants et il est difficile de distinguer et d'extraire la consommation de l'opération qui nous intéresse ;
2. Le nombre d'échantillons correspondant à la puissance de l'opération en question peut être trop faible pour obtenir une moyenne représentative.

2.4 Outils de trace : conclusion

Dans cette section nous avons présenté un ensemble d'outils de traces pour mesurer les performances du stockage à base de FFS sous Linux embarqué. Les outils opèrent à tous les niveaux de la hiérarchie de stockage. Ces niveaux sont le système de fichiers virtuel, le FFS, et le niveau pilote. Les outils présentés dans cette section ont été présentés dans plusieurs articles de recherche (Boukhobza et coll., 2011; Olivier et coll., 2014a). Ils sont utilisés dans plusieurs étapes de ce travail de thèse, en particulier l'étape d'exploration des performances.

3 Exploration des performances et de la consommation des systèmes à base de FFS : lecture et écriture de données dans des fichiers sous JFFS2

Dans cette section on identifie via des mesures de performances et de consommation les différents éléments du système de stockage qui impactent ces métriques. On se focalise sur les opérations de lecture et d'écriture de données dans des fichiers (appels système *read()* et *write()*), et sur le FFS JFFS2. Une étude multi-niveaux est effectuée sur la pile de gestion du stockage, et l'on adopte une approche de type *bottom-up* : on présente d'abord des résultats de mesures extraites au niveau pilote, proche

du matériel. Dans une seconde partie, on s'intéresse au niveau FFS, et dans une troisième partie au niveau VFS. Ces trois niveaux (pilote, FFS, VFS) correspondent respectivement aux points D, C et B sur la figure 1.16 présentée au chapitre 1 page 31. On rappelle que l'objectif ici est de montrer qu'il existe un impact sur la consommation et les performances pour différents éléments d'un système de stockage. Quantifier cet impact sera réalisé dans le chapitre 4.

3.1 Composants matériels et niveau de gestion pilote NAND MTD

Les composants matériels utilisés dans le système de stockage définissent ses performances et sa consommation. Ces composants sont les suivants :

1. Le type de *puce NAND* utilisé détermine les performances des opérations flash de base, et la consommation de la puce pendant l'exécution de ces opérations et au repos ;
2. La *mémoire vive* détermine la vitesse des lectures et écritures en RAM effectués par la couche de gestion, ainsi que la consommation de la RAM à vide / pendant ces transferts ;
3. Le type de *CPU* utilisé détermine la vitesse d'exécution des algorithmes de gestion, et encore une fois la consommation relative à ce composant en activité et au repos.

On rappelle que dans le cadre de ce travail de thèse, on adopte une approche orientée service : c'est une approche ciblant un service spécifique d'un OS. Ce service est le stockage secondaire embarqué à base de FFS. En effet, concernant les performances et de la consommation de la mémoire et du CPU, étudier ces valeurs pour ces composants à part entière est un travail important qui sort du cadre de l'étude présentée dans cette thèse. A noter que la consommation du cœur du processeur Omap3530 à été modélisé par (Kumar Rethinagiri, 2014) via une approche orientée composant. On a donc une vue simplifiée sur les performances et la consommation de la RAM et du CPU. Ces métriques sont ici uniquement explorées dans le cadre des opérations relatives à la gestion du stockage à base de FFS : il s'agit des différentes fonctions de la pile d'appel relative au traitement des E/S sous Linux. Pour ce qui est de la puce flash NAND, ces performances et sa consommation sont explorés dans les détails.

Dans le cadre de notre approche orientée service, le sous-service de plus bas niveau considéré pour la mise en œuvre de mesures de performances et de consommation est le pilote NAND MTD.

a) Performances et consommation des opérations flash de base

• **Objectif :** on souhaite observer l'évolution des performances des accès à la flash au niveau pilote, en fonction de l'opération appliquée (lecture, écriture, effacement), du nombre d'opérations réalisées dans un test, et du mode d'accès (séquentiel / aléatoire).
• **Méthode :** des micro-benchmarks niveau pilote ont été réalisés. Il s'agit de modules pour le noyau Linux, qui utilisent les structures MTD internes au noyau pour accéder directement à la mémoire flash, c'est à dire aux adresses physiques. Trois modules sont développés, un par type d'opération. Chaque module prend en paramètre un nombre d'opérations à réaliser. A l'insertion, le module lance les opérations dans une boucle dont il mesure le temps d'exécution via la primitive *ktime_get()*, avec une granularité de l'ordre de la nanoseconde. Un exemple de pseudo code pour le micro-benchmark en lecture est le suivant :

```
----------------------------------
input : nombre de pages à lire n
output : temps d'exécution total t
----------------------------------
start = get_time()
pour (i = 0 ; i<n ; i++) faire :
| lire page flash d'indice i
```

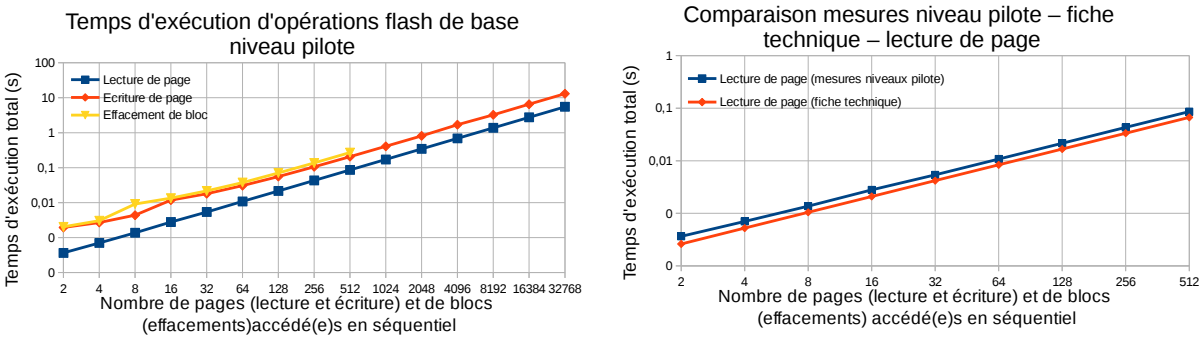


Figure 3.15 – A gauche : Temps d'exécution au niveau pilote de boucles réalisant un nombre donné d'opérations flash de base de même type. A droite : comparaison des temps d'exécution mesuré avec les chiffres provenant des fiches techniques.

```
stop = get_time()
t = stop - start
retourner t
```

On peut voir que les mesures de temps sont directement intégrées au benchmark. Les tests étant très simple, les outils de mesures développés présentés précédemment n'ont pas étés utilisés pour cette expérience. Les tests sont lancés sur la carte Omap (Linux 2.6.37), qui on le rappelle possède une puce flash contenant des pages de 2 Ko et des blocs de 64 pages. On fait varier le nombre de pages à lire / écrire par puissance de 2, de 2 à 32768. Le nombre de blocs à effacer varie de 2 à 512. Avant chaque test en écriture la mémoire flash est effacée. Les tests sont appliqués sur une partition de 100 Mo.

• **Résultats :** les résultats sont présentés sur la gauche de la figure 3.15. Ils montrent clairement que le temps d'exécution de chaque opération évolue de manière linéaire avec le nombre de pages / blocs accédés. A noter qu'ici les accès sont effectués en séquentiel. En accès aléatoire, une variation quasi négligeable peut être constatée : entre 5 et 10 μs par opération flash (Olivier et coll., 2013a). Si l'on compare ces chiffres à ceux fournis par la fiche technique de la puce flash (Micron Inc., 2009), on constate que les temps de réponse sont légèrement supérieurs pour les mesures niveau pilote. Sur la droite de la figure 3.15, on illustre cette différence pour l'opération de lecture. Des constats similaires sont fait pour l'écriture et les effacements. Si l'on part du principe que les informations données par la fiche technique sont correctes, alors la différence de temps représente un sur-coût (*overhead*) dû au driver. Cet overhead correspond au temps CPU et aux transferts RAM dus à la gestion du pilote MTD.

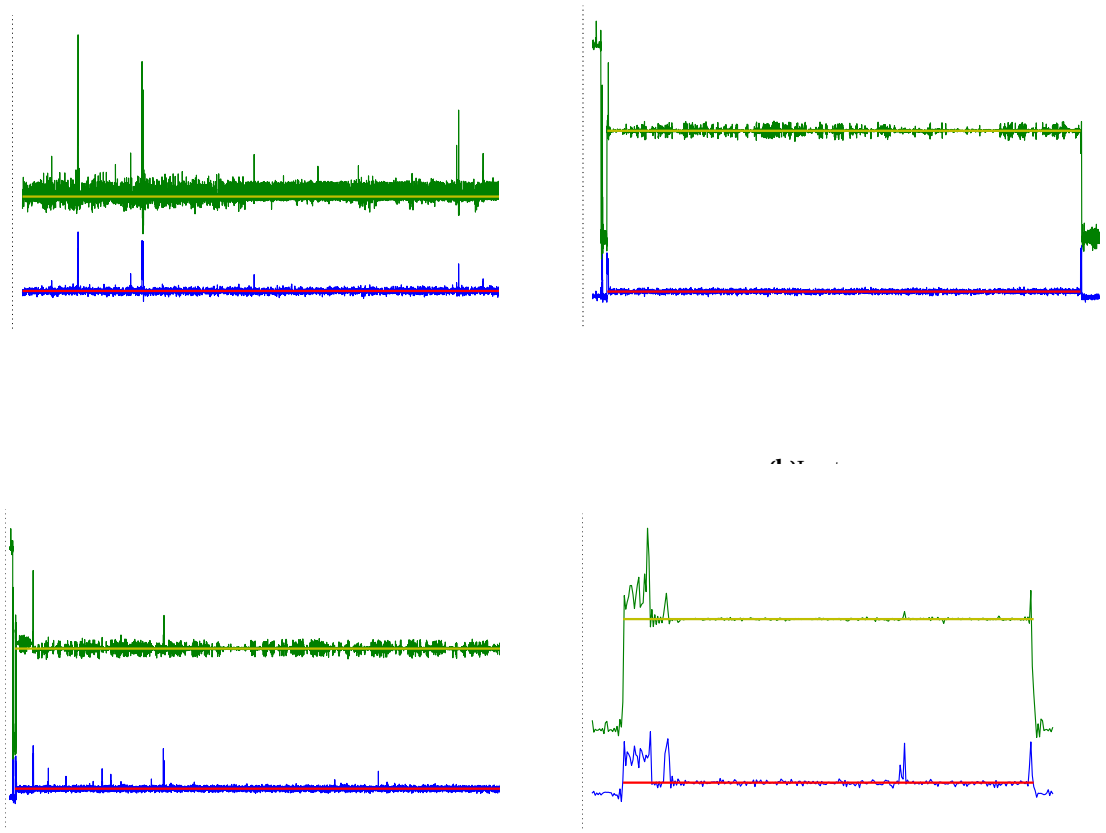
Pour ce qui est de la consommation durant ces opérations, on effectue l'expérience présentée ci-dessous.

• **Objectifs :** observer la consommation énergétique lors d'accès flash au niveau pilote.

• **Méthode :** on mesure la consommation sur le rail d'alimentation du CPU et de la puce mémoire en fonction du temps, pendant (A) la lecture de 32768 pages, (B) l'écriture de 32768 pages et (C) l'effacement de 512 blocs. La puissance à vide est également mesurée. On rappelle que sur la carte Omap la RAM et la flash partagent le même rail d'alimentation.

• **Résultats :** la figure 3.16 présente l'évolution de la puissance mesurée.

La figure 3.16 montre les points suivantes : premièrement, on constate clairement des paliers de puissance de valeur relativement stable pendant la boucle d'accès flash. Les pics de puissance en début et fin de test correspondent au chargement et fermeture du programme de test, et n'ont pas de rapport avec les accès en mémoire flash. Le sur-coût par rapport à la consommation à vide sur la puce mémoire est assez faible : environ 0.025 W (15 % de la puissance à vide sur la puce mémoire) pour l'écriture et l'effacement, et 0.017 W (11 %) pour la lecture. Au niveau du CPU, le coût est plus important : environ 0.15 W (56 % de la puissance à vide sur le CPU) pour l'effacement et l'écriture, et 0.2 W (74 %) pour la lecture.



(c)Écriture **(d)**Effacement

FIGURE 3.16 – Consommation constatée sur la puce mémoire (flash + RAM) et le CPU de la carte omap à vide (a), et lors de l'application au niveau pilote d'opérations flash de base : lecture de page (b), écriture de page (c) et effacement de bloc (d).

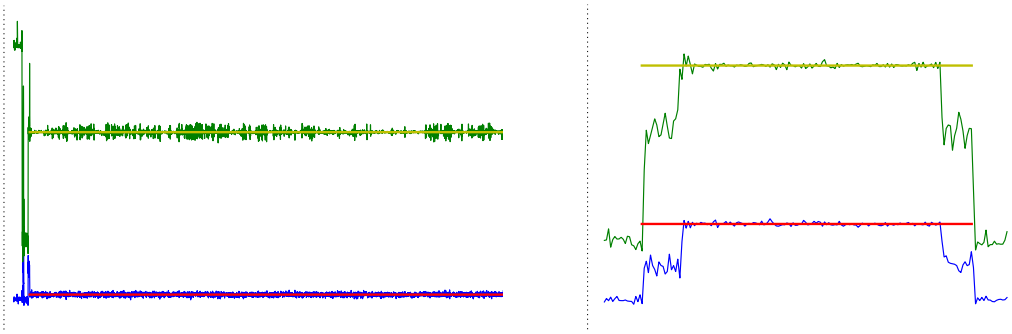


Figure 3.17 – Impact du buffer de lecture MTD sur la consommation

b) Impact du buffer de lecture MTD

Comme indiqué précédemment le pilote MTD maintient en RAM un buffer de lecture de la taille d'une page de la mémoire flash sous-jacente. Ce buffer contient les données de la dernière page lue.

• **Objectif :** évaluer l'impact du bufferde lecture MTD buffer sur les performances.

• **Méthode :** on utilise une version modifiée du micro-benchmark en lecture : cette version accède 32768 fois à la même page de la mémoire flash. On compare les résultats donnés par ce benchmark avec ceux du benchmark concernant la lecture séquentielle. Pour ce qui est des performances, les résultats sont présentés dans la table 3.1.

• **Résultats :** On peut voir qu'accéder 32768 fois à la même page est une opération beaucoup plus rapide qu'accéder à 32768 pages différentes. Sur la carte Omap, une page flash fait 2 Ko donc 32768 pages flash représentent un ensemble de 64 Mo de données. Dans l'expérience faisant un usage intensif du buffer de lecture, on constate donc des débits largement supérieur a ceux d'une mémoire flash. La trace des opérations flash effectuées via Flashmon confirme le fait que la page flash est lue une seule fois.

Opération (niveau pilote)	Temps d'exécution (s)	Débit (Mo/s)
Lecture de 32768 pages flash différentes	5.49	11.6
32768 lectures de la même page flash	0.16	400.0

Table 3.1 – Impact du buffer de lecture MTD sur les performances

Concernant la consommation, la figure 3.17 présente l'évolution de la puissance en fonction du temps pour les deux expériences, sur la puce mémoire et le CPU de la carte Omap. On retrouve la forte hausse de performances, et on peut également voir que la puissance sur le CPU et la puce mémoire est plus importante lors de l'utilisation du buffer. On constate une hausse de 0.13 W sur la puce mémoire (88 % de la puissance à vide sur cette puce), et 0.12 W sur le CPU (45 % de la puissance à vide).

Au niveau matériel, on peut donc dire que la consommation et les performances du système de stockage dépendent :

— des caractéristiques du matériel : latences des opérations flash, puissance moyenne constatée lors de chacune des opérations sur la mémoire flash, la RAM et le CPU ;
— du nombre d'opérations réalisées par une charge donnée ;
— d'un overhead en temps et en consommation, ajouté par le pilote lui-même ;
— de la localité des accès en lecture de la charge, qui vont faire un usage plus ou moins intensif du buffer de lecture.

3.2 Niveau de gestion FFS - focus sur JFFS2

Le pilote NAND, vu dans la section précédente, est utilisé par le FFS pour effectuer des accès flash. Dans cette sous-section on explore l'impact de la partie FFS de la couche de gestion sur les performances et la consommation. Comme dit précédemment on s'intéresse particulièrement aux opérations de lecture et d'écriture de données dans des fichiers. On prend l'exemple du FFS JFFS2. Il s'agit donc des fonctions *jffs2_readpage()* (lecture), *jffs2_write_begin()* et *jffs2_write_end()* (écriture). Ces trois fonctions sont les interfaces entre VFS et le système de fichiers JFFS2. Au niveau MTD on a observé un overhead de performances et de consommation, dû au pilote, par rapport au latences et valeurs de consommation au niveau matériel (puce flash). Le FFS rajoute lui aussi un overhead par rapport au niveau pilote.

a) Lecture de données dans un fichier : la fonction *readpage()*

jffs2_readpage()

• **Objectif :** on souhaite explorer les performances de JFFS2 en lecture.

• **Méthode :** pour ce faire, une première expérience consiste à mesurer les temps d'exécution de la fonction *jffs2_readpage()* lors de l'application, tout en effectuant des accès simples (lecture séquentielle et aléatoire dans un fichier). L'expérience suivante est effectuée : un fichier de taille 400 Ko est créé sur une partition JFFS2. Une taille de 400 Ko correspond a 100 pages Linux, ce qui permet d'effectuer lors de l'expérience un nombre relativement important de lectures de pages Linux différentes. Le fichier est créé sur une partition JFFS2 nouvellement effacée pour éviter tout risque de perturbation du ramasse-miettes. Le fichier est créé et écrit séquentiellement avec des données aléatoires via *write()*, page (Linux) par page. Ce fichier est alors lu séquentiellement, page par page, via un programme de l'espace utilisateur effectuant des appels à *read()* dans une boucle. On rappelle que la taille d'une requête *read()* importe peu car elle est ramenée à une page Linux au niveau de la fonction *readpage()*. A noter que dans une grande partie des expériences présentées dans ce document la taille de requête est fixée à 4 Ko car il s'agit de la taille d'une page Linux. Le page cache est vidé avant le lancement du test pour supprimer son impact. Les tests sont lancés sur la carte Omap. En utilisant Flashmon et Funcmon, on mesure le temps d'exécution de la fonction *jffs2_readpage()*, et le nombre de pages flash lues par chaque appel à cette fonction. On trace également les indices des pages flash physiques lues. Les accès dans le fichier sont réalisés (A) en séquentiel et (B) en aléatoire.

• **Résultats :** Les résultats sont présentés sur la figure 3.18. Sur la gauche on constate voir les résultats concernant les accès séquentiels. Concernant le temps d'exécution de *jffs2_readpage()*, on peut voir que la plupart des appels prennent environ 400 μs. On constate quelques pics autour de 500 μs, et des pics autour de 600 μs. En observant le nombre de pages flash lues lors de chaque appel à *jffs2_readpage()*, on observe que la plupart des appels à cette fonction déclenchent la lecture de deux pages flash. Certains appels déclenchent néanmoins la lecture de trois pages flash : ces évènements correspondent aux pics à 600 μs. En regardant les indices des pages flash physiques lues, on peut voir que JFFS2 effectue des lectures séquentielles. On constate quatre sauts de blocs (changements brutaux d'indices de pages lues sur la courbe en bas à gauche de la figure 3.18), montrant que le fichier est réparti sur cinq blocs flash. Cela signifie que les nodes JFFS2 composant le fichier lu sont réparties séquentiellement sur la flash, ce qui est normal car le fichier a été écrit séquentiellement sur une partition nouvellement créée.

Toujours en accès séquentiel, si l'on revient sur les pics à 600 μs, on peut constater qu'ils sont périodiques. L'explication à cette periodicité est la suivante : le fichier étant écrit page (Linux) par page, la plupart des nodes JFFS2 le composant contiennent les données correspondant à l'intégralité d'une page Linux de ce fichier. La page Linux 0 du fichier est contenue dans une node, la page 1 dans une autre node, etc. Dans le cas de notre expérience, *jffs2_readpage()* lit donc une seule node par appel. Ces nodes sont réparties séquentiellement dans les blocs les contenant. La taille de ces nodes est

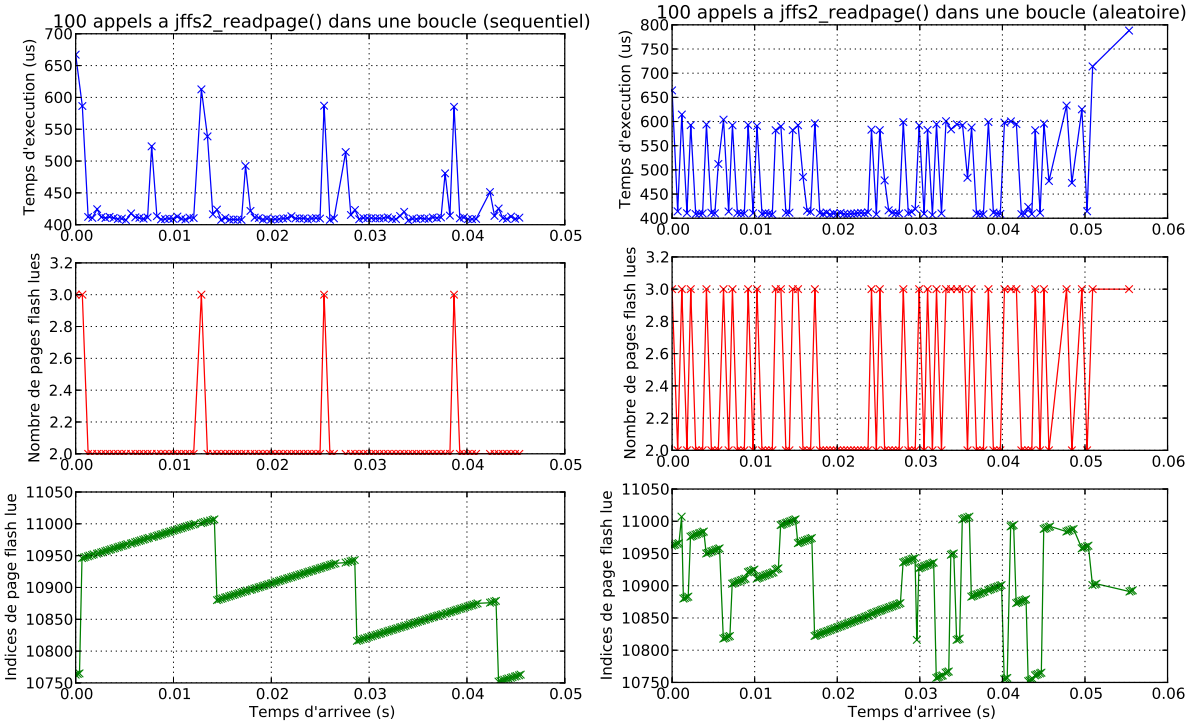


Figure 3.18 – Exploration des performances de *jffs2_readpage()* : A gauche sont présentés les accès séquentiel, à droite les accès aléatoires. En haut on peut voir le temps d'exécution de chaque appel à *jffs2_readpage()* en fonction de son temps d'arrivée, au centre le nombre de page flash lues par chaque appel à *jffs2_readpage()*, et en bas les indices de pages flash lues.

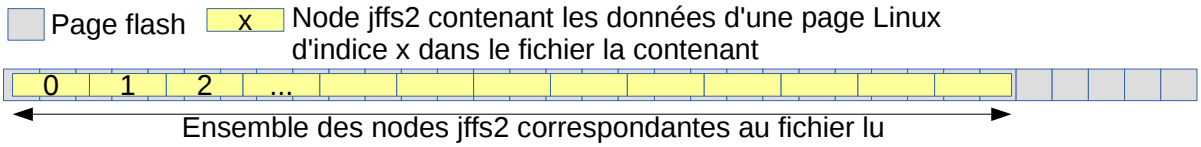


Figure 3.19 – Répartition en flash des nodes JFFS2 d'un fichier écrit séquentiellement page (Linux) par page sur une partition relativement vide.

d'environ 4 Ko : elles contiennent 4 Ko de données plus quelques dizaines d'octets de métadonnées composant l'entête de node JFFS2. On a donc une répartition des nodes en flash correspondant à ce qui est schématisé sur la figure 3.19.

On voit donc qu'avec ce type de répartition, une node est généralement répartie sur trois pages flash. La lecture séquentielle des nodes déclenchée par la lecture séquentielle du fichier demandé par l'applicatif va donc provoquer une lecture elle-même séquentielle des pages flash (c'est ce qu'on constate sur la figure 3.18 en bas à gauche). La lecture de la dernière page flash contenant une node provoque le placement dans le buffer de lecture MTD de cette page flash. La lecture de la prochaine node commence alors par un cache hit dans ce buffer car les nodes sont placées séquentiellement en flash. Ainsi, la lecture via *jffs2_readpage()* déclenche en général $3 - 1 = 2$ lectures de pages flash (node répartie sur 3 pages à lire moins un cache hit). Il arrive néanmoins que certaines nodes soient alignées sur le début d'une page : dans ce cas on ne bénéficie pas du buffer de lecture MTD et 3 pages flash sont lues. Ce comportement est illustré sur la figure 3.20, et se vérifie via la figure 3.18 au centre à gauche).

Concernant les accès aléatoires, les résultats sont présentés sur la figure 3.18 à droite. On peut noter que le nombre de pics aux alentours de 600 μs est beaucoup plus important. En effet, le nombre d'appels à *jffs2_readpage()* déclenchant 3 lectures de pages flash est beaucoup plus important. Cela est

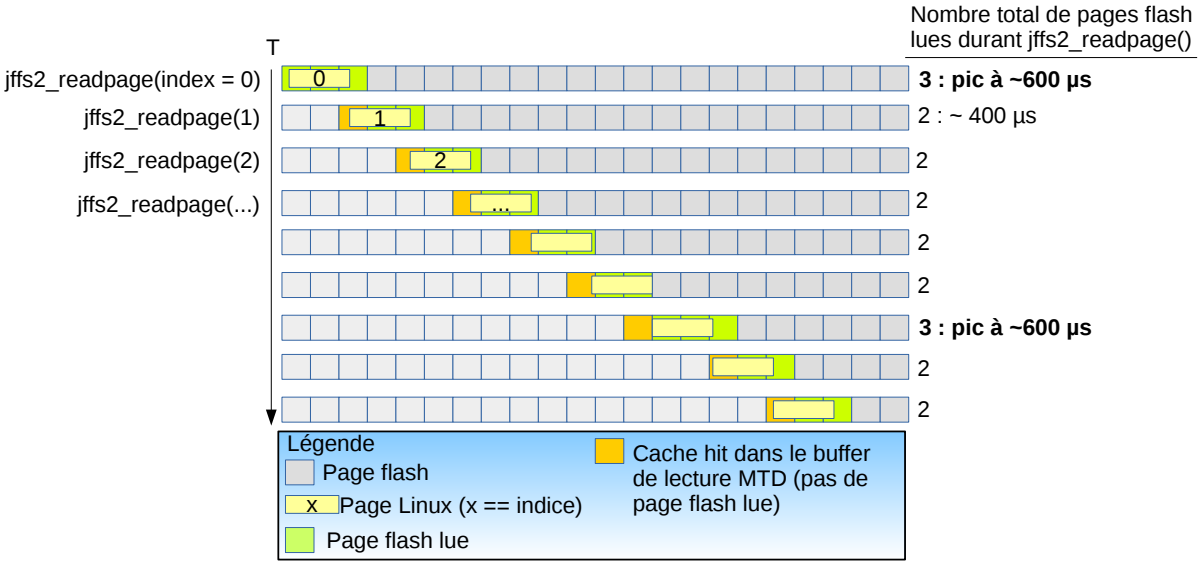


FIGURE 3.20 – Lecture séquentielle d'un ensemble de nodes JFFS2 réparties comme définit sur la figure 3.19

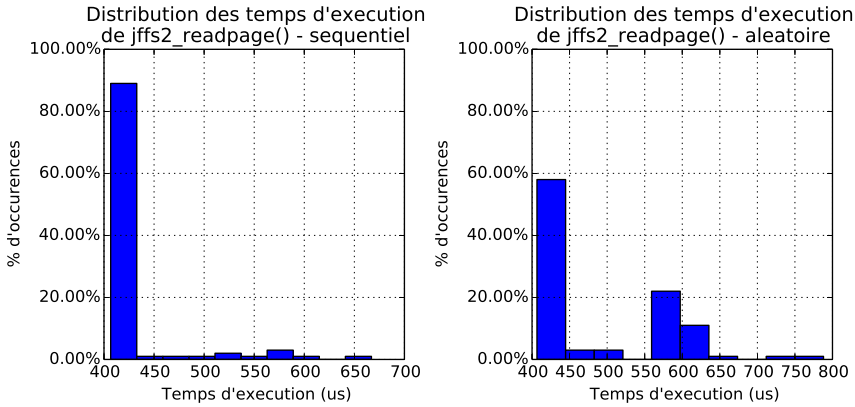


FIGURE 3.21 – Distribution des temps d'exécution de *jffs2_readpage()* en séquentiel (gauche) et aléatoire (droite)

dû au fait que le comportement vu en accès séquentiel ne s'applique pas ici : les accès au niveau flash (voir 3.18 en bas à droite) étant moins séquentiels, le buffer de lecture MTD est beaucoup moins utilisé. On peut voir que le temps d'exécution total est également plus important en aléatoire : environ 0.055 secondes contre environ 0.045 secondes en séquentiel.

Pour ce qui est des pics aux alentours de 500 μs, présents en séquentiel comme en aléatoire, il faut noter que leur impact est négligeable. On peut voir sur la figure 3.21 les distributions des temps d'exécution de *jffs2_readpage()* pour les deux expériences (séquentiel / aléatoire). On voit que le nombre de pics à 500 μs représente un pourcentage négligeable (inférieur à 5 %) des temps d'exécution.

Concernant la consommation, on effectue l'expérience présentée ci-dessous.

- **Objectif :** observer la consommation lors d'accès en lecture à JFFS2.
- **Méthode :** on mesure la puissance sur la carte Omap lors de la lecture séquentielle et aléatoire de 3000 pages Linux dans un fichier JFFS2. Ce nombre assez important de 3000 est choisit pour effectuer un test prenant un temps relativement important, et ainsi obtenir une courbe de consommation exploitable.
- **Résultats :** les résultats sont présentés sur la figure 3.22. En plus des différences en temps d'exécution, on peut constater des pics de consommation réguliers en séquentiel, qui ne sont pas présent en aléatoire. Ces pics en séquentiel correspondent à des *accès RAM* fréquents. En effet, en séquentiel

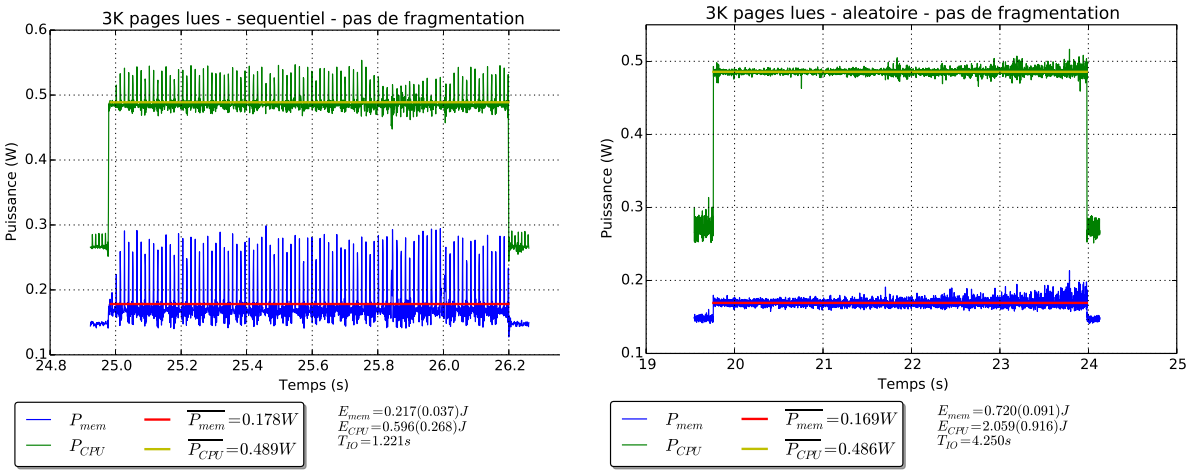


Figure 3.22 – Évolution de la puissance en fonction du temps lors de lectures séquentielle (gauche) et aléatoire (droite) de fichiers JFFS2

read-ahead rentre rapidement en régime permanent. Dans le cas de JFFS2 la quantité maximale de données que read-ahead peut précharger est 32 pages Linux (128 Ko). Des paquets de 32 pages Linux sont donc préchargés régulièrement depuis la flash (temps entre deux pics), et sont suivis de 32 cache hits dans le page cache (pics). Lorsqu'une page Linux est lue par un processus depuis le page cache, une copie de la page est réalisée en RAM : la page présente dans le page cache est située dans l'espace mémoire réservé au noyau, inaccessible au processus s'exécutant lui dans l'espace utilisateur. Une copie est donc réalisée vers l'espace d'adressage du processus (espace utilisateur), dans le buffer utilisateur passé en paramètre à l'appel système *read()* au moment de son invocation. Cela est fait au niveau VFS, dans la fonction *do_generic_file_read()* appelée par *vfs_read()* présentée plus haut (voir section 1.2 de ce chapitre, page 87), via la fonction *copy_to_user()*. Les pics de consommation constatés sont donc dus à des séries de 32 copies mémoires très rapprochées dans le temps.

La raison pour laquelle les copies mémoires ne sont pas visibles sur la courbe concernant les accès est la faible fréquence d'échantillonage du système de mesure de consommation. En effet, même si le nombre de cache hits est moins important car read-ahead est moins efficace en accès aléatoire, dans cette expérience il y a autant de copies mémoire entre l'espace noyau et l'espace utilisateur que dans l'expérience en séquentiel. Néanmoins, il s'agit de copies d'un faible nombre de pages, plus espacées dans le temps. Il s'agit d'opérations très rapides, elles ne sont donc pas visibles. On peut les observer en augmentant la fréquence d'échantillonage à 100 kHz. Cela est présenté sur la figure 3.23, représentant l'expérience en accès aléatoires. Une haute fréquence d'échantillonage rend les courbes moins lisibles, et difficilement traitables car le nombre d'échantillons croît avec la fréquence. Ainsi, la majeure partie des résultats présentés ici sont réalisés à 1 kHz.

Lecture : impact de la fragmentation du fichier lu Dans les expériences précédentes, le fichier lu était créé sur une partition vide, et écrit séquentiellement page par page. Cela résulte en une répartition très ordonnée sur la flash : les nodes JFFS2 contiennent des pages Linux entières, et elles sont réparties séquentiellement dans les blocs les contenant. On réalise l'expérience suivante :

- **Objectif :** explorer l'impact sur les performances de la répartition d'un fichier en mémoire flash.
- **Méthode :** sur une partition JFFS2 nouvellement créée après effacement, deux fichiers de 250 Ko sont créés : un fichier non-fragmenté est créé via l'écriture de données en séquentiel, page Linux par page Linux. Le deuxième fichier est fragmenté, il est construit comme le précédent, puis on écrit dans ce fichier à des offsets aléatoires 500 paquets de données, de 512 octets chacun. Cela a pour effet de créer une fragmentation très importante des nodes JFFS2 en flash, comme illustré sur la figure 3.24

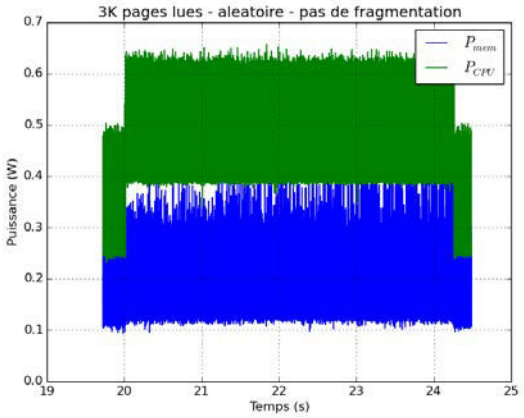


FIGURE 3.23 – Lecture aléatoire de 3000 pages Linux dans un fichier, mesures de consommation à une fréquence de 100 kHz

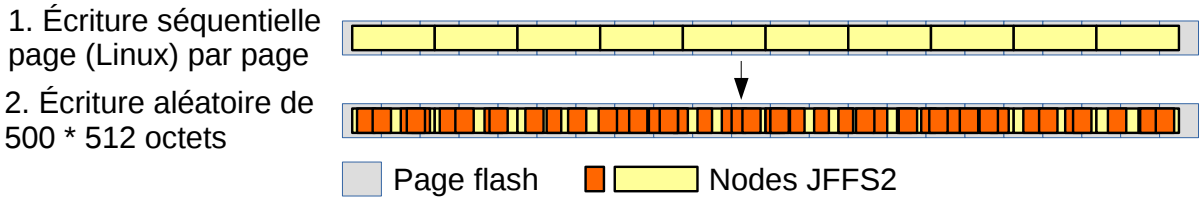


FIGURE 3.24 – Création d'un fichier JFFS2 fragmenté : le fichier est premièrement créé séquentiellement page par page (point 1.), puis mis à jour à des offsets aléatoires par paquets de données de petites taille.

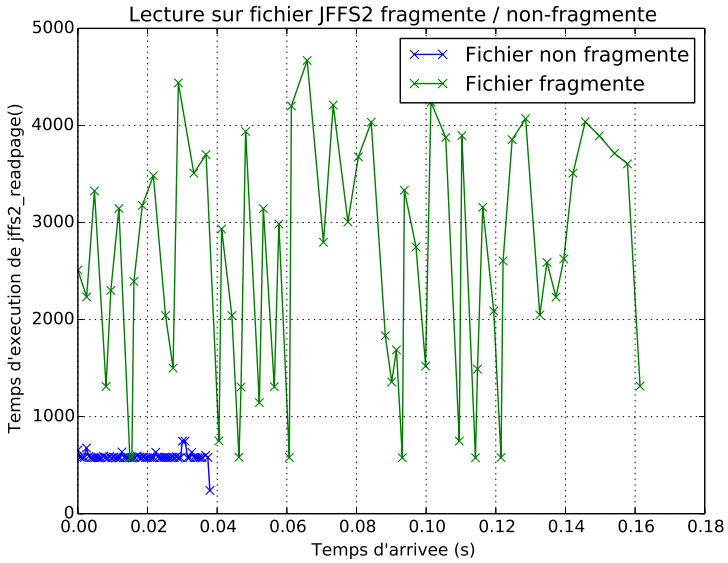


FIGURE 3.25 – Lecture séquentielle, page (Linux) par page, d'un fichier non fragmenté et d'un fichier fragmenté

On lit alors page (Linux) par page chacun des fichiers et on observe la différence de performances.
• **Résultats :** les résultats sont présentés sur la figure 3.25. Cette figure présente le temps d'exécution de chaque appel à *jffs2_readpage()* tracé en fonction de son temps d'arrivée (axe des x). On voit clairement que le temps d'exécution total est bien plus important pour la lecture du fichier fragmenté : 0.16 secondes contre 0.04 pour le fichier non fragmenté. On constate également un temps d'exécution des fonctions *jffs2_readpage()* (1) généralement important et (2) très variable pour le fichier fragmenté : cela s'explique par le fait que le nombre de nodes composant chaque page Linux à lire est important et variable dans le cas du fichier fragmenté. Le nombre de pages flash à lire est donc également important et variable. L'analyse des traces de Flashmon confirme le fait qu'un nombre bien plus important de pages flash est lu pour chaque appel à *jffs2_readpage()* dans le cadre de l'expérience sur fichier fragmenté. Concernant la consommation lors de la lecture d'un fichier fragmenté, on constate une courbe relativement similaire à celle du fichier non fragmenté (voir figure 3.22), mais le temps de l'opération est bien entendu plus important.

Pour conclure sur la lecture au niveau FFS, dans le cas de JFFS2, le facteur principal qui influence les performances et la consommation est le nombre de pages lues par appel à *jffs2_readpage()*. Il dépend du mode d'accès (séquentiel / aléatoire) qui fait un usage plus ou moins important du buffer de lecture MTD, et de la répartition (fragmentation) du fichier en nodes sur la flash.

b) Écriture de données dans un fichier : les fonctions *write_begin()* et *write_end()*

jffs2_write_begin()* et *jffs2_write_end() *jffs2_write_begin()* et *jffs2_write_end()* sont les deux fonctions appelées lors de l'écriture de données au sein d'une page Linux d'un fichier.
• **Objectif :** on souhaite explorer l'impact de la taille de requête sur les performances de ces deux fonctions.
• **Méthode :** on sait en effet que le write buffer de JFFS2 peut tamponner les écritures de taille inférieure à la taille d'une page flash. On réalise l'expérience suivante : grâce à l'infrastructure de trace développée, les temps d'exécution de ces deux fonctions sont tracées, ainsi que le nombre de pages flash écrites durant l'exécution de *jffs2_write_end()* (*jffs2_write_begin()* ne déclenche pas d'écriture en flash). Les benchmarks suivants sont lancés :

1. Écriture séquentielle de 100 pages Linux dans un fichier nouvellement créé. A la fin de l'expérience la taille du fichier est donc de $100 * 4 = 400$ Ko ;
2. Écriture séquentielle de 800 paquets de 512 octets dans un fichier nouvellement créé, ce qui donne également un fichier de 400 Ko au final.

La taille de 400 Ko est choisie pour que lors de l'expérience on puisse réaliser un nombre relativement important d'appels à *jffs2_write_begin()* et *jffs2_write_end()*.
• **Résultats :** Les résultats sont présentés sur la figure 3.26. On peut voir que le temps d'exécution de *jffs2_write_begin()* varie entre 5 et 30 μs. Il s'agit donc d'une opération très rapide, notamment par comparaison avec *jffs2_write_end()* vu dans le paragraphe ci-dessous. Par conséquent l'on ne s'attarde pas sur les performances de *jffs2_write_begin()*.

Concernant *jffs2_write_end()*, on peut voir que le temps d'éxécution de cette fonction dépend fortement du nombre de pages flash écrites : pour les requêtes de 4 Ko, on peut voir que la majeure partie du temps, deux pages flash sont écrites. Cela correspond à un temps d'exécution d'un peu moins d'une milliseconde. A intervalles réguliers, trois pages flash sont écrites. Cela est dû au phénomène vu dans la partie précédente traitant des lectures (voir les figures 3.19 et 3.20 page 105). Lorsque trois pages sont écrites on constate un temps d'exécution pour *jffs2_write_end()* d'environ 1.3 ms.

Concernant les écritures de 512 octets, on peut faire le même constat concernant l'impact du nombre de pages flash écrites. On peut également voir que bon nombre d'appels à *jffs2_write_end()* ne déclenchent aucune écriture de page flash : il s'agit de l'impact du buffer d'écriture de JFFS2 : lorsqu'une écriture au sein d'une page Linux est réalisée avec une taille inférieure à la taille d'une

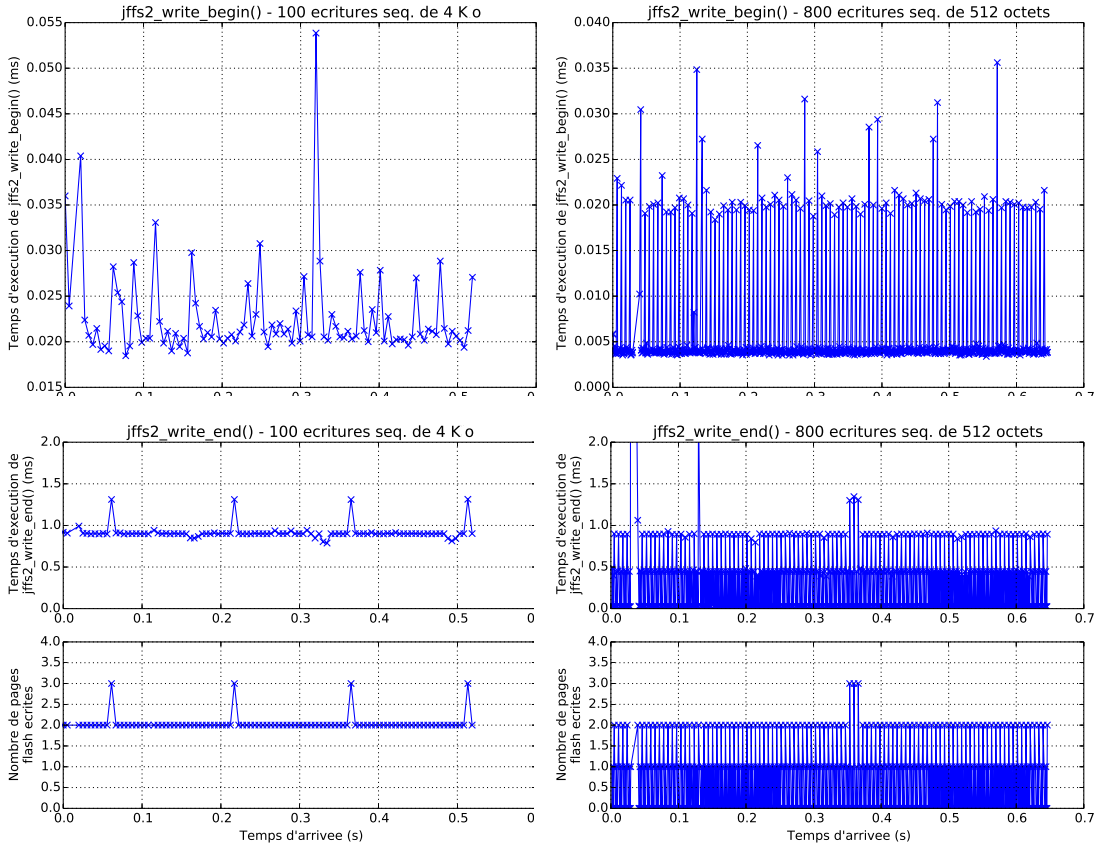


FIGURE 3.26 – Exploration des performances de *jffs2_write_begin()* et *jffs2_write_end()* : A gauche les résultats pour 100 écritures de 4 Ko, à droite 800 écritures de 512 octets. En haut on peut voir les performances de *jffs2_write_begin()*, au centre les performances de *jffs2_write_end()*, et en bas le nombre de pages flash écrites pour chaque appel à *jffs2_write_end()*.

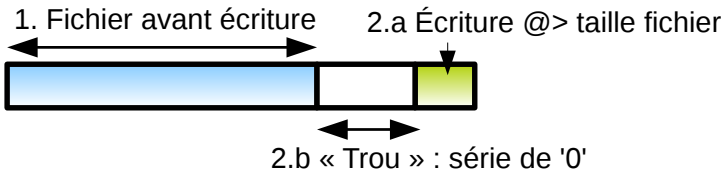


Figure 3.27 – Création d'un "trou" suite à un accès en écriture à un offset supérieur à la taille d'un fichier.

page flash, les données sont empilées dans le *write buffer* de JFFS2 si la quantité d'espace libre au sein de ce buffer est suffisante (voir description fonctionnelle de l'opération d'écriture JFFS2, section b) page 84). On voit dans ce cas que le temps d'exécution de *jffs2_write_end()* est très faible. On sait que lorsque le buffer d'écriture est plein une page flash est écrite. On peut alors se demander pourquoi on constate dans l'expérience concernant les requêtes de 512 octets plusieurs écritures de 2 voir 3 pages flash. Cela vient du fait que lorsque JFFS2 constate l'écriture du dernier octet d'une page Linux, quelle que soit la taille de la requête, la page Linux entière est re-écrite : cela génère donc l'écriture de 4 Ko de données en flash qui peut se traduire en deux ou trois écritures de pages (une page fait 2 Ko sur notre plate-forme matérielle) selon plusieurs facteurs comme la compression et la taille de l'espace libre dans le *write buffer*. Cela est fait pour limiter la fragmentation d'un système de fichiers JFFS2.

On ne constate pas de différence de performances lors de l'écriture dans des fichiers à des offsets aléatoires. Cela est plutôt normal quand on sait que JFFS2 effectue des écritures séquentielles en flash quelque soit le mode d'accès en écriture au niveau logique. On peut diviser les accès aléatoires sur un fichier JFFS2 en deux catégories :

1. Les accès aléatoires *sans écrasement* : il s'agit d'écritures en aléatoire sur un fichier nouvellement créé de taille nulle, ou encore d'accès aléatoire à un offset supérieur à la taille du fichier. Ce sont là des accès assez particuliers, c'est pourquoi on ne s'y attarde pas dans ce travail. Linux définit, dans le manuel de l'appel système *lseek()* (Linux Contributors, 2014), la gestion suivante pour ce type d'accès : ils sont autorisés et déclenchent l'apparition d'un "trou" entre l'offset du dernier octet du fichier avant l'écriture, et l'offset du premier fichier écrit. La lecture des données du trou se traduit par une série de '0'. Cela est illustré sur la figure 3.27. JFFS2 gère cela en écrivant pour le fichier une node de méta-données de petite taille représentant le trou ;
2. Les accès aléatoires écrasant des données dans un fichier : il s'agit de mises à jour (*overwrite*). Il n'y a pas de différences au niveau des performances des fonctions d'écriture lors de ces accès, néanmoins :
   — Cela crée de la fragmentation en fonction de la manière dont les données logiques contenues dans les nouvelles nodes viennent chevaucher les données des anciennes nodes partiellement ou totalement écrasées. La fragmentation impacte les performances en lecture comme vu en section précédente ;
   — Cela crée des données invalides ce qui mène à l'exécution du ramasse-miettes, dont l'impact sera vu dans les paragraphes suivants.

Concernant la consommation on effectue l'expérience suivante :
• **Objectif :** explorer la consommation d'accès en écriture avec JFFS2.
• **Méthode :** on effectue des écritures séquentielle de (A) 1280 requêtes de 4 Ko et (B) 10200 requêtes de 512 octets dans un fichier. Comme pour l'expérience en lecture les nombres de requêtes sont importants pour obtenir une courbe de consommation lisible. La taille totale du fichier après chacune des expérience est donc de $1280 * 4096 = 10240 * 512 = 5$ Mo.
• **Résultats :** la figure 3.28 présente l'évolution de la puissance en fonction du temps sur le CPU et la puce mémoire de la carte Omap. Tout comme pour les expériences en lecture, on constate des paliers de consommation lors des accès, et une valeur de puissance moyenne qui n'est pas affectée par la taille de requête.

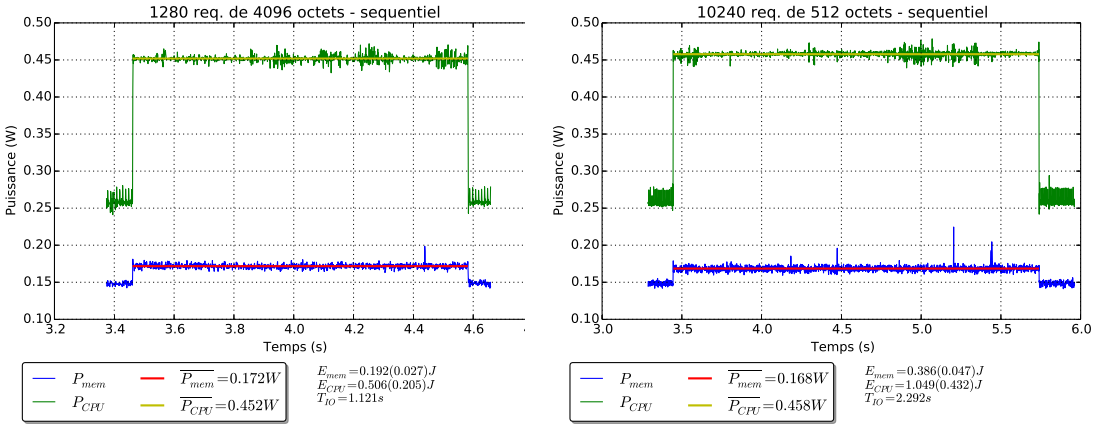


FIGURE 3.28 – Consommation lors de l'écriture séquentielle de fichiers, via 1280 requêtes de 4 Ko (gauche) et 10240 requêtes de 512 octets (droite)

Écriture : ramasse-miettes et état de départ Le ramasse-miettes de JFFS2 est lancé pour recycler l'espace flash invalide, effaçant des blocs pour créer de l'espace libre. On peut différencier deux modes de fonctionnement :

1. Le *fonctionnement du ramasse-miettes en arrière plan* : par le biais d'un thread noyau Linux s'exécutant avec une faible priorité, le ramasse-miettes en arrière plan recycle des blocs pendant les périodes où il n'y a pas d'E/S utilisateur. Par conséquent ce mode d'exécution n'a pas d'impact sur les temps de réponse des E/S ;
2. L'exécution du ramasse-miettes sous *seuil d'espace libre critique* : lorsque JFFS2 souhaite écrire une node en flash, une vérification du seuil d'espace libre est effectué (il s'agit de la réservation d'espace présentée sur la figure 3.5 page 84). Dans ce cas le ramasse-miettes vient retarder la requête d'écriture qui a déclenché l'écriture de la node, car il doit être exécuté en premier lieu : il y a donc un impact sur les temps de réponse.

Bien que le ramasse-miettes *en arrière plan* n'ait pas d'impact sur les performances, il impacte la consommation. Pour observer cet impact on réalise l'expérience suivante :

• **Objectif :** observer l'impact sur la consommation du ramasse-miettes JFFS2 en arrière plan.

• **Méthode :** on génère une quantité importante de données invalides sur une partition JFFS2. Un fichier est créé, écrit séquentiellement page Linux par page Linux. Une fois le fichier créé, ses données sont écrasées via une série de requêtes d'écriture à des offset aléatoire. La taille du fichier est de 5 Mo, et les requêtes d'écriture aléatoires sont au nombre de 1280, chacune de taille 4 Ko. On mesure la consommation pendant et après l'expérience.

• **Résultats :** Les résultats sont présentés sur la figure 3.29. Après l'expérience, une série de pics de consommation correspondant à l'exécution du ramasse-miettes est observée, sur le CPU et la puce mémoire. Il faut noter que ce régime continue pendant 40 secondes après l'expérience (la fin du régime n'est pas présente sur la figure). Ces pics correspondent à l'exécution des algorithmes du ramasse-miettes, en particulier aux accès flash réalisés dans ce cadre : lecture des données toujours valide sur les blocs victimes, écriture de ces données dans le bloc courant, et effacement des blocs victimes. Ce comportement est validé en traçant avec Flashmon les accès flash pendant cette phase de ramasse-miettes en arrière plan.

Si le ramasse-miettes en arrière plan n'altère pas de manière négative les performances des E/S utilisateur, il faut néanmoins noter la chose suivante : dans le cas d'un fichier fragmenté sur la flash, les nodes toujours valides déplacées dans le cadre du GC peuvent être fusionnées en une nouvelle node (de taille maximale 4 Ko). Le ramasse-miettes joue ainsi un rôle de défragmenteur. Les règles suivantes s'appliquent :

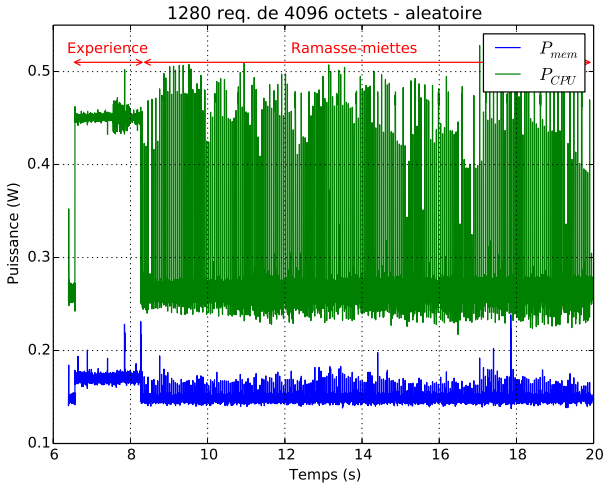


Figure 3.29 – Exécution du ramasse-miettes en arrière plan après une expérience générant une quantité importante de données invalidées.

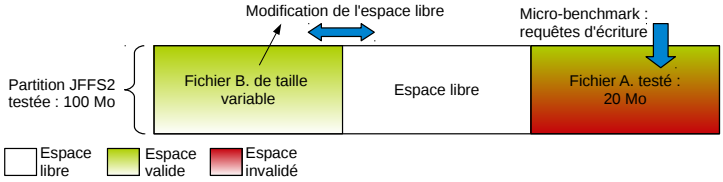


Figure 3.30 – Méthodologie pour l'expérience concernant l'exécution du ramasse-miettes sous seuil d'espace libre critique.

— Les nodes fusionnées doivent être contiguës du point de vue logique ;
— Les nodes fusionnées doivent être présentes sur le même bloc flash, qui est le bloc victime considéré par le ramasse-miettes. Dans le cas contraire, comme la fusion provoque l'invalidation des nodes fusionnées, le ramasse-miettes se verrait créer de l'espace invalide ce qui est contraire à son but primaire.

Pour vérifier cela, on peut observer le temps pris par une lecture du fichier considéré dans l'expérience précédente présentant l'impact du ramasse-miettes en arrière plan sur la consommation. Ce fichier est lut séquentiellement, page (Linux) par page via *read()*. La lecture est effectuée (1) juste après l'écriture aléatoire, c'est à dire avant que le ramasse-miettes n'ai eut le temps de procéder à la défragmentation, et (2) après la défragmentation. Le temps mesuré pour (1) est de 0.83 secondes, et pour (2) on mesure 0.70 secondes soit 18% de différence. Via Flashmon on constate que le nombre de lecture de pages flash est réduit dans le cas de (2).

Dans le cas du ramasse-miettes exécuté *sous seuil d'espace libre critique*, comme dit précédemment son exécution impacte directement les performances des E/S utilisateur.

• **Objectif :** on souhaite montrer que lorsque l'espace libre est très faible, le ramasse-miettes provoque une augmentation des latences en écriture, et également une répartition de l'usure moins efficace.

• **Méthode :** Pour observer cet impact on réalise premièrement l'expérience suivante : sur une partition JFFS2 de 100 Mo un fichier de 20 Mo (20 % de l'espace flash correspondant à la partition) est créé, écrit séquentiellement page (Linux) par page. On l'appelle le fichier A. Par la suite, on applique dans une boucle un nombre important d'écritures via *write()* à ce fichier, de tailles variables à des offsets aléatoires. Cela crée de l'espace invalide et déclenche l'exécution du ramasse-miettes en arrière plan. Sur la partition on crée également avant l'expérience un fichier B de taille variable : 30, 60 et 75 Mo. Ce fichier représente des données valides et non fragmentées. Cela fait que lors des accès aléatoires sur le fichier A, l'espace libre disponible est variable (en fonction de la taille de B). Il est notamment

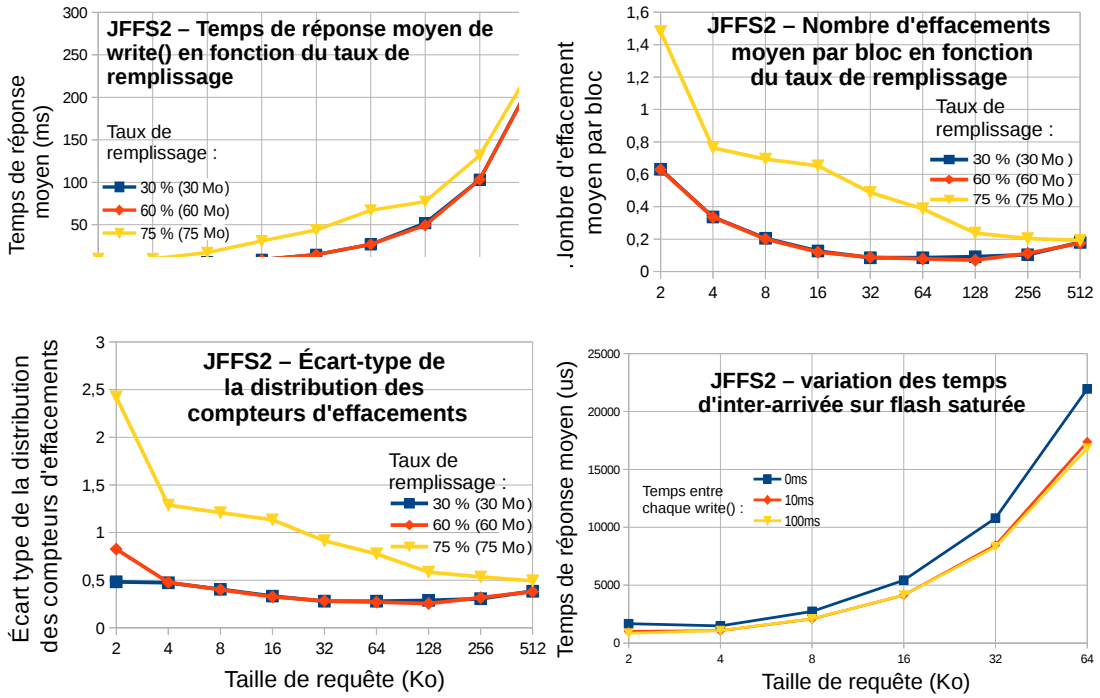


FIGURE 3.31 – Résultats de l'expérience concernant l'exécution du ramasse-miettes sous seuil d'espace libre critique (en haut et en bas à gauche), ainsi que l'expérience ou l'on varie les temps d'arrivée lors de l'exécution du ramasse-miettes en arrière plan (en bas à droite).

très faible dans le cas où B fait 75 Mo : $100 - 20 - 75 = 5$ Mo. On s'attend donc dans ce cas à une exécution du ramasse-miettes sous seuil d'espace libre critique. Le procédé d'expérimentation est illustré sur la figure 3.30. Lors des accès aléatoires sur le fichier A, on mesure le temps d'exécution de chaque appel à *write()*. On mesure également avec Flashmon le nombre d'effacements flash moyen par bloc de la partition, ainsi que la répartition des effacements.

• **Résultats :** Les résultats sont présentés sur les courbes en haut et en bas à gauche de la figure 3.31. On peut voir que si le temps d'exécution moyen de chaque appel à *write()* ne varie pas quand l'espace libre est important, lorsque le taux de remplissage est grand (75 Mo), on constate un temps de réponse moyen supérieur. Cette chute de performances est due à l'exécution du ramasse-miettes sous seuil d'espace libre critique qui retarde les requêtes d'écriture. Le nombre d'effacements est ainsi supérieur, et la répartition de l'usure moins bonne (voir figure 3.31).

Pour explorer les interactions entre les deux modes de ramasse-miettes, une expérience supplémentaire est réalisée.

• **Objectif :** exploration des interactions entre les deux modes du ramasse-miettes JFFS2.

• **Méthode :** une partition JFFS2 de 20 Mo est totalement remplie avec un fichier, qui est effacé juste avant le début de l'expérience, saturant la partition avec de l'espace invalide. On lance alors l'écriture d'un nouveau fichier sur la partition en faisant varier la taille de requête (de 2 à 64 Ko), et le temps entre chaque requête *write()* : 0, 10 et 100 ms. On mesure le temps de réponse moyen de chaque appel à *write()*.

• **Résultats :** Les résultats sont présentés sur la figure 3.31 en bas à droite. On peut voir que lorsque le temps entre chaque *write()* est important (10 et 100 ms), cela permet au ramasse-miettes en arrière plan d'effectuer son travail et de ramener le niveau d'espace libre sous un seuil, ce qui évite l'exécution du ramasse-miettes sous seuil critique : en effet on peut voir une latence moyenne supérieure dans le cas des accès sans temps d'inter-arrivée entre les requêtes d'écriture.

Les expériences concernant le ramasse-miettes présentées sur les figure 3.30 et 3.31 sont extraite d'un article traitant de l'impact de l'état de départ de la mémoire flash dans le cadre du benchmarking

de FFS (Olivier et coll., 2012a).

c) Autres sujets d'exploration niveau FFS

D'autres études concernant l'exploration ont également été menées dans le cadre du travail présenté dans cette thèse. Elles ne sont pas détaillées ici car elles concernent d'autres thèmes que les performances et la consommation des opérations de lecture et d'écriture dans les fichiers.

Une étude comparative des performances des systèmes de fichiers JFFS2, YAFFS2 et UBIFS a été réalisée et présentée dans Olivier et coll. (2012c, b). Cette étude cible de multiples métriques de performances relatives aux systèmes de fichiers dédiés aux mémoires flash :

— L'efficacité de la compression (réduction de quantité de données stockées), ainsi que l'impact de la compression sur les performances de diverses opérations comme le temps de montage, la création de fichiers et le listage via la commande *ls* ;
— Le temps de montage ;
— Le temps de création d'arborescences (de fichiers et dossiers) de complexités variées, et les performances des opérations de manipulation d'arborescence comme la recherche d'un fichier ou la suppression de l'arborescence.

Les conclusions de cette étude sont résumées dans la table 3.2

FFS	Opérations sur les fichiers			Compression
	Création	Suppression	Recherche	
JFFS2	-	+	-	+
UBIFS	+	-	-	+
YAFFS2	-	-	+	-

Table 3.2 – Conclusions de l'étude présentée dans Olivier et coll. (2012c)

Le support de la compression par JFFS2 et UBIFS donne un certain avantage à ces FFS par rapport à YAFFS2 qui n'inclut pas cette fonctionnalité : la compression permet de réduire considérablement la taille des données stockées en flash. Concernant les opérations de manipulation de fichiers à haut niveau, l'étude montre que UBIFS donne les meilleurs résultats lors de la création d'arborescences de fichiers et dossiers. L'une des forces de YAFFS2 est la recherche de méta-données : en effet il donne les meilleures performance lors de la recherche d'un fichier dans une arborescence. Concernant le temps de montage, UBIFS domine les autres FFS en donnant un temps de montage plus de 10 fois plus rapide à ceux de JFFS2 et YAFFS2.

d) Exploration niveau FFS : conclusion

On a vu dans cette section que dans le cadre de la lecture et de l'écriture de données dans des fichiers, les accès flash ont un impact majeur sur les performances de ces opérations. Notre étude se focalise sur JFFS2, des tests supplémentaires non présentés ici confirment cette observation pour les autres FFS supportés par Linux (YAFFS2 et UBIFS). Les paramètres impactant le nombre d'accès flash réalisés sont les suivants :

1. Pour les lectures, les accès sont ramenés par VFS au niveau d'une ou plusieurs pages Linux, lues en boucle via la fonction *readpage* du système de fichiers. Ainsi :
   — Une application effectuant une requête de lecture (*read()*) au sein d'une page Linux, de taille inférieure à celle d'une page déclenchera la lecture de la page entière ;
   — Une application lisant une quantité de données supérieure à la taille d'une page, ou lisant une plage de donnée qui chevauche une ou plusieurs frontières de pages déclenchera la lecture de l'ensemble des pages Linux concernées.

L'éventuelle fragmentation du fichier sous forme de nodes en flash joue un rôle important. Au contraire, un fichier séquentiellement réparti sur la flash profite du buffer de lecture MTD ce qui réduit le nombre d'accès en flash. La répartition du fichier en flash est déterminée par la manière dont le fichier a été écrit (ou mis à jour), et la défragmentation effectuée en arrière plan par le thread ramasse-miettes.

2. Pour les écritures, la présence du write buffer JFFS2 tamponne les écritures de petites tailles. C'est donc la quantité de données écrites par l'applicatif, en relation avec l'effet tampon du write buffer, qui détermine le nombre de pages flash écrites. Sous seuil d'espace libre critique, le ramasse-miette insère des latences supplémentaire dans les opérations d'écritures, principalement dues au accès flash correspondant au recyclage des nodes toujours valides.

Au niveau de la consommation, les différentes mesures réalisées permettent d'identifier des valeurs de puissances moyennes pour les opérations de lecture et écriture en flash.

3.3 Niveau VFS

On peut considérer que le niveau VFS est en contact direct avec l'application effectuant des accès au stockage : c'est lui qui reçoit les appels systèmes réalisés par l'applicatif. Après un traitement interne à VFS, ces E/S sont transférées au niveau FFS, vu en section précédente. Tout comme le pilote et le FFS, VFS ajoute aux E/S un overhead de performances et de consommation. On rapelle que le page cache, qui tamponne les E/S dans les fichiers, est maintenu au niveau VFS. On compte deux mécanismes associés au page cache, *read-ahead* et le *write-back*, relatifs respectivement aux accès en lecture et écriture.

a) Impact du page cache

Comme énoncé précédemment, toutes les données écrites ou lues depuis les fichiers sont placées dans le page cache et y restent après l'accès jusqu'à éviction. L'éviction a lieu lorsque la quantité de RAM libre devient faible et que les processus s'exécutant dans le système d'exploitation demande de l'espace mémoire. On peut également vider le page cache via la commande suivante : `echo 1 > /proc/sys/vm/drop_caches`.

Tout accès en lecture sur des données présentes dans le page cache est donc servi depuis la RAM. On souhaite observer l'impact du page cache via une expérience.

• **Objectif :** constater la différence de performance entre (A) la lecture d'un fichier depuis la flash et (B) la lecture du même fichier depuis la mémoire RAM (page cache).

• **Méthode :** un fichier non fragmenté de 12 Mo est créé sur une partition JFFS2. La taille de ce fichier est volontairement grande car on souhaite que l'expérience prennent un temps important. En effet la lecture depuis le page cache est un transfert depuis la RAM, assez rapide, et on souhaite obtenir une courbe de consommation lisible. Le page cache est vidé et le fichier est lu séquentiellement une première fois. Cela a pour effet de charger les données du fichier dans le page cache. Une seconde lecture séquentielle est alors réalisée. La consommation et le temps d'exécution des deux opérations sont mesurées.

• **Résultats :** les résultats sont présentés sur la figure 3.32. On observe clairement les comportements suivants :

— La lecture depuis le page cache est beaucoup plus rapide par rapport à la lecture depuis la mémoire flash. Cela est normal vu les débits de ces deux types de mémoire. Lire 3000 pages Linux depuis le page cache prend 0.06 secondes, contre 1.2 secondes depuis la flash ;

— On constate pendant les accès une puissance supérieure sur la puce CPU et la puce mémoire lors de la lecture depuis le page cache, par rapport à la lecture en flash. Durant cette dernière on constate une puissance moyenne de 0.178 W sur la puce mémoire, et 0.489 W sur le CPU. La lecture depuis le page cache, quant à elle, présente une puissance moyenne de 0.362 W pour la

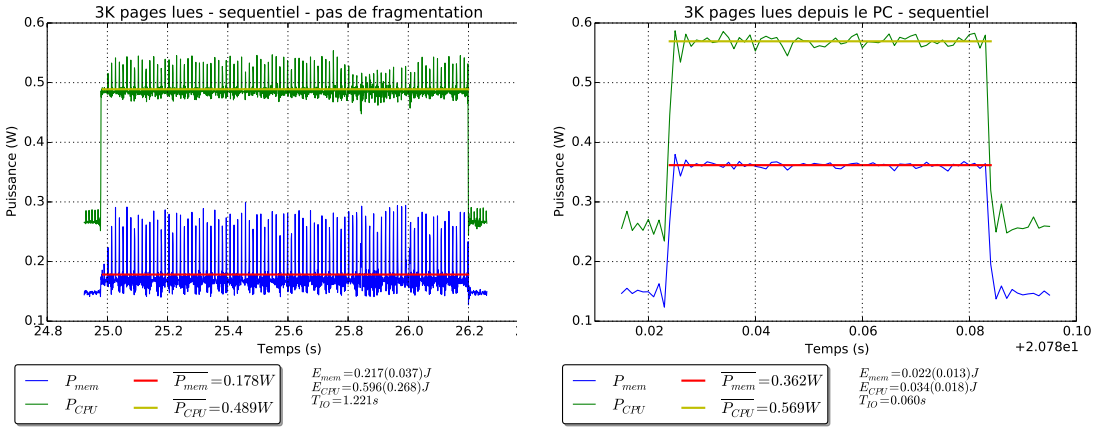


Figure 3.32 – Lecture séquentielle de 3000 pages Linux dans un fichier depuis la mémoire flash (gauche) et depuis la RAM (page cache, droite).

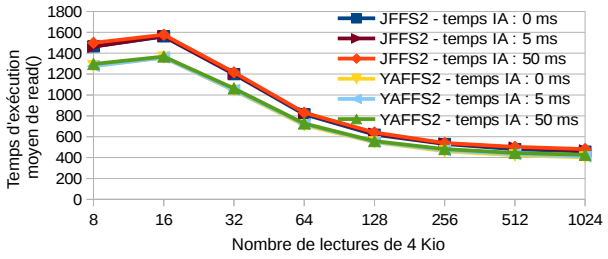


Figure 3.33 – Évolution du temps d'exécution des appels à *read()* en fonction de leur temps d'inter-arrivées.

mémoire et 0.569 W pour le CPU. Néanmoins, le temps d'exécution étant beaucoup plus court lors de la lecture en RAM, l'énergie consommée par cette expérience est largement inférieure à celle de la lecture en flash : 0.013 (mémoire) + 0.018 (CPU) = 0.031 joules contre 0.037 + 0.268 = 0.305 joules (différence d'un facteur 10). A noter que ces valeurs d'énergie sont calculées à partir de valeurs de puissance auxquelles on a soustrait les valeurs de puissance au repos.

On peut donc conclure que, si la consommation (puissance) est supérieure lors des accès RAM, la rapidité des lectures en RAM fait qu'il est plus économique concernant d'énergie de lire depuis la RAM plutôt que depuis la flash. Le page cache apporte donc une économie d'énergie. Cela est notamment dû à la consommation du CPU qui est relativement haute pendant les transferts.

b) Impact de read-ahead

L'impact de read-ahead sur un système géré par FFS a été présenté dans un article (Olivier et coll., 2014b) publié dans les actes d'un workshop. Les résultats présentés ici sont extraits de cet article.

Read-ahead présente un impact particulier sur les performances des accès en lecture sur les fichiers gérés par FFS. On a vu précédemment que l'intérêt principal de read-ahead dans le cas de l'utilisation d'un disque dur est le fait que ces systèmes supportent les E/S asynchrones. La première chose à noter dans le cadre des FFS est le fait que les E/S asynchrones ne sont pas supportées par le pilote NAND MTD. Pour vérifier cela on effectue une expérience simple.

• **Objectif :** vérifier le mode d'exécution synchrone de MTD.

• **Méthode :** un fichier est lu séquentiellement via une boucle d'appels à *read()*. On lance l'expérience 3 fois en variant le temps entre chaque appel à *read()*, égal à 0, 5ms, et 50ms (via *usleep()*). On mesure le temps d'exécution de chaque appel à *read()* avec la primitive *gettimeofday()*. On calcule ensuite le temps d'exécution moyen d'un appel à *read()*. L'expérience est réalisée pour JFFS2, ainsi que pour le FFS YAFFS2. YAFFS2 a été rajouté pour rendre plus complète l'étude présentée dans l'article cité

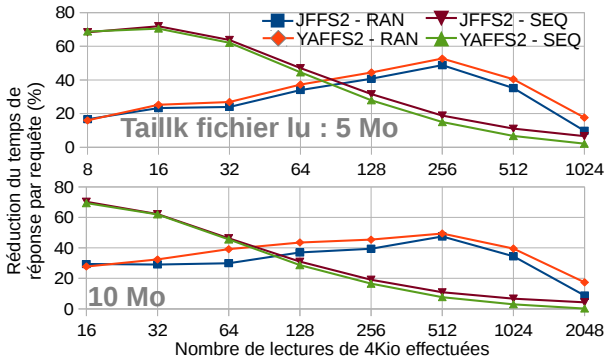


FIGURE 3.34 – Amélioration des performances par la désactivation de read-ahead.

ci-dessus. La taille du fichier est fixée à 5 Mo, et le nombre d'appels à *read()* dans la boucle est varié par puissances de deux, de 8 à 1024 appels. La taille d'une lecture (un appel à *read()*) est de 4 Ko.

• **Résultats :** les résultats sont présentés sur la figure 3.33. En traçant les appels à read-ahead dans le noyau avec VFSmon, on constate que read-ahead est très actif lors de l'expérience. On pourrait donc s'attendre à une augmentation des performances lorsque le temps entre les appels à *read()* est important, ce qui permettrait d'effectuer du préchargement asynchrone. Or, on peut voir que le temps entre chaque appel n'impacte absolument pas les performances. En effet, cela est dû au fait que les appels au driver MTD, et par conséquent les appels à *jffs2_readpage()* et *yaffs2_readpage()* sont synchrones. En d'autres termes, si un appel JFFS2 à *read()* de taille une page Linux déclenche le préchargement de 31 autres pages Linux depuis de FFS, alors la lecture de 32 pages sera réalisée pendant l'exécution de l'appel système *read()*, et bloquera le processus appelant tant que l'intégralité des 32 pages n'est pas présente dans le page cache.

Ce comportement fait que *read-ahead ne peut pas avoir d'effet positif sur les performances des FFS*. Dans le meilleur des cas, lorsque toutes les données préchargées sont effectivement demandées par le processus, read-ahead n'a pas d'impact sur les performances. Dans d'autres cas, lorsque les données préchargées ne sont pas utilisées par le processus (par exemple un flux séquentiel en lecture qui s'arrête avant la fin d'un fichier), read-ahead a un impact négatif sur les performances. Dans ces cas là le coût en énergie de read-ahead est également augmenté par le nombre de pages Linux lues en trop.

Nous avons modifié le code du noyau pour désactiver read-ahead pour JFFS2 et YAFFS2. Nous avons alors tenté de quantifier l'amélioration de performances apportée par la désactivation de read-ahead grâce à une expérience.

• **Objectif :** observer l'amélioration des performances due à la désactivation de read-ahead.

• **Méthode :** L'expérience consiste à lire deux fichiers de 5 Mo et 10 Mo, de manière séquentielle et aléatoire, dans une boucle de 8 à 2048 appels à *read()*.

• **Résultats :** Les résultats sont présentés sur la figure 3.34.

On peut voir que l'amélioration de performances due à la désactivation de read-ahead est particulièrement importante en séquentiel, et lorsque le nombre de pages lues est faible. En traçant les accès flash via Flashmon, on peut valider le fait que le nombre de lectures de pages flash est effectivement réduit lorsque read-ahead est désactivé. On peut voir une amélioration de performances également conséquente en aléatoire, ce qui signifie que read-ahead est aussi actif lors de ce type d'accès. De plus on peut voir que plus la quantité de données lue est importante, plus l'amélioration des performances et faible : cela est dû au fait que les données préchargées sont effectivement utilisées lorsque la quantité de données demandées par le processus approche la taille du fichier lu.

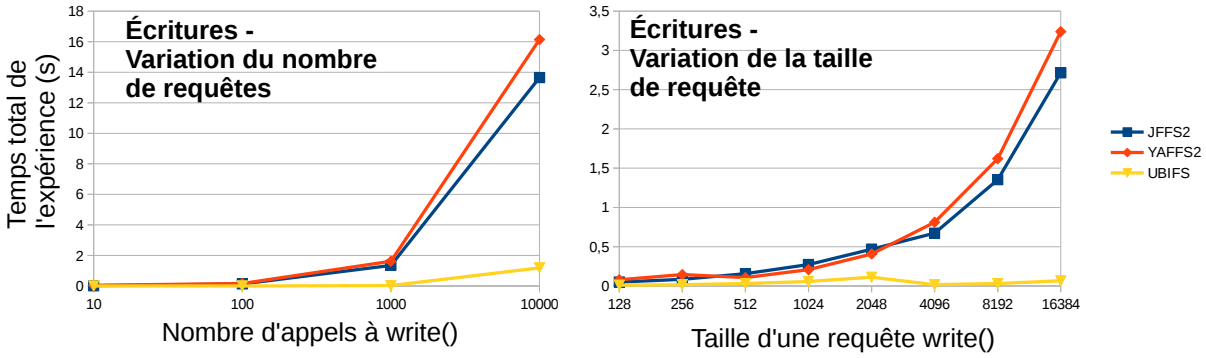


Figure 3.35 – Impact du mécanisme de *write-back* du page cache, activé seulement pour UBIFS. Lorsque l'on fait varier le nombre de requêtes, la taille est fixée à 4 Ko. Lorsque c'est la taille qui varie, le nombre est fixé à 500. Échelle logarithmique en X.

c) Impact du write-back

Le mécanisme de *write-back* dans le page cache n'est pas supporté par JFFS2 (ni par YAFFS2). En effet l'écriture en flash est réalisée pendant l'exécution de la fonction *jffs2_write_end()*. UBIFS, quant à lui, supporte cette fonctionnalité.

• **Objectif :** observer l'impact sur les performances du write-back dans le page cache • **Méthode :** on écrit séquentiellement via *write()* un fichier dans une partition flash (JFFS2, YAFFS2 et UBIFS), en faisant varier le nombre de requêtes d'écriture et leur taille. Lorsque l'on fait varier le nombre de requête, la taille est fixée à 4 Ko. Lorsque c'est la taille qui varie, le nombre est arbitrairement fixé à 500. On mesure le temps d'exécution total de la boucle d'écriture.

• **Résultats :** les résultats sont présentés sur la figure 3.35. On constate que les temps d'exécution pour JFFS2 et YAFFS2 suivent une évolution similaire par rapport à la quantité de données écrites. UBIFS quant à lui présente des temps d'exécution sensiblement plus courts : grâce au support du write-back dans le page cache, les écritures effectives en flash ne sont pas réalisées pendant l'exécution des appels à *write()*. Les données sont mises à jour uniquement dans le page cache (en RAM), ce qui accélère l'opération de manière importante.

Du fait que le write-back n'est pas supporté par JFFS2, qui est le FFS considéré comme cas d'étude dans ce travail de thèse, on ne rentre pas dans les détails concernant son fonctionnement et son impact sur la consommation et les performances.

d) Niveau VFS : conclusion

Le page cache et les mécanismes associés impactent fortement les performances. Concernant la consommation on peut comme au niveau FFS, identifier des valeurs de puissance moyennes, notamment pour ce qui est des accès RAM dus au page cache hits en lecture. Si la puissance lors des accès RAM est supérieure à celle constatée lors des accès flash, l'énergie totale des opérations est principalement impactée par le temps d'exécution de ces opérations.

4 Conclusion

Dans ce chapitre on a présenté les différents éléments impactant les performances et la consommation des systèmes de stockage à base de mémoire flash embarquée gérée par FFS sous Linux. On s'est concentré plus particulièrement sur le contexte de l'écriture et la lecture de données dans des fichiers.

On a premièrement présenté les différents mécanismes fonctionnels traitant la charge d'E/S applicative à tous les niveaux de la pile de gestion du stockage flash : le *système de fichier virtuel*, le *système*

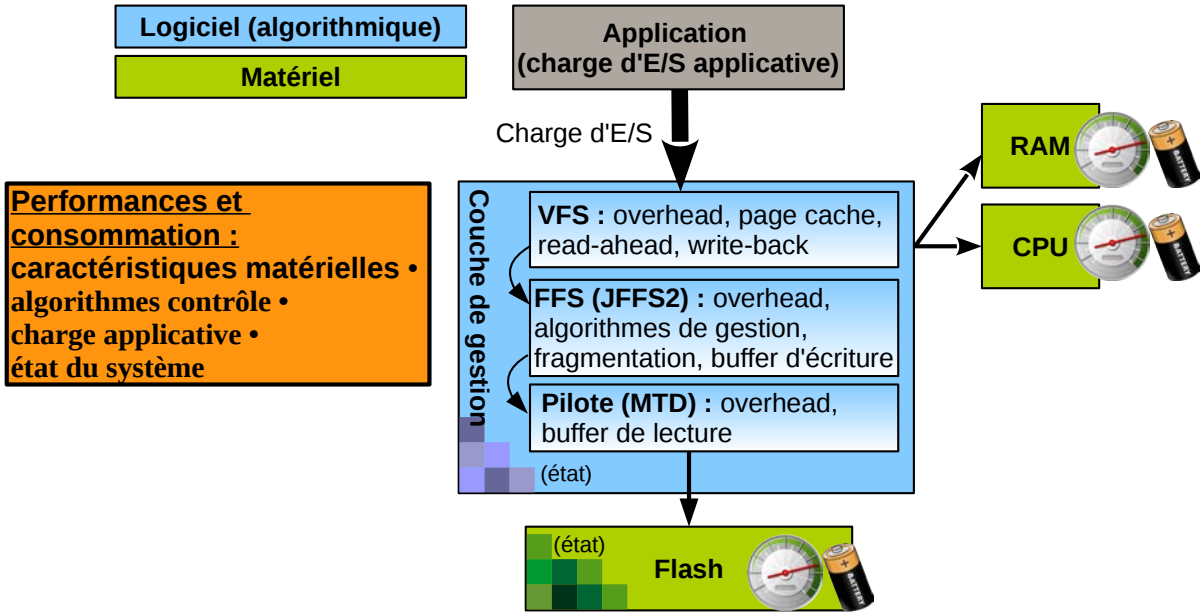


FIGURE 3.36 – Schématisation des différents éléments impactant les performances et la consommation dans un système de stockage à base de mémoire flash.

de fichiers concret (FFS), et le *pilote NAND*. Au niveau de VFS on retient principalement la présence du page cache, qui tamponne les données accédées, et des mécanismes associés : *read-ahead* précharge les données depuis le FFS en fonction de la séquentialité apparente de la charge. Le *write-back*, supporté uniquement par UBIFS, tamponne dans le page cache les opérations d'écritures qui sont par la suite réalisées de manière asynchrone via le FFS. Au niveau du FFS, on s'est intéressé en particulier à JFFS2. JFFS2 empaquette les écritures dans des nodes qui sont écrites séquentiellement en flash. Un *tampon d'écriture* de la taille d'une page flash absorbe les écritures de petites tailles. Divers éléments, en particulier la manière dont un fichier est accédé en écriture, déterminent la *fragmentation* plus ou moins importante des fichiers en flash. Une forte fragmentation peut impacter de manière importante les performances en lecture. Le ramasse-miettes de JFFS2 peut impacter les performances en écriture, notamment lorsque l'espace libre devient faible. Il est important de caractériser la quantité d'espace libre / valide / invalide avant toute expérience de mesure des performances ou de la consommation, car ces valeurs peuvent avoir un impact sur les résultats. Le pilote NAND MTD est utilisé par le FFS pour accéder à la puce de mémoire flash. Il implémente notamment un buffer absorbant les lectures de pages répétées.

Une suite d'outils de trace a été développée pour l'exploration des performances au niveau du noyau Linux. *Flashmon* permet de tracer les opérations effectuées sur la puce flash à bas niveau (pilote). *VFSMon* trace les opérations à haut niveau : VFS, et au niveau de l'interface avec le FFS. Enfin, *FuncMon* permet d'obtenir des temps d'exécution de fonctions noyau relatives à la gestion du stockage, à tous les niveaux de la pile logicielle de gestion. Ces outils sont complémentaires et inter-opérables. Ils seront également utilisés en phase de modélisation et de validation.

Nous avons par la suite montré l'impact effectif des différents éléments précédemment listés sur les performances et la consommation via une série d'expériences. Ces expériences ont étés réalisées sur une plate-forme matérielle qui est la carte *Mistral Omap3evm*. Ces expériences font usage des outils de trace précédemment présentés, et de la plate-forme de mesure de consommation à distance *Open-PEOPLE*. Ces expériences valident le fait que les éléments listés impactent les performances et la consommation.

Pour résumer, on peut généraliser en disant la consommation et les performances d'un système de stockage à base de mémoire sont le résultats des interactions entre plusieurs éléments qui sont :

— Les *caractéristiques des éléments matériels* composant le système de stockage : par exemple les latences des opérations flash de base ou la puissance au repos ou en activité sur les puces CPU, flash et RAM ;
— Les *algorithmes des couches de gestion* qui traitent la charge. Chaque couche (VFS, FFS, MTD) ajoute un overhead en temps et en énergie. Par couche, on peut noter :
  — Au niveau pilote, MTD permet d'effectuer des opérations de base sur la mémoire flash. Un buffer de lecture est présent à ce niveau et peut impacter les performances et la consommation en cas de forte localité temporelle ;
  — Au niveau FFS, on s'est intéressé à JFFS2. On a vu que la manière dont le fichier est réparti en mémoire flash (fragmentation des nodes) peut impacter fortement les performances et la consommation des opérations de lecture. Concernant les écritures, on note la présence d'un buffer tamponnant les écritures de taille inférieure à celle d'une page flash ;
  — Au niveau VFS le page cache tamponne tous les accès et permet de servir des lectures répétées sur les mêmes pages Linux depuis la RAM, ce qui impacte fortement les performances et la consommation en fonction de la charge d'E/S réalisée par l'applicatif. Associés au page cache, *read-ahead* et le *write-back* effectuent respectivement un pré-chargement (synchrone dans le cas des FFS) et des écritures différées depuis le page cache (seulement pour UBIFS).
— Les *caractéristiques de la charge applicative* d'E/S ;
— L'état du système : état des caches, état de la mémoire flash (quantité d'espace libre / valide / invalide), fragmentation d'un fichier JFFS2, etc.

Cela est illustré sur la figure 3.36.

Dans ce contexte les performances et la consommation sont donc influencées par une quantité importante de facteurs. Dans une optique d'estimation, cela prouve la nécessité de recourir à la simulation. Dans ce chapitre on a identifié, dans un système de stockage à base de mémoire flash, l'ensemble des éléments impactant les performances et la consommation. Au chapitre suivant, on présente la manière dont cet impact est caractérisé, puis modélisé.

CHAPITRE 4

Modélisation des performances et de la consommation des systèmes de stockage à base de mémoire flash NAND

Dans le chapitre précédent, on a présenté les différents éléments de la hiérarchie de la couche de gestion du stockage flash embarqué qui impactent les performances et la consommation des systèmes de stockage. Dans ce chapitre, on décrit comment on peut représenter l'impact de ces éléments sous la forme de modèles. On rappelle que les modèles sont construits à des fins d'estimations de performances et de consommation par la simulation.

Dans une première section on définit les différents types de modèles considérés dans ce travail. On retient trois types de modèles principaux : (A) les modèles fonctionnels décrivent l'algorithmique des couches de gestion. Il s'agit d'algorithmes décrivants les traitements fonctionnels appliqués à la charge d'E/S par la couche de gestion, et permettant le maintient de l'état du système de stockage (état des caches, état des pages de la mémoire flash). On utilise également (B) des modèles de performances et (C) de consommation. Il s'agit là de formules permettant de calculer les performances (temps d'exécution) et la consommation (énergie consommée) de différents évènements dont les occurrences et caractéristiques sont calculées par les modèles fonctionnels. On présente comment ces modèles interagissent et se complètent pour obtenir les estimations concernant les performances et la consommation du système de stockage global.

Les modèles réalisés dans le cadre de ce travail de thèse sont présentés en sections 2, 3 et 4. En considérant les différents niveaux de la hiérarchie de stockage vus dans le chapitre précédent (flash et pilote, FFS, VFS), une modélisation multi-niveaux a été réalisée. On adopte une approche de type bottom-up*, présentant les modèles relatifs à un niveau par section, en partant du niveau matériel (flash et pilote) jusqu'au niveau VFS. Pour chaque niveau on présente (A) les différents modèles relatifs à ce niveau, (B) la méthodologie d'extraction de paramètres des modèles de performances et de consommation pour ce niveau et (C) un exemple de profil de performance et de consommation obtenus via extraction de paramètres sur une plate-forme matérielle réelle.*

Finalement, même si le travail présenté ici cible en priorité les systèmes à base de FFS, on souhaite que le simulateur (présenté au chapitre suivant) mette également en œuvre des modèles pour systèmes de type FTL, en particulier des SSD. On présente dans une dernière section une série de modèles qui sont une extension de ceux précédemment présentés. Ces modèles permettent de représenter la structure, les performances, la consommation, ainsi que les opérations applicables à une infrastructure multi-puces flash complexe comme celle que l'on peut retrouver dans les SSD.

Sommaire

1 Modélisation générale d'un système de stockage flash de type FFS

1.1 Notions de modèles

On utilise des modèles pour décrire le système de stockage dont on souhaite estimer les performances et la consommation. Les modèles de types variés sont destinés à interagir, implémentés au sein d'un simulateur. Dans le cadre de ce travail, on définit trois types de modèles principaux : les modèles *fonctionnels*, *de performance*, et *de consommation*. Ils sont utilisés pour décrire respectivement les algorithmes de gestion, et le comportement concernant les performances et la consommation des différents composants matériels du système de stockage de type FFS (flash, CPU et RAM). Pour compléter la description d'un système de stockage, on y ajoute un modèle de *charge* pour décrire la charge d'E/S applicative, et des modèles *structurels* et *opérationnels*, spécifiques à la description du composant matériel flash. Tous ces types de modèles sont présentés dans cette section.

a) Modèles fonctionnels, de performance et de consommation

Modèles fonctionnels Les *modèles fonctionnels* peuvent être décrits sous forme d'algorithmes représentant le comportement de la couche de gestion. Il s'agit d'une simplification des algorithmes de gestion réels. Ils représentent le fonctionnement des mécanismes de ces couches de gestion qui impactent la consommation et les performances de manière significative. Les modèles représentent une description fonctionnelle des couches de gestion flash. Ces modèles prennent en entrée une description de la charge applicative. A partir de cette description, ils permettent de :

1. Calculer les occurrences d'évènements (dont les accès flash, voir plus bas) et paramètres associés qui seront passés aux modèles de performance et de consommation. Ces modèles calculeront pour ces évènements des informations concernant ces métriques. Par exemple un modèle fonctionnel peut calculer le nombre de pages flash dont la lecture est déclenchée par une requête applicative de lecture ;
2. Maintenir l'état du système, en particulier les informations contenues dans les différents caches présents à divers niveaux de la couche de gestion. Le maintient de l'état du système est indispensable à la réalisation du point ci-dessus. Dans le cadre de l'utilisation de JFFS2, comme vu au chapitre précédent, un autre exemple d'information qu'il est essentiel de maintenir est la répartition des fichiers du FFS en *nodes* sur la flash ;
3. Calculer les opérations flash qui sont appliquées à la représentation de la puce flash. La puce flash est elle-même représentée via une série de modèles présentés ci-dessous.

Dans ce chapitre on présente une série de modèles fonctionnels représentant les algorithmes de traitement pour la gestion des fichiers par JFFS2. Des modèles sont réalisés pour tous les niveaux de la couche de gestion : le système de fichiers virtuel et le page cache, le FFS, et le pilote flash. Les modèles fonctionnels sont selon leur définition paramétrables par l'utilisateur. Par exemple, dans le cas des modèles concernant l'utilisation de JFFS2, il est possible de configurer des valeurs telles que la taille d'une page Linux, l'activation ou la désactivation du ramasse-miettes JFFS2 en arrière plan, etc.

JFFS2 a été choisi comme exemple de modèle fonctionnel pour deux raisons principales. Premièrement, il s'agit d'un FFS très utilisé dans le domaine de l'embarqué car il est mature et stable : il est intégré dans le noyau Linux depuis 2001 (Linux version 2.4.10). Il s'agit du système de fichiers conseillé [1]

1. voir `http://processors.wiki.ti.com/index.php/Creating_a_Root_File_System_for_Linux_on_OMAP35x#Configure_the_Linux_Kernel_for_Flash_File_Systems_.28JFFS2_and_CRAMFS.29` [accédé 07/10/2014]

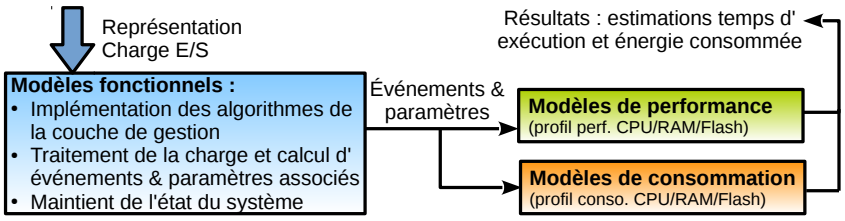


Figure 4.1 – Schéma simplifié des interactions entre les modèles fonctionnels, de performances et de consommation.

lors de l'utilisation de la flash pour stocker le système de fichiers racine sur la carte Omap, plateforme matérielle principale utilisée dans le cadre de cette thèse. Dans un document datant de 2011 (Boschetti, 2011), le constructeur de mémoire flash NAND *Micron* cite JFFS2 comme *un bon compromis entre performances, robustesse, et sur-coût en espace flash* (espace flash utilisé par JFFS2 pour stocker ses propres structures). Le deuxième argument motivant le choix de JFFS2 est la relative simplicité des algorithmes implémentés par ce système de fichiers, par comparaison avec les autres FFS (UBIFS et YAFFS2). Un système de fichiers est un logiciel à la base très complexe. Si des articles scientifiques ou des documents techniques peuvent aider à la compréhension du fonctionnement d'un FFS, une modélisation fonctionnelle efficace passe par l'exploration du code source du FFS concerné. Pour ce qui est de la version de Linux utilisée ici (2.6.37), La taille du code de JFFS2 fait environ 13 000 lignes, contre plus de 30 000 pour YAFFS2, et plus de 20 000 pour UBIFS (sans compter UBI).

Modèles de performance et de consommation Les *modèles de performances* et les *modèles de consommation* représentent le comportement de performances et de consommation, des différents éléments matériels qui entrent en jeu dans la gestion du stockage : la flash, le CPU, et la RAM. Ces modèles sont représentés sous forme de formules qui correspondent aux différents évènements calculés par les modèles fonctionnels (une équation par évènement), comme représenté sur la figure 4.1. Ces équations prennent des paramètres qui sont relatifs à ces évènements, et permettent à partir de ces paramètres d'obtenir des estimations concernant les temps d'exécution et les valeurs énergétiques consommées par les composants lors de ces évènements.

Pour ce qui est de la mémoire flash, des modèles concernant spécifiquement le composant flash lui-même sont présentés, c'est à dire des modèles représentant les temps d'exécution et valeurs de consommation des opérations flash supportées par la puce. Pour ce qui est du CPU et de la RAM, des modèles simples sont réalisés, représentant les temps d'exécution et valeurs de consommation constatés au niveau de ces composants dans le cadre des opérations qui nous intéressent : les accès au stockage flash. Réaliser des modèles de performances et de consommation spécifiques à ces composants matériels est un travail à part entière qui sort du cadre de cette thèse.

L'intégration des modèles de performances et de consommation avec les autres types de modèle, en particulier les modèles fonctionnels, est détaillée dans la section *Modéle général* ci-dessous, traitant des interactions entre les différents types de modèles présentés ici. On y présente également plus en détail le fonctionnement des modèles de performances et de consommation.

b) Modèles de charge et modèles spécifiques au composant flash

Aux types de modèles présentés ci-dessus s'ajoutent des types de modèles concernant la charge d'E/S et le composant flash. Le modèle *de charges d'E/S* permet de représenter une charge applicative appliquée au système de stockage.

On définit également deux types de modèles spécifiques au composant matériel flash : le modèle *structurel* représente les paramètres architecturaux d'une puce de mémoire flash NAND. Le modèle *opérationnel*, quant à lui, représente les opérations qu'il est possible d'appliquer sur la puce, ainsi que

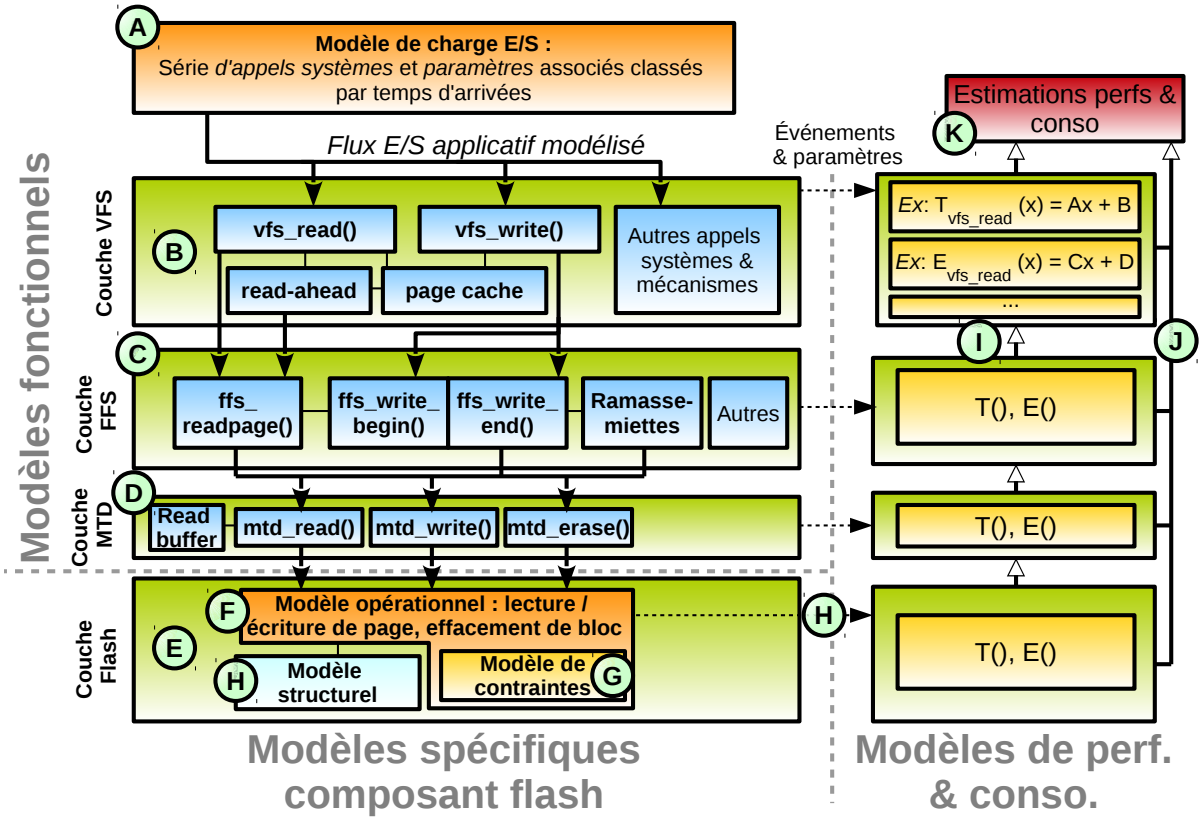


Figure 4.2 – Interactions entre les différents modèles permettant la représentation d'un système de stockage à base de FFS sous Linux embarqué.

la manière dont ces opérations modifient l'état de la puce : par exemple une écriture change l'état d'une page de *libre* à *occupée*. D'une certaine manière, le modèle opérationnel de la puce flash est relativement similaire au modèle fonctionnel de la couche de gestion, dans le sens où il traite une charge en entrée et maintient l'état d'un sous-système. On préfère néanmoins l'appeler *opérationnel* plutôt que *fonctionnel* car il s'applique à un élément matériel, et ne doit pas être confondu avec les modèles représentant les algorithmes de la couche de gestion. Le modèle opérationnel permet également de représenter les contraintes relatives à l'utilisation des opérations supportées par la puce : par exemple le fait qu'il ne soit pas possible d'écrire dans une page non libre. La raison pour laquelle les contraintes flash sont prises en compte dans nos modèles est la suivante : on souhaite que le simulateur développé (présenté au chapitre suivant) propose une infrastructure pour le développement de nouveaux mécanismes de gestion. Ce point est développé au chapitre suivant.

1.2 Modèle général

Dans cette sous-section on récapitule le fonctionnement et les interactions entre tous les types de modèles précédemment présentés. Ces interactions sont illustrées sur la figure 4.2.

Le modèle de charge (A sur la figure 4.2) définit un format pour représenter la charge applicative reçue par le système de stockage. Cette charge est traitée par la couche de gestion, représentée par l'ensemble des modèles fonctionnels. A haut niveau la charge est reçue par une représentation fonctionnelle de la couche VFS (B). Cette couche doit implémenter, entre autres, des modèles reproduisant le comportement d'appels systèmes relatifs au stockage comme par exemple *vfs_read()*. Des mécanismes tels que le page cache ou read-ahead doivent également être intégrés. La couche VFS fait elle-même appel à la couche FFS (C). Les fonctions les plus importantes dans le cas de transfert de données

dans des fichiers sont les fonctions de lecture et d'écritures de pages. Le comportement fonctionnel du ramasse-miettes doit également être modélisé, ainsi que d'autres mécanismes spécifiques au modèle de FFS concerné (par exemple le write buffer pour JFFS2). La couche FFS effectue des accès flash via le pilote NAND (D) : ce dernier comporte des représentations du comportement des fonctions de lecture de page, écriture de page et effacement de bloc. De plus il faut également modéliser le tampon de lecture présent à ce niveau. Enfin, le pilote applique des opérations flash sur la représentation fine du composant matériel "puce flash" (E). Ces opérations sont définies par le modèle opérationnel de la puce (F). Il s'agit d'un modèle "fonctionnel" qui représente la manière avec laquelle ces opérations modifient le contenu de la mémoire flash, et les contraintes associées à ces opérations (G). Le modèle flash structurel (H) définit les paramètres architecturaux de la puce comme par exemple la taille d'une page.

Interactions entre les modèles fonctionnels et les modèles de performances et de consommation A tous les niveaux de la couche de gestion (modèles fonctionnels), le traitement de la charge d'E/S déclenche l'occurrence d'*évènements*, dont il est possible de calculer le temps d'exécution et l'énergie consommée, via les modèles de performances et de consommation (H sur la figure 4.2). Il existe au niveau de ces modèles une formule par évènement calculé et par métrique (temps d'exécution et énergie). La liste des évènements principaux considérés est la suivante :

- Les fonctions de lecture et d'écriture VFS haut niveau *vfs_read()* et *vfs_write()*, ce sont les fonctions noyaux correspondant aux appels systèmes Linux *read()* et *write()* ;
- Les fonctions FFS de lecture et d'écriture de pages, *xxx_readpage()*, *xxx_write_begin()* et *xxx_write_end()*, *xxx* correspondant au modèle de FFS considéré. Au niveau FFS, on considère également la fonction relative à l'exécution d'une passe du ramasse-miettes, pour JFFS2 il s'agit de la fonction noyau *jffs2_garbage_collect_pass()* ;
- Les fonctions du pilote correspondant à la lecture et l'écriture de page, et à l'effacement de bloc ;
- Une lecture, une écriture, un effacement sur le composant flash lui-même.

Dans la section ci-dessous les modèles réalisés sont présentés. On y présente deux formules pour chacun des évènements, permettant de calculer (A) le temps d'exécution de cet évènement et (B) l'énergie consommée par les composants matériels durant cet évènement. Ces formules possèdent des termes qui représentent :

des appels à d'autres formules des modèles de performances / consommation On rappelle que l'on définit les évènements comme étant les appels de fonction dans la pile d'appel de fonctions noyau de la gestion du stockage flash sous Linux. Ainsi, dans une formule correspondant à un évènement haut niveau on peut retrouver des appels à d'autres formules de niveau inférieur. Par exemple, le temps d'exécution d'un appel à *jffs2_readpage()* dépend du nombre de pages flash à lire et donc du temps d'exécution total correspondant à un ou plusieurs appels à *mtd_read()*. La formule pour calculer le temps d'exécution de *jffs2_readpage()* fait donc appel à la formule calculant le temps d'exécution de *mtd_read()*. Cela est également vrai pour l'énergie, et est représenté par la flèche ascendante représentant l'échange d'information entre les modèles de performances / de consommation sur la figure 4.2 (point I). Dans la présentation des formules correspondant aux modèles de performances et de consommation, les termes correspondant à l'appel d'autres formules relatives à ces modèles seront colorés en rouge ;

des paramètres fonctionnels relatifs à l'évènement lui-même Il s'agit d'inconnues dans les formules. Ils sont calculés par la couche fonctionnelle et passés aux modèles de performances et de consommation lors de l'occurrence d'un évènement. Par exemple, comme vu au chapitre précédent, le temps d'exécution (ainsi que l'énergie) d'un appel à *vfs_read()* dépend complètement du fait que les données à lire soient présentes ou non dans le page cache. Ces informations

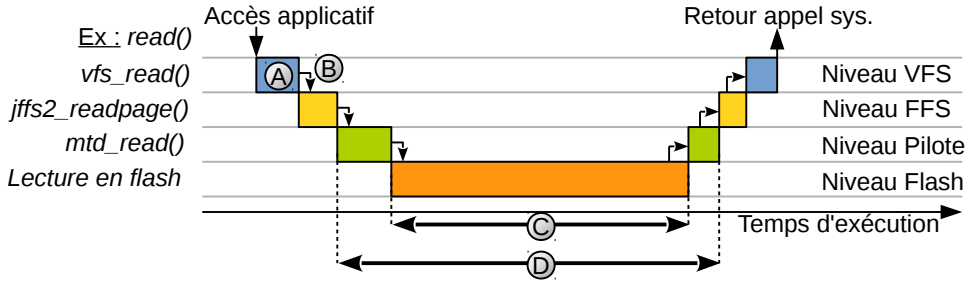


Figure 4.3 – Temps d'exécution de la pile d'appels de fonctions correspondant à la couche de gestion (FFS) et aux accès flash (exemple pour une lecture)

sont maintenues par les modèles fonctionnels et passées aux modèles de performances et de consommation lors de l'occurrence de l'évènement en question. Cela est représenté par le point H sur la figure 4.2. Dans la présentation des formules correspondant aux modèles de performances et de consommation, les termes correspondant à des paramètres calculés par la couche fonctionnelle seront colorés en vert ;

les caractéristiques des composants matériels considérés Il s'agit également d'inconnues dans les formules. Ce sont des valeurs qui indiquent les caractéristiques des composants matériels qui entrent en jeu dans la gestion du stockage (RAM, CPU, flash) concernant les performances et la consommation. L'ensemble de ces valeurs représente un profil de performances pour la plate-forme matérielle considérée. Contrairement aux paramètres fonctionnels qui sont calculés dynamiquement à partir de la charge d'E/S pendant l'utilisation des modèles, les caractéristiques des composants matériels sont fixées pendant l'exécution des modèles. Ils sont destinés à être déterminés par des mesures sur des plate-formes matérielles réelles. L'action d'effectuer ces mesures et de les traiter pour construire un profil de performances / de consommation comme défini par les modèles est appelé *l'extraction de paramètres*. L'extraction de paramètre peut également être réalisée à partir d'informations contenues dans les fiches techniques des composants considérés. Pour la majeure partie des modèles de performances et de consommation présentés dans les sections suivantes, on donne la méthodologie d'extraction de paramètres et on présente un exemple de résultat obtenu sur la plate-forme matérielle correspondant à la carte Omap. Dans la présentation des formules correspondant aux modèles de performances et de consommation, les termes correspondant aux caractéristiques des composants matériels seront colorés en bleu.

On modélise les performances et la consommation en partant du principe suivant : le traitement d'une requête applicative par la couche de gestion correspond à une pile d'appels de fonctions comme illustrée par la figure 4.3. Le temps d'exécution des fonctions haut niveau comprend le temps d'exécution des fonctions de niveau inférieur. Il en est de même pour l'énergie consommée.

Pour construire les modèles de performances et de consommation, on adopte l'approche suivante : on considère l'un après l'autre les différents niveaux formant la pile de gestion du stockage par FFS sous Linux : (A) la puce flash NAND, (B) le pilote MTD, (C) le FFS et (D) VFS. On sait que le traitement de chaque appel système considéré ici (principalement *read()* et *write*) peut être représenté comme une pile d'appels de fonctions. Le sommet de cette pile d'appels est l'appel système lui-même. Au niveaux inférieur on retrouve les différents niveaux de traitement considérés. Par exemple, la figure 4.3 illustre le temps d'exécution du traitement d'un appel système *read()*. Après le lancement de l'appel système, la fonction *vfs_read()* est exécutée (niveau VFS, point A sur la figure 4.3). Cette fonction fait appel à la fonction de lecture de page au niveau FFS (point B). Le niveau FFS fait appel au niveau pilote, qui fait lui-même appel au niveau matériel.

Les paragraphes suivants présentent la méthode de modélisation du temps d'exécution. Une méthode similaire est adoptée pour la modélisation de l'énergie. Le cœur de notre approche se base sur la réflexion suivante : le temps d'exécution à un niveau donné (FFS / VFS / pilote / matériel) est égal à la somme du temps d'exécution des opérations au niveau inférieur et du temps d'exécution supplémentaire apporté par le niveau courant. Par exemple, si l'on prend notre exemple sur la figure 4.3, on peut dire que le temps d'exécution de la lecture au niveau pilote (point D) est égal à la somme entre d'une part le temps de lecture en flash (point C), et d'autre part un coût en temps additionnel au niveau pilote (qui vaut sur notre exemple D moins C). On fait référence au coût additionnel introduit par un niveau comme son *overhead*. Le modèle à un niveau donné est donc de la sorte (exemple pour les performances) :

$$T_{niveau\_courant} = T_{niveau\_inf rieur} + T_{overhead\_niveau\_courant} \tag{4.1}$$

Comme énoncé précédemment on utilise une approche de type *bottom-up*. En partant du plus bas niveau (matériel et pilote), on modélise l'énergie et les performances jusqu'au plus haut niveau (VFS). A un niveau donné, on mesure le temps d'exécution d'une opération, et on détermine l'overhead relatif à ce niveau par soustraction avec le temps ou l'énergie du niveau inférieur. Cela peut se représenter en ré-ordonnant les termes de l'équation 4.1 présentée ci-dessus (exemple pour les performances) :

$$T_{overhead\_niveau\_courant} = T_{niveau\_courant} - T_{niveau\_inf rieur} \tag{4.2}$$

On procède ainsi pour tous les niveaux de la couche de gestion (pilote, FFS, VFS). Bien entendu cette approche implique également de déterminer les temps des opérations flash de base (qui sont le niveau le plus bas considéré). L'objectif de la phase d'extraction de paramètres est donc de déterminer pour une plate-forme donnée (1) les temps et énergies relatifs aux opérations flash de base (lecture, écriture, effacement, point C sur notre exemple de la figure 4.3) et (2) l'overhead en temps et énergie pour chacun des niveaux supérieurs considérés (pilote, FFS, VFS). L'overhead représente donc les temps et énergies relatifs à l'exécution sur les composants CPU et RAM des algorithmes de la couche de gestion.

Cette approche est générique car elle n'est ni spécifique à une plate-forme matérielle particulière, ni relative à un FFS donné. Ainsi, on peut dire qu'elle est applicable à tous les systèmes embarqués intégrant un système de stockage à base de FFS sous Linux.

1.3 Présentation des modèles

Dans les sections suivantes on présente l'intégralité des modèles réalisés dans le cadre de ce travail de thèse. On rappelle que ces modèles ciblent la gestion du stockage flash embarqué dans des systèmes à base de FFS sous Linux. On se concentre sur les accès aux données en lecture et écriture dans les fichiers (appels système *read()* et *write()*), et sur l'utilisation du FFS JFFS2.

On adopte une approche ascendante (*bottom-up*) : on présente tout d'abord les modèles réalisés pour le composant matériel flash et le pilote NAND, puis les modèles correspondant au niveau FFS, puis enfin au niveau VFS. A chaque niveau on présente les modèles fonctionnels, de performances et de consommation réalisés. On donne également à chaque niveau un exemple d'extractions de paramètres, en la réalisation d'un profil de performances / consommation pour une plate-forme matérielle donnée au niveau concerné. Comme expliqué précédemment il existe des modèles spécifiques au niveau flash : ce sont les modèles structurels et opérationnels sont présentés dans la partie concernant le composant matériel flash. Enfin, le modèle de charge est présenté dans la section traitant de VFS car il s'agit de la couche de plus haut niveau, en contact direct avec l'applicatif : le modèle de charge doit donc se conformer aux interfaces de cette couche.

Les interactions entre les différents types de modèles décrits dans ce chapitre ont fait l'objet d'une présentation sous forme de poster (Olivier et coll., 2013b).

2 Modélisation niveau puce flash et pilote

2.1 Flash et pilote : modèles

a) Composant flash - modèle structurel

Le modèle structurel correspond aux caractéristiques architecturales de la puce flash concernée. Il s'agit d'une série de paramètres prenant chacun une valeur. Ces paramètres sont présentés dans la table 4.1.

Paramètre	Description	Unité	Exemple(s)
$N_{pages\_per\_block}$	Nombre de pages flash par bloc	pages	64, 128
N_{blocks_total}	Nombre de blocs flash par puce (ou pour une partition)	blocs	2048
S_{page}	Taille de la partie "données utilisateur" d'une page flash	octets	4096
S_{oob}	Taille de la zone "out-of-band" d'une page flash	octets	32
S_{io}	Largeur du bus d'E/S reliant la puce flash à l'hôte (CPU)	bits	8, 16

Table 4.1 – Paramètres du modèle structurel

Il s'agit d'un modèle de structure relativement simple. La définition de ces paramètres est basée sur les hypothèses suivantes : Dans le cadre d'un système de stockage géré par FFS, le système embarqué ne comporte en général qu'une seule puce flash (certes souvent divisée en plusieurs partitions). De plus, les opérations avancées (*multi-plan*, etc.) ne sont pas supportées par le pilote NAND : on peut alors ne modéliser que les niveaux architecturaux *page* et *bloc* car on sait que le modèle opérationnel ne représentera que les opérations de type *legacy*.

La taille d'une page (zone utilisateur et *OOB*) et la largeur du bus d'E/S sont utilisés notamment par les modèles de performances présentés ci-dessous. Les paramètres, comme la taille d'une page et le nombre de pages par blocs, sont bien sur utilisés de manière intensive par les modèles fonctionnels.

Ainsi, le modèle structurel pour une partition flash de 100 Mo créée sur la puce de la carte Omap (Micron Inc., 2009) est le suivant :

$$N_{pages\_per\_block} = 64; N_{blocks\_total} = 800; S_{page} = 2048; S_{oob} = 64; S_{io} = 8 \tag{4.3}$$

b) Composant flash - modèle opérationnel

Le modèle opérationnel définit les opérations supportées par la puce, la manière dont elles modifient l'état des pages de la puce, et les contraintes associées aux opérations. On rappelle que ce type de modèle peut être vu comme un modèle fonctionnel relatif aux opérations exécutées sur le composant matériel flash. En simulation, les différentes opérations représentées par le modèle opérationnel de la puce flash sont appelées par le modèle fonctionnel correspondant au pilote NAND, présenté en section suivante.

Le modèle opérationnel maintient l'état de chaque page de la mémoire flash. Chaque page est à tout moment dans l'un des deux états suivants : *libre* (prête à écrire) et *occupée* (contenant des données). La page passe de l'état libre à occupé suite à une écriture, et de l'état occupé à libre suite à un effacement du bloc la contenant. Cela est illustré sur la figure 4.4. Du point de vue de la couche de gestion, l'état *occupée* peut se subdiviser en deux sous états : *occupée - contenant des données valides* et *occupée - contenant des données invalides*. Néanmoins, cette distinction n'a pas de sens au niveau de la flash elle-même.

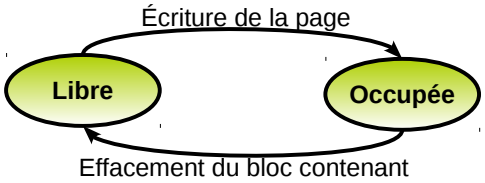


Figure 4.4 – Diagramme d'état d'une page flash modélisée

Comme dit précédemment, dans le cadre d'un système de stockage de type FFS, on applique sur la puce uniquement des opérations de type *legacy* : lecture et écriture de page, effacement de bloc. Un modèle opérationnel est donc présenté, sous la forme d'un algorithme, pour chacune de ces opérations. A noter que des modèles concernant les opérations *avancées* (non legacy) seront également présentés dans ce chapitre, en section 5.

Lecture flash Concernant cette opération le modèle opérationnel est très simple car la lecture ne modifie pas l'état du système, et aucune contrainte n'y est associée. Néanmoins, lorsque les modèles sont exécutés en simulation, il est possible de configurer le système pour qu'un avertissement (*warning*) soit émis lors de la lecture d'une page libre. On rappelle que l'un des rôles du simulateur présenté dans le cadre du travail de thèse est de permettre le développement de nouveaux systèmes de gestion. En simulation la lecture d'une page libre peut être le résultat d'un bug ou d'une mauvaise implémentation de la couche de gestion. A l'inverse dans certains systèmes il est normal de lire une page vide, par exemple JFFS2, sous certaines configurations, lit toutes les pages d'un bloc nouvellement effacé pour la détection des *bad blocks*. L'algorithme 1 représente le modèle opérationnel de l'opération de lecture de page flash.

Algorithme 1 : Modèle opérationnel de l'opération de lecture de page en flash, *legacy read*

```
1 Fonction FlashRead(P)
      entrées  : P indice de la page à lire
      données : P.state, état de P au moment de la lecture, WARN_ON_FREE_PAGE_READ booleen
                indiquant s'il faut ou non émettre un warning en cas de la lecture d'une page libre
2     si P.state == free et WARN_ON_FREE_PAGE_READ alors
3         Afficher un warning;
4     fin
```

A noter que dans les algorithmes représentant les modèles fonctionnels présentés dans ce chapitre, on ne représente pas la mise à jour de statistiques réalisée dans l'implémentation effective des modèles. Par exemple, dans le cas de la lecture de page, un compteur propre à la page est incrémenté à chaque lecture.

Écriture flash Le modèle opérationnel de l'écriture de page flash a deux rôles principaux : premièrement, il permet de vérifier la contrainte posée par la règle *effacer avant d'écrire* : l'écriture dans une page *occupée* est impossible et déclenche une erreur en simulation. Il vérifie également la contrainte d'écriture séquentielle au sein d'un bloc flash. Deuxièmement, ce modèle implémente le changement d'état *libre* vers *occupée* pour une page écrite. Ce modèle est décrit dans l'algorithme 2.

Algorithme 2 : Modèle opérationnel de l'opération d'écriture de page en flash, *legacy write*

```
1 Fonction FlashWrite(P)
      entrées  : P indice de la page à écrire
      données : P.state, état de P au moment de la lecture, P_last adresse de la dernière page écrite dans le bloc
                contenant P
2     si P.state == occupied alors
3         Émettre une erreur;
4     fin
5     si P n'est pas la première page dans le bloc contenant et (P_last + 1) ≠ P alors
6         Émettre un warning si on simule une mémoire SLC, une erreur en MLC;
7     fin
8     P.state = occupied;
```

Effacement flash Ce modèle, décrit dans l'algorithme 3, fait passer l'état de toutes les pages du bloc effacé à *libre*.

Algorithme 3 : Modèle opérationnel de l'opération d'effacement de bloc en flash, *legacy erase*

```
1 Fonction FlashErase(B)
2     pour toute page P ∈ B faire
3         P.state = free;
4     fin
```

c) Composant flash - modèles de performances et de consommation

On peut modéliser au niveau du composant flash le temps des opérations de lecture de page ($T_{FlashRead}$), écriture de page ($T_{FlashWrite}$) et effacement de bloc ($T_{FlashErase}$) à partir du temps d'exécution des sous-opérations suivantes : T_{TON} le temps de transfert des données depuis la matrice NAND vers le page buffer du plan concerné, T_{TIN} l'opération de transfert inverse, T_{IO} le temps de transfert d'une page flash sur le bus (indépendamment du sens) et T_{BERS} le temps d'effacement d'un bloc flash. Des sous-opérations comme les cycles d'envoi de commandes ou adresses sur la puce ne sont pas modélisées car le temps correspondant est négligeable (quelques cycles d'horloge) par rapport aux sous-opérations prises en compte. Les modèles de performances sont présentés ci-dessous. Pour la lecture et l'écriture il s'agit de la somme du temps de transfert sur le bus et de l'opération effective en flash. Pour l'effacement il s'agit uniquement du temps de l'effacement en flash car il n'y a pas de transfert de données dans ce cas.

$$T_{FlashRead} = T_{TON} + T_{IO} \tag{4.4}$$

$$T_{FlashWrite} = T_{IO} + T_{TIN} \tag{4.5}$$

$$T_{FlashErase} = T_{BERS} \tag{4.6}$$

Ces modèles sont une version simplifiée des modèles présentés dans Jung et coll. (2012). Les auteurs de cette étude représentent en détails toutes les sous-opérations mises en œuvre dans l'exécution des commandes flash, cycles de commandes et d'adressage compris.

A noter que T_{IO} peut être modélisé en fonction de la taille d'une page (S_{page} + S_{oob}, en octets), de la fréquence et de la largeur du bus d'E/S (T_{IO_cycle} temps d'un cycle de l'horloge, S_{io} largeur du bus en bits) :

$$T_{IO} = \frac{(S_{page} + S_{oob})}{S_{io}/8} * T_{IO\_cycle} \tag{4.7}$$

T_{TON} et T_{TIN} sont considérés constants dans les modèles concernant les FFS, en effet la plupart des systèmes à base de FFS travaillent sur des puces de type SLC, on ne retrouve donc pas les variations de latences que l'on peut constater sur des puces MLC (Grupp et coll., 2009; Jung et coll., 2012). Concernant

la consommation, l'énergie consommée par la puce flash lors des opérations est la somme des énergies consommées pendant les sous-opérations. Cette énergie consommée pendant les sous-opérations est égale à la multiplication du temps d'exécution de la sous-opération par la puissance constatée sur le rail d'alimentation de la puce lors de l'exécution de la sous-opération (P_{TON}, P_{TIN}, P_{IO}, et P_{BERS}).

$$E_{FlashRead} = T_{TON} * P_{TON} + T_{IO} * P_{IO} \tag{4.8}$$

$$E_{FlashWrite} = T_{IO} * P_{IO} + T_{TIN} * P_{TIN} \tag{4.9}$$

$$E_{FlashErase} = T_{BERS} * P_{BERS} \tag{4.10}$$

Ici $E_{FlashRead}$, $E_{FlashWrite}$ et $E_{FlashErase}$ sont les valeurs d'énergie consommées par la puce flash lors des opérations de lecture / écriture / effacement. Bien que simples, ces modèles de performances et de consommation se situent à un niveau de granularité relativement fin, en comparaison avec des études qui modélisent les temps des opérations flash par une constante simple, par exemple Kim et coll. (2009b). Il est difficile de valider ces modèles par des mesures sans disposer d'un équipement spécifique de mesure fortement précis, pour évaluer les temps d'exécution de sous-opérations au sein d'une puce. Dans notre cas, ces modèles ne sont pas validés via des mesures. Ils sont néanmoins intégrés au simulateur, les différents paramètres de ces modèles pouvant être extraits des fiches techniques de puces flash NAND, ou déterminés de manière empirique.

Dans le cas où l'on dispose d'instruments de mesures suffisamment précis, l'extraction de paramètres des modèles consiste à déterminer par la mesure les valeurs suivantes :

— Les temps d'exécutions des opérations internes à la mémoire flash T_{TON}, T_{TIN} et T_{BERS} ainsi que le temps de transfert sur le bus T_{IO} ;

— Les valeurs de puissances P_{TON}, P_{TIN}, P_{BERS} et P_{IO}, constatées pendant ces opérations.

Le procédé pour déterminer ces valeurs est présenté dans un article (Grupp et coll., 2009) où les auteurs effectuent des campagnes de mesures détaillées sur plusieurs puces flash NAND.

d) Pilote NAND - modèles fonctionnels

Au niveau pilote NAND, on définit trois modèles fonctionnels qui sont des représentations des opérations de lecture, écriture de page, et effacement de blocs. Ces opérations sont appelées en simulation par la couche supérieure, c'est à dire le modèle fonctionnel correspondant au FFS, et font elles-mêmes appel à la couche sous-jacente, le modèle opérationnel flash.

Lecture de page L'algorithme 4 représente le modèle fonctionnel de lecture de page au niveau pilote. A noter la modélisation du buffer de lecture MTD présenté précédemment, qui ne déclenche pas de lecture flash dans le cas d'une page accédée deux fois (ou plus) d'affilée.

Algorithme 4 : Description algorithmique représentant le modèle fonctionnel de la fonction du pilote NAND MTD permettant d'effectuer la lecture d'une page flash, ***MtdRead()***

```
1 Fonction MtdRead(P)
      entrées  : P page flash à lire
      sorties  : Lectures de page en flash
      données  : Buffer de lecture MTD mtdbuf
2     si P ∈ mtdbuf alors
3         Lecture depuis la RAM;
4     sinon
5         FlashRead(P);
6         Placement de P dans mtdbuf (écrase la page précédemment stockée dans mtdbuf);
7     fin
```

Écriture de page Le modèle fonctionnel de l'écriture de page niveau pilote est représenté par l'algorithme 5. L'opération d'écriture est transmise au modèle opérationnel flash. Si la page concernée est

présente dans le buffer de lecture MTD, ce dernier est invalidé (il n'y a pas d'écriture réalisée dans ce buffer).

Algorithme 5 : Description algorithmique représentant le modèle fonctionnel de la fonction du pilote NAND MTD permettant d'effectuer l'écriture d'une page flash, ***MtdWrite()***

```
1 Fonction MtdWrite(P)
      entrées  : P page flash à écrire
      sorties  : ecriture de page en flash
      données : Buffer de lecture MTD mtdbuf
2     si P ∈ mtdbuf alors
3         Invalidation du contenu de mtdbuf ;
4     fin
5     FlashWrite(P);
```

Effacement de bloc Dans le cas de l'effacement de bloc, le rôle du modèle fonctionnel du pilote (algorithme 6) est uniquement de transférer l'opération au modèle opérationnel flash.

Algorithme 6 : Description algorithmique représentant le modèle fonctionnel de la fonction du pilote NAND MTD permettant d'effectuer l'effacement d'un bloc flash, ***MtdErase()***

```
1 Fonction MtdErase(B)
      entrées  : B bloc à effacer
      sorties  : effacement de bloc en flash
2     FlashErase(B);
```

e) Pilote NAND - modèles de performances et de consommation

La construction des modèles de performances et de consommation niveau pilote, ainsi que l'extraction de paramètres associée est un travail qui a fait l'objet d'une publication dans les actes d'une conférence (Olivier et coll., 2013a).

On modélise les temps d'exécutions des opérations de lecture / écriture / effacement flash au niveau pilote comme la somme du temps de l'opération sur la puce elle-même (comprenant le transfert sur le bus) et d'un temps supplémentaire (*overhead*). Cet overhead est dû au pilote lui-même, et représente le temps supplémentaire ajouté par le pilote, correspondant à tout ce qui n'est pas propre à l'opération flash elle-même : allocation mémoire, utilisation de verrous pour éviter les accès concurrents, application d'éventuels codes correcteurs d'erreur, etc. C'est de cette manière que l'impact des performances et de la consommation des composants autres que la flash (CPU et RAM) est intégré dans les modèles, car ces composants impactent les temps d'exécution et la consommation.

Les modèles de performances relatifs au pilote sont présentés ci-dessous :

$$T_{MtdRead}(CacheMiss) = T_{FlashRead} * CacheMiss + T_{MtdRead\_Overhead} \tag{4.11}$$

$$T_{MtdWrite} = T_{FlashWrite} + T_{MtdWrite\_Overhead} \tag{4.12}$$

$$T_{MtdErase} = T_{FlashErase} + T_{MtdErase\_Overhead} \tag{4.13}$$

Dans ces équations, $T_{MtdRead}$, $T_{MtdWrite}$ et $T_{MtdErase}$ sont respectivement les temps d'une lecture de page, écriture de page et d'un effacement de bloc au niveau pilote. On peut noter que concernant la lecture de page, le modèle permettant le calcul du temps d'exécution prend un paramètre nommé $CacheMiss$. Ce paramètre peut prendre deux valeurs, soit 0 soit 1. Il est calculé par le modèle fonctionnel de l'opération de lecture au niveau pilote et prend la valeur 1 lorsque la page à lire n'est pas située dans le buffer de lecture MTD. Dans le cas contraire il vaut 0 et il n'y a pas d'accès flash.

Les modèles de consommation relatifs au pilote fonctionnent sur le même principe que les modèles

de performances. Ils sont les suivants :

$$E_{MtdRead}(CacheMiss) = E_{FlashRead} * CacheMiss + E_{MtdRead\_Overhead} \quad (4.14)$$

$$E_{MtdWrite} = E_{FlashWrite} + E_{MtdWrite\_Overhead} \quad (4.15)$$

$$E_{MtdErase} = E_{FlashErase} + E_{MtdErase\_Overhead} \quad (4.16)$$

2.2 Flash et pilote : méthodologie et exemple d'extraction de paramètres

Au niveau pilote, l'extraction de paramètres consiste à déterminer via des mesures les valeurs correspondantes aux overheads temporels $T_{MtdWrite\_Overhead}$, $T_{MtdRead\_Overhead}$, et $T_{MtdErase\_Overhead}$, et de consommation : $E_{MtdWrite\_Overhead}$, $E_{MtdRead\_Overhead}$, et enfin $E_{MtdErase\_Overhead}$.

On propose pour ce faire d'adopter la méthodologie suivante : par des mesures, on détermine pour une plate-forme matérielle donnée les temps $T_{MtdRead}$, $T_{MtdWrite}$ et $T_{MtdErase}$, et les valeurs d'énergie $E_{MtdRead}$, $E_{MtdWrite}$ et $E_{MtdErase}$. Par la suite, connaissant les valeurs concernant les opérations flash ($T_{FlashRead}$, $E_{FlashRead}$, etc.), on procède par soustraction pour déterminer les valeurs d'overheads comme définies dans équations 4.11 à 4.16. Les latences des opérations flash peuvent être connues soit par la mesure, soit venant des fiches technqiues. Par exemple pour l'overhead d'une opération de lecture de page au niveau pilote on peut réécrire l'équation 4.11 comme suit :

$$T_{MtdRead\_Overhead} = T_{MtdRead} - T_{FlashRead} \quad (4.17)$$

Concernant les temps et valeurs d'énergie pour la mémoire flash elle-même (par exemple $T_{FlashRead}$ ci-dessus), comme énoncé précédemment il faut disposer d'instruments de mesures spécifiques pour les déterminer. Si ce n'est pas le cas, une estimation peut être obtenue à partir de la fiche technique de la puce flash considérée. Dans le cadre de ce travail de thèse c'est cela qui a été effectué. On peut noter que d'après les informations récoltées dans le chapitre concernant l'état de l'art, les données contenues dans les fiches techniques peuvent différer de la réalité (Grupp et coll., 2009), de manière plus ou moins importante. Néanmoins, dans le cadre de notre travail, la majeure partie des estimations calculées par les modèles n'est pas impactées par cette imprécision. En effet les modèles de performances et consommation de niveau supérieur font appel aux termes $T_{MtdRead}$, $E_{MtdRead}$, etc. qui eux sont validés par des mesures.

Les modèles de performances et de consommation niveau flash et pilote sont génériques et utilisable sur différentes plates-formes dans lesquelles Linux gère une puce de mémoire flash. L'action d'utiliser les modèles pour extraire un profil de performances et de consommation pour une plate-forme d'onnée s'appelle l'extraction de paramètres. Le cœur de la méthodologie d'extraction de paramètres pour les modèles de performance et consommation du pilote consiste donc à mesurer les valeurs $T_{MtdRead}$, $T_{MtdWrite}$, $T_{MtdErase}$, $E_{MtdRead}$, $E_{MtdWrite}$ et $E_{MtdErase}$.

a) Extraction de paramètres niveau pilote : performances

On présente ici la méthodologie adoptée pour mesurer les temps de lecture de page flash, écriture de page flash et effacement de bloc flash au niveau pilote : $T_{MtdRead}$, $T_{MtdWrite}$ et $T_{MtdErase}$.

• **Objectif :** extraction par la mesure de $T_{MtdRead}$, $T_{MtdWrite}$ et $T_{MtdErase}$.

• **Méthode :** sur la carte Omap, trois micro-benchmarks sont lancés, chacun ciblant un type d'opération flash. Il s'agit de modules pour le noyau Linux, s'appuyant sur les interfaces noyau offertes par le pilote NAND MTD pour effectuer des accès bas niveau aux pages et blocs physiques de la mémoire flash. Chaque module effectue dans une boucle un nombre important d'accès d'un type donné. Le temps d'exécution total est mesuré par un programme dédié, similaire à la commande Linux *time*, mais utilisant l'appel système *gettimeofday()* pour obtenir une précision de l'ordre de la microseconde. Le nombre d'opérations effectuées par chaque micro-benchmark est varié à partir de 2, par puissance de 2, jusqu'à 5120 (800 pour les effacements dont le nombre est limité par la taille de la partition

de test, 100 Mo). Les opérations sont effectuées en séquentiel et en aléatoire. Le comportement d'un micro-benchmark est présenté dans l'algorithme 7.

Algorithme 7 : Algorithme décrivant le comportement d'un micro-benchmark dédié à l'extraction de paramètres au niveau pilote NAND

entrées : N_{op} nombre d'opérations à effectuer, $type_{op}$ type des opérations à effectuer, $Start_{index}$ indice de page / bloc de départ, $Access\_Pattern$ mode d'accès (Seq ou Ran)

```
Initialisation spécifique au type d'opération à effectuer;
pour i allant de Start_index à Start_index + N_op faire
    si Access_Pattern == Ran alors
        addr = GetRandomIndex();
    sinon
        addr = i;
    fin
    PerformFlashAccess(type_op, addr);
fin
```

La phase d'initialisation est spécifique au type d'opération réalisés : pour la lecture de page, un buffer de la taille d'une page flash est alloué en RAM pour recevoir les données lues. Chaque lecture de page flash écrase le contenu du buffer. Pour ce qui est de l'écriture, un buffer, également de la taille d'une page, est alloué et initialisé avec des données aléatoires, données qui seront écrites dans chaque page flash concernée par le test. Les tests sont réalisés sur une partition flash dédiée, le système de fichier racine (*rootfs*) est situé sur une carte SD et les accès au rootfs n'impactent pas les expériences. Avant chaque test en écriture la partition est effacée. Avant chaque test en lecture la partition est effacée puis écrite avec des données aléatoires.

Le temps d'exécution d'un micro-benchmark peut être représenté comme la somme du temps passé à effectuer des opérations flash. Ce temps dépend du nombre d'opérations réalisées, plus un overhead correspondant aux phases d'initialisation et de terminaison du test. Pour un même type de test (lecture / écriture / effacement), cet overhead reste constant lorsque l'on fait varier le nombre d'opérations réalisées dans la boucle. Le temps d'exécution du micro-benchmark peut donc être modélisé par une équation linéaire : $T_{micro-benchmark}(N_{op}) = \alpha * N_{op} + \beta$. Dans cette équation N_{op} est le nombre d'opérations réalisées, α est le temps d'exécution d'une opération et β correspond à l'overhead. α est la valeur qui nous intéresse : par exemple pour l'opération de lecture, α correspond à $T_{MtdRead}$ dans notre modèle de performances.

On utilise alors une technique de régression linéaire (via le logiciel *R* (R. Core Team, 2012)) sur les couples (*nombre d'opérations réalisées*, *temps d'exécution*) pour déterminer la valeur de α pour chaque type d'opération réalisée au niveau pilote. La figure 4.5 illustre l'évolution du temps d'exécution du micro-benchmark concernant les effacements en séquentiel par rapport au nombre d'effacements réalisés, ainsi que la courbe obtenue via régression linéaire.

Outre les micro-benchmarks présentés ci-dessus, qui se situent au niveau noyau, des micro-benchmarks ont également été réalisés au niveau espace utilisateur de Linux. Ces derniers font usage des outils MTD de l'espace utilisateur pour accéder directement aux adresses physiques de la mémoire flash :

- Le programme **nanddump** permet de lire en mémoire flash. Il peut être lancé avec plusieurs options, pour lire avec / sans utiliser les codes correcteurs d'erreur (ECC), et avec / sans les données OOB ;
- Le programme **nandwrite** permet d'écrire des données sur la mémoire flash, avec / sans ECC et / ou données OOB ;
- Le programme **flash_erase** permet d'effacer des plages de bloc ou l'intégralité de la mémoire flash. Il possède une option qui, en plus d'effacer la mémoire, formate cette dernière pour une utilisation ultérieure avec JFFS2 (écriture de méta-données).

Les programmes MTD de l'espace utilisateur sont conçus pour être utilisés en accès séquentiel, on

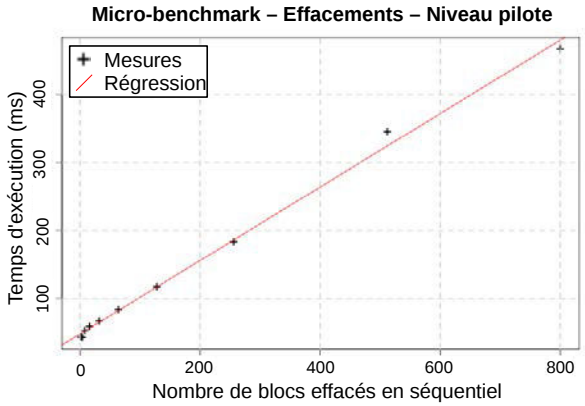


Figure 4.5 – Régression linéaire appliquée aux résultats du micro-benchmark niveau pilote concernant les effacements de blocs

Niveau	Programme / Module	Option	Mode d'accès	α (temps pour 1 accès, µs)[a]	β (µs)	r^2	Valeur-p
MTD noyau	Effacement noyau	-	Seq.	**540.9**	47507.5	0.996	$5.39 * 10^{-11}$
			Aléa.	**531.9**	46157.2	0.999	$1.32 * 10^{-13}$
	Lecture noyau	-	Seq.	**183.0**	41648.5	0.999	$2.79 * 10^{-16}$
			Aléa.	**187.13**	46029.5	0.997	$1.59 * 10^{-14}$
	Ecriture noyau	-	Seq.	**411.5**	48275.8	0.999	$6.15 * 10^{-23}$
			Aléa.	**403.7**	50368.4	0.999	$2.24 * 10^{-17}$
MTD espace utilisateur	Flash_erase	-	Seq.	636.8	43541.7	0.999	$1.92 * 10^{-17}$
	Flash_erase (jffs2 format)	-j	Seq.	988.2	43641.5	0.999	$1.46 * 10^{-18}$
	Nanddump	-	Seq.	225.6	64152.1	0.999	$3.10 * 10^{-31}$
	Nanddump sans ECC	-n	Seq.	198.4	67168.0	0.998	$7.44 * 10^{-16}$
	Nanddump avec OOB	-o	Seq.	302.0	69192.6	0.999	$1.40 * 10^{-28}$
	Nandwrite	-	Seq.	472.7	93579.2	0.977	$1.56 * 10^{-7}$
	Nadwrite sans ECC	-n	Seq.	459.4	58948.5	0.999	$1.65 * 10^{-19}$
	Nandwrite avec OOB	-o	Seq.	871.0	56130.0	0.999	$4.85 * 10^{-33}$

Table 4.2 – Résultats de l'application de la régression linéaire sur les valeurs temporelles extraites par les micro-benchmarks niveau pilote

n'effectue donc pas d'accès aléatoire les concernant. Cette étude au niveau de l'espace utilisateur est réalisée uniquement dans le but d'étudier l'overhead des outils cités ci-dessus (dans le cadre de l'article cité en début de section), et les résultats ne sont pas utilisés dans le processus de modélisation.

• **Résultats :** Les résultats de la régression linéaire appliquée aux sorties des micro-benchmarks sont présentés dans la table 4.2. Comme indiqué dans la table, la régression linéaire donne des résultats très précis. Pour tous les types d'opération, les différences pour α entre accès séquentiels et aléatoires sont relativement faibles (en lecture le nombre d'opérations réalisées est trop faible par rapport à la taille de la partition pour que le buffer de lecture MTD ait un impact). Pour déterminer la valeur des paramètres $T_{MtdRead}$, $T_{MtdWrite}$ et $T_{MtdErase}$ de notre modèle, on choisit donc de prendre la moyenne des valeurs en séquentiel et en aléatoire :

$$T_{MtdRead} = \frac{183.0 + 187.13}{2} = 185.065\mu s \tag{4.18}$$

$$T_{MtdWrite} = \frac{411.5 + 403.7}{2} = 407.6\mu s \tag{4.19}$$

$$T_{MtdErase} = \frac{540.9 + 531.9}{2} = 536.4\mu s \tag{4.20}$$

On constate qu'assez logiquement, l'opération de lecture est plus rapide que l'opération d'écriture, elle-même plus rapide que l'opération d'effacement. Si l'on compare ces chiffres à la fiche technique de la mémoire flash de la carte Omap, on peut estimer l'overhead dû au pilote lors des différentes opérations

aux valeurs suivantes :

$$T_{MtdRead\_Overhead} = 52.4\mu s \tag{4.21}$$

$$T_{MtdWrite\_Overhead} = 105.9\mu s \tag{4.22}$$

$$T_{MtdErase\_Overhead} = 31.9\mu s \tag{4.23}$$

Il s'agit d'un overhead non négligeable qui représente plus d'un quart du temps de réponse total mesuré au niveau du pilote pour les opérations de lecture et d'écriture.

Concernant les programmes de l'espace utilisateur, on constate effectivement des temps de réponse supérieurs, dus aux couches logicielles supplémentaires situées entre l'applicatif et le matériel. Le formatage JFFS2 relatif à *flash_erase* prend un temps particulièrement important car pour chaque effacement de bloc une petite quantité de méta-données est écrite.

b) Extraction de paramètres niveau pilote : consommation énergétique

Dans cette section on présente la méthode de mesure des paramètres du modèle de consommation du pilote, $E_{MtdRead}$, $E_{MtdWrite}$ et $E_{MtdErase}$.

• **Objectif :** Extraction de $E_{MtdRead}$, $E_{MtdWrite}$ et $E_{MtdErase}$.

• **Méthode :** Les micro-benchmarks sont lancés, effectuant cette fois chacun un nombre fixe d'opérations flash : 5120 pour les lectures et écritures, 800 pour les effacements. L'objectif est de mesurer sur les rails d'alimentation des différents composants matériels rentrant en jeu (RAM, CPU, flash) la puissance constatée pendant les 3 types d'accès flash (lecture, écriture, effacement). Cette puissance peut par la suite être multipliée par le temps d'exécution d'un accès (mesuré en section précédente) pour obtenir les valeurs d'énergie qui correspondent aux paramètres du modèle.

On rappelle que sur la carte Omap, notre plate-forme matérielle de test, les mémoires flash et RAM partagent le même rail d'alimentation. On abstrait ces deux composants au sein d'un seul composant nommé *mémoire*. On avait donc précédemment la liste de valeurs de puissances à mesurer initiale :

— $P_{MtdRead\_CPU}$; $P_{MtdWrite\_CPU}$; $P_{MtdErase\_CPU}$;
— $P_{MtdRead\_RAM}$; $P_{MtdWrite\_RAM}$; $P_{MtdErase\_RAM}$;
— $P_{MtdRead\_flash}$; $P_{MtdWrite\_flash}$; $P_{MtdErase\_flash}$.

Cette liste devient :

— $P_{MtdRead\_CPU}$; $P_{MtdWrite\_CPU}$; $P_{MtdErase\_CPU}$;
— $P_{MtdRead\_memoire}$; $P_{MtdWrite\_memoire}$; $P_{MtdErase\_memoire}$.

Lors de l'exécution des micro-benchmarks, on constate des paliers de consommation comme vu au chapitre précédent (voir par exemple la figure 3.16 page 102). Il est ainsi aisé de déterminer une puissance moyenne pour chacun des types d'accès flash, et pour chacun des composants matériels considérés. A ces valeurs de puissance, on soustrait la puissance à vide mesurée sur chacun des composants.

• **Résultats :** Les résultats des mesures sont présentés dans la table 4.3. Les valeurs d'énergie sont obtenues en multipliant la puissance moyenne mesurée pendant l'accès par le temps d'exécution d'un accès unitaire, mesuré dans la section précédente concernant les performances. On peut voir que le type d'opération affecte peu la puissance mesurée : sur le CPU le surcoût de puissance varie entre 180.02 mW (effacement séquentiel) et 192.4 mW (lecture aléatoire). Au niveau de la mémoire, ces valeurs varient entre 16.25 mW (effacement séquentiel) et 11.02 mW (lecture aléatoire). Ainsi, les valeurs d'énergie sont principalement affectées par le temps d'exécution : on retrouve donc une valeur d'énergie plus faible en lecture, car il s'agit d'une opération plus rapide par rapport aux écritures et effacements.

Tout comme pour les performances le mode d'accès n'impacte pas la puissance moyenne. On note néanmoins une légère variation (environ 5%) sur la puissance CPU en lecture. Comme pour les performances, on détermine les valeurs d'énergie pour les paramètres du modèle en prenant

Niveau	Programme / module	Option	Mode d'accès	Puissance mesurée pendant un accès (mW)		Energie pour un accès (µJ)	
				CPU	*Mémoire*	*CPU*	*Mémoire*
MTD noyau	Effacement noyau	-	Séq.	180.08	16.25	**97.40**	**8.79**
			Aléa.	183.54	15.38	**97.62**	**8.18**
	Lecture noyau	-	Séq.	181.39	12.75	**33.19**	**2.33**
			Aléa.	192.4	11.02	**36.00**	**2.06**
	Ecriture noyau	-	Séq.	180.97	15.84	**74.47**	**6.52**
			Aléa.	185.02	15.29	**74.69**	**6.17**
MTD espace utilisateur	Flash_erase	-	Séq.	188.74	14.06	120.19	8.95
	Flash_erase JFFS2	-j	Séq.	187.67	14.29	185.455	14.12
	Nanddump	-	Séq.	188.37	11.07	42.50	2.50
	Nanddump sans ECC	-n	Séq.	182.11	9.12	36.13	1.81
	Nanddump avec OOB	-o	Séq.	200.39	7.60	60.52	2.29
	Nandwrite	-	Séq.	186.72	22.05	88.26	10.42
	Nadwrite sans ECC	-n	Séq.	186.59	22.44	85.71	10.31
	Nandwrite avec OOB	-o	Séq.	189.42	19.08	164.98	16.62

Table 4.3 – Résultats de l'extraction de paramètres concernant l'énergie pour le modèle niveau pilote

pour chaque type d'accès au niveau pilote la moyenne des valeurs d'énergie des accès séquentiels et aléatoires (voir zone grisée sur la table 4.4) :

$$E_{MtdRead} = E_{MtdRead\_CPU} + E_{MtdRead\_mem} \tag{4.24}$$
$$= \frac{33.19 + 36}{2} + \frac{2.33 + 2.06}{2} = 34.6 + 2.2 = 36.8\mu J \tag{4.25}$$
$$E_{MtdWrite} = E_{MtdWrite\_CPU} + E_{MtdWrite\_mem} \tag{4.26}$$
$$= \frac{74.47 + 74.69}{2} + \frac{6.52 + 6.17}{2} = 74.6 + 6.3 = 81\mu J \tag{4.27}$$
$$E_{MtdErase} = E_{MtdErase\_CPU} + E_{MtdErase\_mem} \tag{4.28}$$
$$= \frac{97.40 + 97.62}{2} + \frac{8.79 + 8.18}{2} = 97.5 + 8.5 = 106\mu J \tag{4.29}$$

Contrairement aux performances, on ne présente pas ici d'estimation de l'overhead du pilote concernant l'énergie. En effet, s'il est possible de calculer E_{TON}, E_{TIN} et E_{BERS} d'après les informations contenues dans la fiche technique de la puce flash de la carte Omap, les valeurs de consommation concernant les temps de transferts sur le bus (E_{IO}) ne sont pas incluses dans la fiche technique.

Il est néanmoins possible d'obtenir une estimation de l'overhead du pilote en lecture : on sait que lors d'un cache hit dans le buffer de lecture MTD, il n'y a pas d'accès flash réalisé. L'énergie consommée par un cache hit dans ce buffer est donc égale à l'overhead en lecture pour MTD. Ainsi, on peut se baser sur les résultats de l'expérience réalisée au chapitre précédent (voir figure 3.17 page 103) qui consiste à lire, au niveau pilote, 32768 fois la même page flash. Cela correspond à 32767 cache hits dans le buffer de lecture MTD. On voit que l'opération consomme 0.021 J sur la puce mémoire, et 0.046 J sur la puce CPU. En divisant ces valeurs par le nombre de lectures de pages réalisées (32768), on obtient une estimation de l'overhead en énergie pour la lecture d'une page au niveau pilote :

$$E_{MtdRead\_Overhead\_CPU} = 0.046/32768 * 10^6 = 1.40\mu J \tag{4.30}$$
$$E_{MtdRead\_Overhead\_mem} = 0.021/32768 * 10^6 = 0.64\mu J \tag{4.31}$$

Certes dans ces 32768 accès il y a un cache miss (premier accès) qui présente une consommation supérieure aux autres, mais au vu du nombre important de pages lues son impact sur le résultat final est négligeable. On peut également noter que ces valeurs d'énergies correspondent à la consommation d'un cache hit dans le buffer de lecture MTD.

c) Extraction de paramètres niveau pilote : conclusion

Les valeurs des paramètres des modèles de consommation et de performances, extraits via la méthodologie présentée dans cette section, sont résumées dans la table 4.4.

Performance		Consommation			
Paramètre	**Valeur**	**Paramètre**	**CPU**	**Puce mémoire**	**Total**
$T_{MtdRead}$	185.065 μs	$E_{MtdRead}$	34.6 μJ	2.2 μJ	36.8 μJ
$T_{MtdWrite}$	407.6 μs	$E_{MtdWrite}$	74.6 μJ	6.3 μJ	81 μJ
$T_{MtdErase}$	536.4 μs	$E_{MtdErase}$	97.5 μJ	8.5 μJ	106 μJ

Table 4.4 – Résultats de l'extraction de paramètres au niveau pilote

Ces résultats constituent le profil de performances et de consommation de la carte Omap. Alors que ce profil est spécifique à cette plate-forme matérielle, la méthodologie d'extraction de paramètre mise en œuvre est réutilisable sur d'autres plate-formes, pour construire leur propres profils.

3 Modélisation niveau FFS : Focus sur JFFS2

Dans cette section on présente les modèles relatifs au FFS. A ce niveau, les modèles sont très spécifiques au type de FFS concerné, en particulier les modèles fonctionnels. On prend pour cas d'étude le système de fichiers JFFS2.

3.1 JFFS2 : modèles

a) JFFS2 : Modèles fonctionnels

On définit au niveau FFS quatre modèles fonctionnels : les fonctions de lecture et d'écriture de page Linux (*xxx_readpage()*, *xxx_write_begin()* et *xxx_write_end()*, ainsi qu'un modèle fonctionnel correspondant à l'exécution d'une passe du ramasse-miettes. On présente donc dans cette section un modèle fonctionnel pour les fonctions suivantes : *jffs2_readpage()*, *jffs2_write_begin()*, *jffs2_write_end()* et *jffs2_garbage_collect_pass()*.

jffs2_write_begin() On a vu dans le chapitre précédent que les rôles principaux de *jffs2_write_begin()* sont (A) la lecture de la page concernée par l'écriture si cette dernière n'est pas déjà dans le page cache et (B) la gestion des "trous" de données dans les fichiers (écriture à une adresse supérieure à la taille du fichier). Les "trous" sont un cas particulier qui n'est pas traité dans ce travail de thèse. On considère que les applications qui écrivent à un offset supérieur à la taille d'un fichier sont en faible nombre. On se concentre donc sur le point (A). L'algorithme 8 représente le modèle fonctionnel relatif à cette fonction. Sa tâche est de vérifier si la page concernée est présente dans le page cache, si ce n'est pas le cas elle est lue via la fonction de lecture de page *jffs2_readpage()* qui sera présentée plus loin.

Algorithme 8 : Description algorithmique représentant le modèle fonctionnel de ***jffs2_write_begin()***

```
1 Fonction Jffs2WriteBegin(F, P_current)
      input  : F identifiant de fichier accédé, P_current page touchée par l'écriture
      output : Un éventuel appel à Jffs2ReadPage si la page touchée n'est pas présente dans le page cache
      données : page_cache ensemble des pages présentes dans le page cache
2     si P_current ∉ page_cache alors
3         Jffs2ReadPage (F, P_current)
4     fin
```

jffs2_write_end() Concernant JFFS2, l'écriture effective des données est réalisée par la fonction *jffs2_write_end()*. On a vu au chapitre précédent ses multiples rôles : (A) la réservation en flash de l'espace nécessaire pour écrire les données ; (B) le découpage des données à écrire en fonction de leur

taille et de l'état du bloc courant ; (C) l'écriture via le write buffer et (D) une tentative de défragmentation en ré-écrivant la page Linux complète lorsque les données à écrire contiennent initialement le dernier octet de la page Linux en question. Tous ces rôles sont implémentés dans le modèle fonctionnel de *jffs2_write_end()*, présenté dans l'algoritmhe 9.

Algorithme 9 : Description algorithmique représentant le modèle fonctionnel de ***jffs2_write_end()***

```
1  Fonction Jffs2WriteEnd(F, P_current, offset, count)
       entrées : F identifiant de fichier accédé, P_current page touchée par l'écriture, offset octet dans la page où
                 commence l'écriture, count nombre d'octets à écrire
       sorties : Écritures de node(s) en flash via le write buffer JFFS2. Potentiellement, appel au ramasse-miettes via
                 Jffs2GarbageCollectPass.
2      si offset + count == taille page Linux // dernier octet de la page concernée
3      alors
4          offset = 0 ; count = taille page Linux;
5          // Réécriture page complète
6      fin
7      Réserver un espace en flash de taille count + 2*(taille metadonnees pour une node);
8      si la réservation échoue alors
9          répéter
10             Jffs2GarbageCollectPass() // Ramasse-miettes sous seuil critique
11         jusqu'à réservation réussie;
12     fin
13     Bcur_free = espace libre dans le bloc courant en octets;
14     // Découpage en nodes :
15     si count <= Bcur_free alors
16         Créer une node N_1 contenant les count octets à écrire;
17     sinon
18         Créer N_1 contenant Bcur_free octets;
19         Créer N_2 contenant le reste des données à écrire;
20     fin
21     pour Chaque node N_i précédemment créée faire
22         WB_free = espace libre dans le write buffer en octets;
23         si Taille N_i < WB_free alors
24             Empiler N_i dans le write buffer;
25         sinon
26             Flush du write buffer en flash, écriture via MtdWrite();
27             si Le bloc courant est plein alors
28                 Choisir un nouveau bloc courant dans la liste free;
29             fin
30             si Taille N_i > taille d'une page flash alors
31                 Ecrire N_i directement en flash via MtdWrite();
32             sinon
33                 Empiler N_i dans le write buffer;
34             fin
35         fin
36     fin
```

La tentative de défragmentation (D) est réalisée au niveau des lignes 2 à 4, et peut potentiellement modifier les caractéristiques de l'écriture en cours en la faisant cibler la page Linux complète. A noter que la page Linux est bien dans le page cache car elle a été chargée via *jffs2_write_begin()* : dans le cas où la requête d'écriture initiale ne cible qu'un sous-ensemble de la page, il n'y a pas de donnée à charger depuis la flash lorsque JFFS2 décide de ré-écrire la page complète. La réservation d'espace flash (A) est réalisée aux lignes 5 à 10. A noter l'exécution du ramasse-miettes si l'espace libre est trop faible pour assurer la réservation. Le découpage en nodes correspond aux lignes 11 à 17. L'écriture via

le write buffer correspond quant à elle aux lignes 18 à 33.

Algorithme 10 : Description algorithmique représentant le modèle fonctionnel de *jffs2_readpage()*

```
1 Fonction Jffs2ReadPage(F, P_current)
      entrées : F identifiant de fichier accédé, P_current page lue
      sorties : Lectures flash via le driver, par la fonction MtdRead
2     Déterminer un ensemble de nodes N ∈ F contenant l'ensemble des dernières version des données de P_current;
3     pour N_i ∈ N faire
4         Déterminer la ou les pages flash FlashPages où N_i est inscrite;
5         pour FlashPages_i ∈ FlashPages faire
6             MtdRead(FlashPages_i)
7         fin
8     fin
```

jffs2_readpage( ) Il s'agit de la fonction de lecture de page Linux. Son modèle fonctionnel est représenté dans l'algorithme 10. Son rôle consiste à (A) déterminer les nodes du fichier contenant les données nécessaires pour reconstruire la page Linux demandée (ligne 2) ; et (B) lire ces nodes en mémoire flash (lignes 3 à 8).

JFFS2 : ramasse-miettes Le modèle du ramasse-miettes de JFFS2 est subdivisé en deux fonctions : la première, décrite par l'algorithme 11, représente la fonction principale du ramasse-miettes. La seconde (algorithme 12) correspond au choix du bloc victime.

Concernant la fonction principale (algorithme 11), le ramasse-miettes vérifie si un bloc contenu dans les listes *erase_pending* ou *erase_complete* peut directement être recyclé (ligne 2). Si ce n'est pas le cas un nouveau bloc victime est choisi si nécessaire (lignes 3 à 8, rien n'est fait si aucun bloc victime ne peut être trouvé). Le ramasse-miettes tente alors de déplacer une node valide du bloc victime dans le bloc en cours d'écriture via le write buffer (lignes 9 à 11). Si le bloc victime est complètement obsolète, il est placé dans *erase_pending*, en attente d'effacement lors de la prochaine passe (ligne 12 à 15).

Pour ce qui est du choix du bloc victime (algorithme 12), il est réalisé selon les règles établies par JFFS2, présentées au chapitre précédent (voir page 87). Les listes de blocs *eraseable*, *very_dirty*, *dirty*, et *clean* sont utilisées. Le bloc victime a un certain pourcentage de chances d'être choisi dans la liste des blocs complètement invalides (*eraseable*, ligne 3), contenant beaucoup de données invalides (*very_dirty*, ligne 5), contenant au moins une node valide (*dirty*, ligne 7), ou contenant uniquement des données valides (*clean*, ligne 9). Si cette opération n'aboutit pas, le bloc victime est choisit en fonction de l'état (vide ou pas) des listes *dirty*, *very_dirty*, et *eraseable*.

On a vu précédemment qu'une passe de ramasse-miettes peut être réalisée à différentes occasions : lorsque l'espace libre devient faible, par exemple via *jffs2_write_end( )* comme présenté ci-dessus, mais également de manière relativement régulière par le biais d'un *thread* noyau s'exécutant en arrière plan. La fréquence d'activation du ramasse-miettes est un problème relativement complexe à modéliser. Ce sujet sera abordé dans la section suivante concernant le simulateur. En effet c'est dans le simulateur qu'est implémentée l'infrastructure de gestion du temps et des évènements via laquelle on met en œuvre les appels asynchrones au ramasse-miettes de JFFS2.

b) JFFS2 : modèles de performances et de consommation

Des modèles de performances et de consommation sont proposés pour représenter les temps d'exécution et l'énergie consommée par les fonctions présentées ci-dessus : lecture et écriture de page (*jffs2_readpage( )*, *jffs2_write_begin( )* et *jffs2_write_end( )*), et ramasse-miettes *jffs2_garbage_collect_pass( )*. On utilise au niveau FFS la même technique que pour le niveau pilote : les temps d'exécution et

Algorithme 11 : Description algorithmique du modèle fonctionnel du ramasse-miettes de JFFS2 (fonction principale)

```
1  Fonction Jffs2GarbageCollectPass()
       sorties  : Recyclage d'un bloc ou déplacement d'une node valide hors du bloc victime
       données : B_vict Bloc victime courant, initialisé à null, B_cur bloc courant en cours d'écriture par JFFS2, listes de
                 blocs maintenues par JFFS2 L_erase_pending, L_erase_complete, L_clean
2      si L_erase_pending et L_erase_complete sont vides alors
3          si B_vict == null alors
4              B_vict = ChooseVictimBlock ( );
5              si B_vict == null alors
6                  retourner ;
7              fin
8          fin
9          si B_vict contient au moins une node valide alors
10             déplacer une node valide de B_vict dans B_cur;
11         fin
12         si B_vict ne contient que des nodes obsolètes alors
13             placer B_vict dans L_erase_pending;
14             B_vict = null;
15         fin
16     sinon
17         si L_erase_complete est vide alors
18             Effacer un bloc de L_erase_pending, le déplacer dans L_erase_complete;
19         fin
20         Vérifier un bloc de L_erase_complete, le déplacer dans L_clean;
21     fin
22 retourner ;
```

Algorithme 12 : Description algorithmique du modèle fonctionnel du ramasse-miettes de JFFS2 (fonction de choix du bloc victime)

```
1  Fonction ChooseVictimBlock()
       sorties  : Un bloc sélectionné comme victime
       données : Listes de bloc JFFS2 L_eraseable, L_dirty, L_very_dirty, L_clean
2      DiceRoll = rand ( )  mod 100;
3      si DiceRoll < 40 et L_eraseable est non vide alors
4          retourner le bloc en tête de L_eraseable;
5      sinon si DiceRoll < 86 et L_very_dirty est non vide alors
6          retourner le bloc en tête de L_very_dirty;
7      sinon si DiceRoll < 98 et L_dirty est non vide alors
8          retourner le bloc en tête de L_dirty;
9      sinon si DiceRoll < 98 et L_clean est non vide alors
10         retourner le bloc en tête de L_clean;
11     sinon si L_dirty est non vide alors
12         retourner le bloc en tête de L_dirty;
13     sinon si L_very_dirty est non vide alors
14         retourner le bloc en tête de L_very_dirty;
15     sinon si L_eraseable est non vide alors
16         retourner le bloc en tête de L_eraseable;
17     fin
18     retourner null;
```

énergies consommées sont modélisés comme la somme entre le temps / l'énergie correspondant au niveau inférieur (pilote) plus un overhead correspondant au niveau FFS.

JFFS2 : lecture de page Les modèles de performances et de consommation concernant *jffs2_readpage()* sont représentés par la série d'équations ci-dessous :

$$T_{jffs2\_readpage}(N_{flash\_pages}) = T_{MtdRead} * N_{flash\_pages} + T_{Jffs2\_Readpage\_Overhead} \quad (4.32)$$

$$E_{jffs2\_readpage}(N_{flash\_pages}) = E_{MtdRead} * N_{flash\_pages} + E_{Jffs2\_Readpage\_Overhead} \quad (4.33)$$

Dans ces équations N_{flash_pages} représente le nombre de pages flash à lire pour récupérer les données concernant la page Linux demandée. Il s'agit d'un paramètre calculé par les modèles fonctionnels qui maintiennent l'état de la répartition du fichier concerné en nodes sur la flash. $T_{MtdRead}$ et $E_{MtdRead}$ sont des appels au modèle de performances / consommation de niveau inférieur (pilote). A noter que le fait qu'une lecture de page flash puisse déclencher un *cache hit* dans le buffer de lecture MTD n'est pas représenté ici pour des raisons de lisibilité, mais est calculé par le modèle fonctionnel à l'appel de $T_{MtdRead}$ et $E_{MtdRead}$, en fonction de l'indice de page lue et de l'état du buffer. Pour chaque équation l'overhead représente le sur-coût en temps et énergie ajouté par l'exécution de la fonction *jffs2_readpage()* en plus des temps / énergies dus aux accès pilote.

JFFS2 : écriture de page Les modèles concernant *jffs2_write_begin()* sont les suivants :

$$\begin{aligned} T_{jffs2\_write\_begin}(PageNotInPageCache) &= PageNotInPageCache * T_{jffs2\_readpage} \\ &\quad + T_{Jffs2\_WriteBegin\_Overhead} \end{aligned} \quad (4.34)$$

$$\begin{aligned} E_{jffs2\_write\_begin}(PageNotInPageCache) &= PageNotInPageCache * E_{jffs2\_readpage} \\ &\quad + E_{Jffs2\_WriteBegin\_Overhead} \end{aligned} \quad (4.35)$$

Dans les équations concernant *jffs2_write_begin()*, le paramètre $PageInPageCache$ est calculé par le modèle fonctionnel. Il vaut 1 lorsque la page subissant l'écriture n'est pas présente dans le page cache, 0 sinon.

Pour ce qui est de *jffs2_write_end()*, on utilise les équations ci-dessous :

$$\begin{aligned} T_{jffs2\_write\_end}(Gc, N_{flash\_pages}) &= N_{flash\_pages} * T_{MtdWrite} \\ &\quad + Gc * T_{Jffs2GarbageCollectPass} \\ &\quad + T_{Jffs2\_WriteEnd\_Overhead} \end{aligned} \quad (4.36)$$

$$\begin{aligned} E_{jffs2\_write\_end}(Gc, N_{flash\_pages}) &= N_{flash\_pages} * E_{MtdWrite} \\ &\quad + Gc * E_{Jffs2GarbageCollectPass} \\ &\quad + E_{Jffs2\_WriteEnd\_Overhead} \end{aligned} \quad (4.37)$$

Dans ces équations N_{flash_pages} et Gc sont calculés par le modèle fonctionnel de JFFS2. La première valeur représente le nombre de pages flash écrites via le pilote par *jffs2_write_end()*. Cette valeur peut éventuellement valoir 0 dans le cas où une écriture de petite taille est absorbée dans le write buffer de JFFS2. Le terme Gc représente un éventuel nombre de passes du ramasse-miettes exécutées pour réserver l'espace flash nécessaire à l'écriture. On peut noter que sur une flash avec une quantité d'espace libre non critique, cette valeur vaut généralement 0.

JFFS2 : ramasse-miettes Le temps pris par l'exécution d'une passe du ramasse-miettes, ainsi que l'énergie consommée par cette exécution dépendent (A) du nombre d'accès flash effectués pendant la passe et (B) d'un overhead représentant le reste du temps / de l'énergie, indépendant des accès flash :

$$\begin{aligned} T_{Jffs2GarbageCollectPass}(N_r, N_w, N_e) &= N_r * T_{MtdRead} + N_w * T_{MtdWrite} \\ &\quad + N_e * T_{MtdErase} + T_{Jffs2\_Gc\_Overhead} \end{aligned} \quad (4.38)$$

$$\begin{aligned} E_{Jffs2GarbageCollectPass}(N_r, N_w, N_e) &= N_r * E_{MtdRead} + N_w * E_{MtdWrite} \\ &\quad + N_e * E_{MtdErase} + E_{Jffs2\_Gc\_Overhead} \end{aligned} \quad (4.39)$$

Dans ces équations, N_r, N_w et N_e sont respectivement le nombre de lectures, d'écritures de pages et d'effacements de blocs flash réalisés via le pilote durant la passe du ramasse-miettes. Concernant les overheads, il faut noter que si pour des raisons de simplicité ils sont représentés par une valeur constante ($T_{Jffs2\_Gc\_Overhead}$ et $E_{Jffs2\_Gc\_Overhead}$), dans la réalité ils peuvent être très variables suivant l'état du système : on peut voir dans l'algorithme 11 que le comportement du ramasse-miettes présente de nombreuses instructions conditionnelles qui font qu'il effectue un travail qui varie selon l'état du système. Dans le cas spécifique de JFFS2, une modélisation précise du ramasse-miettes consisterait à subdiviser et modéliser les cas d'exécutions principaux suivants :

- Lorsque la liste *erase_complete* est non vide en début de passe :
  - de manière générale le ramasse-miettes déplace le bloc dans la liste *free* et retourne directement, on peut penser que l'overhead est quasiment négligeable ;
  - Lorsque *erase_pending* est non vide, un bloc est effacé. Il est par la suite intégralement lu pour vérifier qu'il ne s'agit pas d'un *bad block*. Le temps d'exécution de la passe est donc relativement long dans ce cas (géré par notre modèle car il prend en compte les accès flash). L'overhead est quant à lui certainement différent des autres cas ;
- Lorsque *erase_complete* et *erase_pending* sont vides en début de passe, le ramasse-miettes itère sur le bloc victime jusqu'à trouver une node valide (overhead variable selon l'état du bloc victime). La node est alors copiée dans le bloc courant (lecture(s) puis écriture(s) de pages flash).

3.2 JFFS2 : méthodologie et exemple d'extraction de paramètres

L'extraction de paramètres consiste à calculer les valeurs d'overheads pour les différents modèles de performances et de consommation présentés ci-dessus. On propose la méthodologie suivante : il s'agit de mesurer les temps d'exécution et valeurs d'énergie consommée pour chacune des fonctions présentées ci-dessus (les fonctions de lecture et écriture de pages, et la fonction correspondante au ramasse-miettes). Ce faisant, on utilise les outils de traces présentés au chapitre précédent pour mesurer le nombre d'accès flash effectués pendant chaque appel aux fonctions concernées. On peut alors évaluer les temps et valeurs d'énergies relatives aux accès flash en utilisant le modèle niveau pilote. Ces valeurs relatives aux accès flash sont alors soustraites des temps d'exécution / valeurs d'énergies totales des fonctions JFFS2 pour obtenir la valeur d'overhead relative à chaque fonction.

Dans cette section on présente la méthodologie d'extraction de paramètres pour les modèles de performances appliquée à la fonction *jffs2_write_end()*. Une méthodologie similaire a été appliquée pour l'extraction de paramètres des modèles de performances de *jffs2_write_begin()* et *jffs2_readpage()*.

Concernant l'extraction de paramètres des modèles de consommation, elle n'a pas été réalisée au niveau FFS. Cela est dû aux raisons suivantes : comme expliqué précédemment, les instruments de mesures de consommation utilisés opèrent à une granularité assez large (voir le chapitre précédent page 98). Pour cette raison, lorsque l'on souhaite mesurer la puissance constatée pendant une opération dont le temps d'exécution est très court, il est nécessaire de lancer de nombreuses fois cette opération dans une boucle, et de faire une moyenne. C'est par exemple ce qui a été fait lors de la mesure de la puissance moyenne constatée pendant les accès flash, au niveau pilote. Or, il n'est pas possible de lancer dans une boucle de manière rapprochées les fonctions JFFS2 : en effet *JFFS2 n'exporte pas d'interface dans le noyau.* Cela signifie que les deux seuls moyens pour exécuter une fonction JFFS2 dans le noyau sont les suivants : (A) depuis le code de JFFS2 lui-même et (B) depuis l'interface standard entre VFS et JFFS2. Il n'est pas possible de créer un module Linux qui lance en boucle une fonction JFFS2 comme cela a été réalisé pour le pilote MTD, car les fonctions JFFS2 ne sont pas accessibles depuis un module. Utiliser l'interface standard de VFS revient à effectuer des requêtes *read()* et *write()* depuis une application. On mesure alors, en soustrayant la consommation au niveau pilote connue via les modèles précédemment présentés, l'overhead du FFS et de VFS. C'est ce qui a été réalisé dans le cadre du travail présenté ici. Les résultats seront décrits en section suivante traitant de VFS.

A noter que ce problème se pose uniquement pour la consommation. Concernant les performances, les outils de trace développés permettent de mesurer le temps d'exécution de fonctions à une granularité fine.

a) Extraction de paramètres JFFS2 : *jffs2_write_end()*

Le but de l'extraction de paramètre concernant *jffs2_write_end()* est de déterminer la valeur de $T_{Jffs2\_WriteEnd\_Overhead}$ présenté plus haut dans l'équation 4.40.

• **Objectif :** extraction de $T_{Jffs2\_WriteEnd\_Overhead}$.

• **Méthode :** on utilise un micro-benchmark qui effectue des accès en écriture séquentielle sur un fichier créé en début de benchmark sur une partition JFFS2 effacée. Les écritures sont réalisées via *write()* dans une boucle. Le micro-benchmark est lancé six fois, à chaque lancement on définit la taille de chaque écriture à 512, 1024, 1536, 2048, 3072 et 4096 octets. Il est inutile de dépasser la taille d'une page Linux (4096 octets) car les requêtes *write()* sont divisées en pages Linux avant appels par VFS à *jffs2_write_end()*. Pour tous les lancements le nombre de requêtes est fixé à 10 000. Lors du lancement des benchmarks, le temps d'exécution de chaque appel à *jffs2_write_end()* est tracé avec FuncMon. Avec Flashmon on trace également le nombre d'écritures flash relatif à chaque appel à *jffs2_write_end()*.

On obtient donc une liste de couples $(temps\_execution_i,\ nb\_pages\_flash\_ecrites_i)$, chacun correspondant à un appel à *jffs2_write_end()*. La liste intègre les résultats de tous les lancements de micro-benchmarks. $T_{Jffs2\_WriteEnd\_Overhead}$ est alors calculé de manière suivante :

$$T_{Jffs2\_WriteEnd\_Overhead} = \frac{\sum_{i=1}^{n}[temps\_execution_i - (nb\_pages\_flash\_ecrites_i * T_{MtdWrite})]}{n} \tag{4.40}$$

Dans cette équation n est le nombre total d'appels à *jffs2_write_end()* tracés pour l'intégralité des lancements de micro-benchmarks. Le résultat pour la carte Omap est le suivant :

$$T_{Jffs2\_WriteEnd\_Overhead} = 54.6\mu s \tag{4.41}$$

L'écart-type concernant cette moyenne est de $17.4\ \mu s$. Il faut noter que cette moyenne générale est une représentation à une granularité large de l'overhead de *jffs2_write_end()*. Dans les faits, cet overhead est impacté par plusieurs facteurs, notamment la taille et l'alignement des requêtes *write()* (en fonction des pages Linux composant le fichier), ainsi que la présence du write buffer JFFS2. La figure 4.6 présente de manière détaillée les résultats de l'exécution des micro-benchmarks.

• **Résultats :** sur la figure 4.6, on retrouve en abscisse les différentes tailles de requêtes *write()* effectuées par les micro-benchmarks. En ordonnées, on retrouve les moyennes des séries d'overheads calculées par différence avec la multiplication de $T_{MtdWrite}$ par le nombre de pages lues pour chaque appel à *jffs2_write_end()* (voir équation 4.40). En prenant par exemple le point situé à (1024, ~30), le graphique se lit comme suit : *dans le cadre du micro-benchmark fixant la taille de* write() *à 1024 octets, les appels à* jffs2_write_end() *déclenchant la lecture d'une page présentent un overhead évalué moyen de 30 μs.*

La figure 4.6 montre que l'overhead varie en fonction de la taille de données à écrire (au niveau de *jffs2_write_end()*) et d'autres facteurs internes à JFFS2. Ces facteurs influent également sur le nombre de pages flash écrites, ce qui explique les écarts entre les courbes de couleurs différentes sur la figure 4.6. Parmi ces facteurs on retrouve notamment l'état du write buffer et l'espace libre restant dans le bloc courant.

L'impact de ces facteurs sur l'overhead n'est pas modélisé car de manière générale, même s'il varie, l'overhead reste d'une valeur très faible par rapport au temps dû aux accès flash : il a donc un impact très faible sur le temps d'exécution total de *jffs2_write_end()*. Il y a une différence d'un facteur égal à presque 10 entre l'overhead moyen (54.6 μs) et le temps d'*une unique* écriture flash niveau pilote (407.6

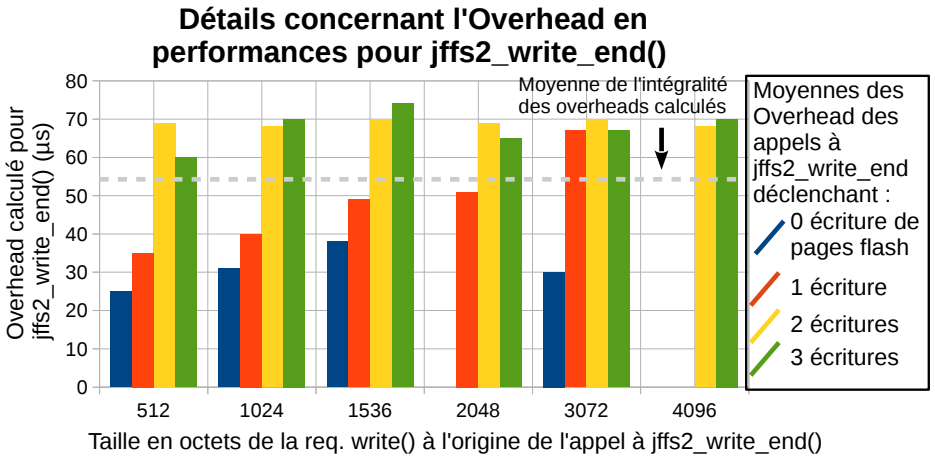


Figure 4.6 – Résultats détaillés des micro-benchmarks concernant le calcul de l'overhead de *jffs2_write_end()*

μs). On choisit donc d'utiliser une valeur constante dans nos modèles, 54.6 μs pour la carte Omap, représentée par le trait gris pointillé sur la figure 4.6.

De plus, on peut noter, concernant la figure 4.6, les observations suivantes :

— Les micro-benchmarks dont la taille d'écriture (*write()*) est fixée à 2048 et 4096 octets déclenchent toujours au moins une écriture de page flash. Cela est normal car :
  — Ces tailles sont supérieures (ou égales) à la taille d'une page flash, c'est à dire la taille du write buffer JFFS2 : ce dernier n'absorbe donc jamais ces écritures, il y a au moins une écriture de page flash (au moins 2 pour une taille de 4096 octets) ;
— Concernant le micro-benchmark de taille 3072 octets, on constate des écritures de 0 page flash, ce qui semble contredir le point ci-dessus. Pour expliquer cela, on rappelle que le micro-benchmark effectue des écritures en séquentiel, et que VFS découpe les appels à *write()* en potentiellement plusieurs appels à *jffs2_write_end()* si la requête *write()* chevauche une ou plusieurs frontières de pages Linux. C'est le cas pour le micro-benchmark de taille 3072 octets. Prenons par exemple les deux premières requêtes *write()* générées par ce test. La première est de taille 3072 octets à l'adresse 0 dans le fichier (page Linux d'indice 0). Un unique appel à *jffs2_write_end()* est effectué. La seconde est de même taille, ciblant l'adresse 3072 : elle chevauche les pages Linux d'indice 0 et 1. Elle est découpée par VFS en deux appels à *jffs2_write_end()*, l'un à l'adresse 3072 (page Linux 0 dans le fichier) pour $4096 - 3072 = 1024$ octets, l'autre à l'adresse 4096 (page Linux 1) pour $3072 - 1024 = 2048$ octets. L'un de ces appels est de taille inférieure à celle du write buffer JFFS2 et peut donc potentiellement ne déclencher aucune écriture de page flash, en fonction de l'état du write buffer.

Sur la gauche de la figure 4.7 on présente le temps d'exécution des 100 premiers appels à *jffs2_write_end()* déclenchés par le micro-benchmark pour lequel on fixe la taille d'écriture à 3072 octets. On peut premièrement noter les différents paliers de temps d'exécution qui correspondent à différents nombres de pages flash écrites pendant l'exécution de la fonction d'écriture JFFS2. Ces paliers sont séparés par une valeur temporelle égale à $T_{MtdRead}$ (407.6 μs). La valeur de $T_{Jffs2\_Write\_Begin\_Overhead}$ peut être observée au niveau des appels ne déclenchant pas d'écriture de page, pour lesquelles les données à écrire sont absorbées par le write buffer JFFS2. On voit que cette valeur est très faible par rapport aux appels déclenchant des accès flash. Sur la droite de la figure 4.7 on peut voir le nombre d'appels à *jffs2_write_end()* déclenchant l'écriture de 0, 1, 2 et 3 pages flash (en considérant le nombre total d'appels à cette fonction réalisés pendant le micro-benchmark).

Une étude similaire à ce qui est fait pour *jffs2_write_end()* a été menée concernant la fonction *jffs2_readpage()*, en traçant via Funcmon son temps d'exécution et avec Flashmon le nombre de pages

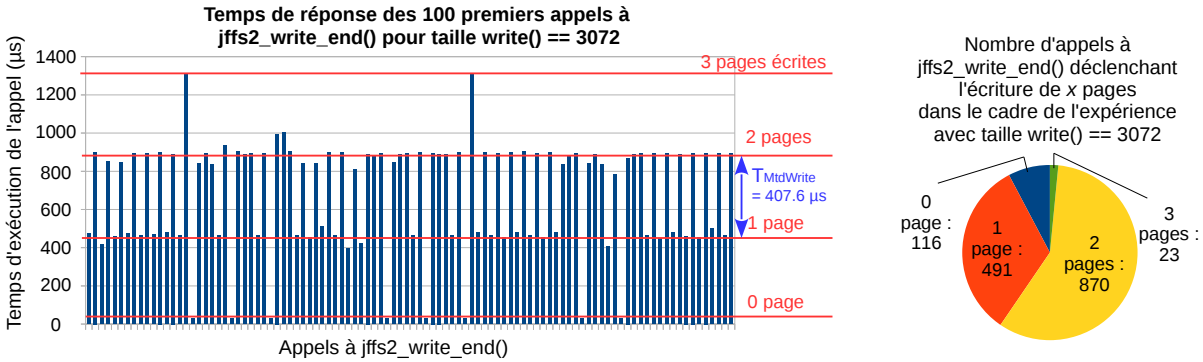


Figure 4.7 – A gauche, le graphique présente le temps d'exécution (tracé avec FuncMon) des 100 premiers appels à *jffs2_write_end()* dans le cadre de l'expérience dont la taille est fixée à 3072 octets. A droite, concernant la totalité des appels à la fonction jffs2, il s'agit du nombre d'appels déclenchant l'écriture de 0, 1, 2 et 3 pages.

lues. Concernant *jffs2_write_begin()*, en faisant en sorte que les micro-benchmarks touchent des pages qui sont déjà dans le page cache ou qui n'existent pas, on s'assure que *jffs2_write_begin()* ne déclenche pas d'accès flash (lecture de la page concernée par l'écriture). L'overhead est donc simplement égal au temps d'exécution de *jffs2_write_begin()*.

Concernant la consommation, comme énoncé précédemment elle sera modélisée avec celle de VFS en section suivante.

b) Extraction de paramètres JFFS2 : conclusion

Les résultats sont les suivants :

$$T_{Jffs2\_ReadPage\_Overhead} = 46.8\mu s \text{ (écart-type : } 14.8\mu s) \tag{4.42}$$

$$T_{Jffs2\_Write\_Begin\_Overhead} = 5.7\mu s \text{ (écart-type : } 0.7\mu s) \tag{4.43}$$

$$T_{Jffs2\_WriteEnd\_Overhead} = 54.6\mu s \text{ (écart-type : } 17.4\mu s) \tag{4.44}$$

On constate que ces overheads sont tous très faibles par rapport aux temps des accès flash, ce qui justifie le fait de les modéliser par une constante. L'extraction de paramètre confirme l'observation réalisée au chapitre précédent : l'élément principal impactant les performances (et par conséquent l'énergie) est le nombre d'accès flash réalisés dans les fonctions relatives au FFS.

3.3 JFFS2 : autres fonctions modélisées

Outre les fonctions de lecture / écriture de pages et du ramasse-miettes d'autres fonctions relatives à JFFS2 ont été modélisées. Il s'agit des fonctions JFFS2 appelées par VFS lors de :

- la création (appel système *creat()*) et la suppression (*remove()*) d'un fichier ;
- l'ouverture (*open()*) et la fermeture (*close()* d'un fichier ;
- la troncature (*truncate()*) d'un fichier ;
- le re-nommage (*rename()* d'un fichier ;
- la synchronisation d'un système de fichiers (*sync()* avec le média de stockage concerné.

Concernant ces fonctions, uniquement les performances et la consommation dues aux accès flash ont été modélisées. Certaines sont surtout modélisées pour assurer le bon fonctionnement du simulateur lors de l'exécution des modèles (par exemple une lecture sur un fichier non créé doit déclencher une erreur, indiquant un problème d'implémentation ou dans la représentation de la charge d'E/S). Les équivalents JFFS2 de *creat()*, *remove()*, *truncate()* et *rename()* provoquent la création d'une node de méta-données de petite taille représentant la modification apportée au système de fichiers, et son écriture via le write-buffer. Ce comportement est inclus dans le modèle fonctionnel de JFFS2.

4 Modélisation niveau VFS

Dans cette section on présente les modèles fonctionnels, de performance et de consommation relatifs au niveau VFS. On présente également ici le modèle de charge d'E/S applicative. Cela est dû au fait que VFS est le premier maillon de la chaîne de traitement des E/S provenant de l'applicatif.

4.1 VFS : modèles

a) VFS : Modèles fonctionnels

On présente ici les modèles fonctionnels relatifs à la couche VFS. Il s'agit des modèles concernant les fonctions *vfs_read()* et *vfs_write()*. Comme dit précédemment, ce sont les fonctions appelées par les appels systèmes Linux *read()* et *write()*. On propose également un modèle simple de représentation du page cache. De plus, on propose un modèle représentant en détail le comportement de l'algorithme *read-ahead*, qui est activé par défaut pour JFFS2. Ce FFS n'utilise pas le mécanisme de *write-back*, donc ce dernier n'est pas modélisé. Concernant les performances et la consommation du page cache et de read-ahead, ces derniers sont inclus dans les modèles de performances et consommation des fonctions *vfs_read()* et *vfs_write()*.

Modèle fonctionnel VFS : *page cache* Le page cache est une structure très complexe au sein de Linux (Bovet et Cesati, 2005), qui dans les systèmes actuels peut potentiellement utiliser la majorité de la mémoire vive disponible (Rao et coll., 2005). Le page cache contient en RAM des données provenant de multiples sources :

- Des données appartenant à des fichiers en stockage secondaire, accédées en lecture et écriture (c'est le cas qui nous intéresse dans le cadre de ce travail de thèse) ;
- Des données appartenant à des fichiers mappés en mémoire (via l'appel système *mmap()*) ;
- Des données tamponnées lors des accès directs aux périphériques de type bloc. Précédemment ces données tamponnées étaient stockées dans ce que l'on appelait le *buffer cache*, qui a été fusionné avec le page cache dans la version 2.4 de Linux (Riel, 2001) ;
- D'autres types de données comme par exemple des pages appartenant à l'espace d'adressage de processus qui ont été placées en *swap* (fichier d'échange) mais dont une partie doit rester en mémoire vive.

Dans ce travail de thèse on ne s'intéresse qu'au premier point. On modélise le page cache comme une liste de couples $(F_i, Index_i)$. Chaque couple représente une page présente dans le page cache. F_i est un identifiant de fichier auquel appartient cette page. On peut le voir comme un numéro d'*inode*, identifiant unique d'un fichier dans le système d'exploitation, pour une partition donnée au niveau VFS. $Index_i$ représente l'indice de la page dans le fichier concerné.

En réalité, sous Linux les pages contenues dans le page cache sont structurées et indexées de manière complexe. L'utilisation d'un type d'arbre nommé le *radix tree* (Bovet et Cesati, 2005; Rao et coll., 2005) permet de chercher, insérer et supprimer des pages dans le page cache de manière très rapide. De plus, l'évolution du temps d'exécution de ces opérations par rapport au nombre d'objets contenu dans le page cache (passage à l'échelle) est très efficace. Par exemple, Rao et coll. (2005) reportent des temps de recherche et insertion d'environ une seconde dans un page cache contenant 100 millions d'éléments.

Lorsque la quantité de mémoire vive disponible devient faible, la taille du page cache doit être réduite. Pour les pages appartenant à des fichiers, lorsque l'une d'elles est sélectionnée comme victime, elle est supprimée du page cache de manière suivante : si le *write-back* est actif et que les données n'ont pas étés écrites ens tockage secondaire, l'écriture est effectuée. Lorsque les données de la page sont en phase avec le stockage secondaire (*clean*), la page est simplement supprimée du cache.

L'évolution de la quantité de pages dans le page cache en fonction de la pression mémoire dans le système est un phénomène très complexe à modéliser, car il est impacté par de multiples facteurs indépendants de la gestion du stockage. En particulier, on peut penser aux multiples processus qui sont lancés dans le système, à l'allocation de mémoire dynamique réalisée par ces processus, etc. Modéliser un tel phénomène sort du cadre du travail présenté dans cette thèse. C'est pourquoi, dans notre implémentation fonctionnelle du page cache, on fait l'hypothèse que le système dispose d'une quantité de mémoire fixe (définie en nombre maximal de pages pouvant être stockées dans le page cache). Lorsque le page cache est plein, les pages contenues sont évincées suivant une politique LRU. L'algorithme de réclamation par Linux des pages dans le page cache est relativement complexe du fait des multiples types de pages contenues dans le cache comme indiqué dans Bovet et Cesati (2005). Comme JFFS2 ne supporte pas le *write-back*, les données en flash sont toujours à jour par rapport aux pages dans le page cache, une page évincée du cache est donc simplement supprimée.

Modèle fonctionnel VFS : *vfs_read()* et *read-ahead* On présente ici les modèles fonctionnels de *vfs_read()* et de l'algorithme *read-ahead*. Ces derniers sont, comme dans la réalité, fortement couplés.

L'algorithme 13 représente le modèle fonctionnel pour la fonction noyau *vfs_read()*. Ses rôles principaux sont (A) le découpage de la requête en pages Linux (lignes 2 et 3) et (B) le lancement de la lecture via le FFS (niveau inférieur) sur chacune de ces pages (lignes 4 à 18), si toutefois la page concernée n'est pas présente dans le page cache. Cette lecture peut appeller l'algorithme *read-ahead* si ce dernier est activé. C'est le cas par défaut avec JFFS2.

Dans le cas où read-ahead est activé, la procédure est la suivante : si l'une des pages à lire est trouvée dans le page cache et possède le flag read-ahead activé, une passe read-ahead asynchrone est effectuée (lignes 5 à 9). Si la page n'est pas trouvée dans le page cache, une passe synchrone est lancée (ligne 11 à 13). Lorsque read-ahead est desactivé, les pages Linux concernées sont lues dans une boucle (ligne 15).

Algorithme 13 : Description algorithmique représentant le modèle fonctionnel de ***vfs_read()***

```
1  Fonction VfsRead(F, offset, count)
       entrées  : F identifiant de fichier accédé, offset adresse cible de la lecture dans le fichier (en octets), count
                  nombre d'octets à lire
       sorties  : Appel(s) à read-ahead si ce mécanisme est activé : AsynchronousReadAhead,
                  SynchronousReadAhead, ou directement au FFS si read-ahead est désactivé : FfsReadPage
       données  : page_cache ensemble des pages présentes dans le page cache, Linux_page_size taille d'une page
                  Linux (généralement 4 Kio)
2      first_page_index = offset / Linux_page_size;
3      last_page_index = (offset + count − 1) / Linux_page_size;
4      pour P_current = P_first_page_index jusqu'à P_last_page_index faire
5          si P_current ∈ page_cache alors
6              si Read-ahead est actif et P_current possède un flag read-ahead actif alors
7                  Désactiver le flag read-ahead de P_current;
8                  AsynchronousReadAhead (F, P_current, last_page_index − current + 1);
9              fin
10         sinon
11             si Read-ahead est actif alors
12                 SynchronousReadAhead (F, P_current, last_page_index − current + 1);
13             fin
14             si P_current ∉ page_cache alors
15                 FfsReadPage (F, P_current)
16             fin
17         fin
18     fin
```

Concernant le modèle fonctionnel pour read-ahead, les deux fonctions présentées dans l'algorithme 14 représentent les deux points d'entrées (passes synchrones et asynchrones) du mécanisme. Il s'agit d'une simple encapsulation de la fonction principale présentée dans l'algorithme 15.

Algorithme 14 : Modélisation fonctionnelle de l'algorithme de pré-chargement read-ahead, partie 1 : points d'entrées *SynchronousReadAhead()* et *AsynchronousReadAhead()*.

```
1 Fonction SynchronousReadAhead(F, P_current, request_size)
      entrées : F identifiant de fichier accédé, P_current page actuellement demandée par le processus, request_size
                nombre de pages demandées par le processus
      sorties : Appel à OnDemandReadAhead, fonction centrale de l'algorithme de pré-chargement
2     OnDemandReadAhead (F, P_current, request_size, async_mode = 0);
3 Fonction AsynchronousReadAhead(F, P_current, request_size)
      entrées : Idem que pour SynchronousReadAhead
      sorties : Idem que pour SynchronousReadAhead
4     OnDemandReadAhead (F, P_current, request_size, async_mode = 1);
```

Le cœur du modèle fonctionnel de read-ahead est présenté dans l'algorithme 15. Au niveau noyau il s'agit de la fonction *ondemand_readahead()*. Le comportement algorithmique de read-ahead a été décrit en détail au chapitre précédent (voir pages 88 à 90). L'algorithme 15 reprend les mêmes principes.

Lorsque la taille de la fenêtre read-ahead est définie par la fonction principale, la lecture sur la fenêtre est assurée par la fonction représentée sur l'algorithme 16. Dans une boucle, les pages de la fenêtre sont lues une à une. On rappelle que MTD ne supportant pas les accès asynchrones, l'intégralité des pages de la fenêtre est lue avant que l'algorithme read-ahead ne retourne.

Modèle fonctionnel VFS : *vfs_write()* Le modèle fonctionnel pour *vfs_write()* est présenté dans l'algorithme 17. La requête d'écriture applicative est premièrement découpée en pages Linux (lignes 2 et 3). Ensuite, la fonction itère sur chaque page à écrire et appelle les fonctions du FFS concerné : dans le cas de JFFS2 il s'agit de *jffs2_write_begin()* suivi de *jffs2_write_end()* (lignes 8 et 9). A noter que contrairement à la fonction de lecture de page (*jffs2_readpage()*) qui lit des pages Linux entières, indépendamment de la plage de données demandée par l'applicatif, les fonctions FFS d'écriture ne ré-écrivent pas une page Linux entière lorsque seulement un sous-ensemble de la page est accédé. Ainsi, pour chaque page concernée par la requête d'écriture applicative, on calcule la plage de données concernée par l'écriture au sein de la page (lignes 6 et 7, 10 et 11).

VFS : Autres appels systèmes modélisés En plus de *vfs_read()* et *vfs_write()*, les équivalents VFS des appels système *creat()*, *remove()*, *open()*, *close()*, *truncate()*, *sync()* et *rename()* sont modélisés au niveau fonctionnel dans VFS. Ils appellent directement les implémentations du modèle fonctionnel FFS sous-jacent. De plus, on rajoute au niveau de l'interface entre l'applicatif et VFS une fonction que l'on nomme *vfs_drop_cache()*, qui correspond au vidage du page cache. Il ne s'agit pas d'un appel système à proprement parler, mais on rappelle que sous Linux on peut vider le page cache via la commande suivante :

```
$ echo 1 > /proc/sys/vm/drop_caches
```

b) VFS : Modèle de charge d'E/S

La couche VFS présente une liste de fonctions représentant une interface avec la couche applicative. Cette interface est constituée des fonctions suivantes :

- *vfs_read()* et *vfs_write()* ;
- *vfs_creat()* et *vfs_remove()* ;
- *vfs_open()* et *vfs_close()* ;
- *vfs_truncate()* ;

Algorithme 15 : Modélisation fonctionnelle de read-ahead, partie 2 : coeur de l'algorithme *On-DemandReadAhead()*.

```
1  Fonction OnDemandReadAhead(F, P_current, request_size, async_mode)
       entrées : F identifiant de fichier accédé, P_current page actuellement demandée par le processus, request_size
                 nombre de pages demandées par le processus, async_mode booleen indiquant le mode asynchrone
                 (True) ou synchrone (False)
       sorties : Un appel à DoFsCalls() qui lance la lecture sur la fenêtre
       données : Variable globale W représentant la fenêtre read-ahead
2      si P_current est la première page de F alors
3          aller à reset_window;
4      fin
5      si async_mode == 1 alors
6          si la requête actuelle est séquentielle par rapport à la précédente alors
7              aller à perform_read;
8          sinon
9              si Distance à la prochaine page ∈ F dans le PC est inférieure à un seuil donné alors
10                 aller à perform_read;
11             sinon
12                 retourner sans lecture au niveau read-ahead // vfs_read() se chargera des lectures
13             fin
14         fin
15     fin
16     si request_size > un seuil donné ou la requête actuelle est séquentielle par rapport à la précédente alors
17         aller à reset_window;
18     fin
19     si il existe des traces d'un flux séquentiel passé dans F alors
20         aller à perform_read;
21     fin
22     positionner la fenêtre W pour qu'elle n'inclue uniquement que les pages demandées par le processus;
23     DoFsCalls (W);
24     retourner
25     reset_window:
26     re-initialiser W en fonction de la requête courante (P_current et request_size) et de la taille max de fenêtre
       supportée par le FFS;
27     perform_read:
28     repositionner W et augmenter sa taille si nécessaire;
29     DoFsCalls (W);
30     retourner
```

Algorithme 16 : Modélisation fonctionnelle de read-ahead, partie 3 : accès au FFS par *DoFsCalls()*.

```
1  Fonction DoFsCalls(W)
       entrées : Fenêtre à lire W
       sorties : Série d'appels à FfsReadPage
2      pour P_current ∈ W faire
3          FfsReadPage (W);
4          si l'indice de P_current == W.size − W.async_size alors
5              activer le flag read-ahead sur P_current;
6          fin
7      fin
```

— *vfs_rename()* ;
— *vfs_sync()*
— *vfs_drop_caches()*.

Algorithme 17 : Description algorithmique représentant le modèle fonctionnel de ***vfs_write()***

```
1  Fonction VfsWrite(F, offset, count)
       entrées : F identifiant de fichier accédé, offset adresse cible de l'écriture dans le fichier, count nombre
                 d'octets à écrire
       sorties : Série d'appel(s) en écriture au FFS sous-jacent, via FfsWriteBegin et FfsWriteEnd
       données : Linux_page_size taille d'une page Linux (généralement 4 Ko)
2      first_page_index = offset/Linux_page_size;
3      last_page_index = (offset + count − 1)/Linux_page_size;
4      bytes_written = 0;
5      pour P_current = P_first_page_index jusqu'à P_last_page_index faire
6          offset_in_page = offset mod Linux_page_size;
7          count_in_page = min(count − bytes_written, Linux_page_size − offset_in_page);
8          FfsWriteBegin (F, P_current, offset_in_page, count_in_page);
9          FfsWriteEnd (F, P_current, offset_in_page, count_in_page);
10         offset = offset + count_in_page;
11         bytes_written = bytes_written + count_in_page;
12     fin
```

Le modèle de charge représente la charge appliquée par l'application sur le système de stockage. On définit la représentation d'une charge d'E/S comme une suite d'appels système avec (A) un temps d'arrivée dans le système (le moment ou l'appel est émis par l'application) et (B) une liste de paramètres relatifs à l'appel en question. La liste des appels supportés par le modèle de charge est représenté dans la table 4.5.

Nom de l'appel	Description	Paramètres	Modèle fonctionnel VFS appelé
read()	Lecture dans un fichier	• Identifiant de fichier • Adresse cible • Taille à lire	*vfs_read()*
write()	Écriture dans un fichier	• Identifiant de fichier • Adresse cible • Taille à écrire	*vfs_write()*
open()	Ouverture d'un fichier	• Identifiant de fichier	*vfs_open()*
close()	Fermeture d'un fichier	• Identifiant de fichier	*vfs_close()*
creat()	Creation d'un fichier	• Identifiant de fichier • Taille du nom	*vfs_creat()*
remove()	Suppression d'un fichier	• Identifiant de fichier	*vfs_remove()*
rename()	Re-nommage d'un fichier	• Identifiant de fichier • Taille du nouveau nom	*vfs_rename()*
truncate()	Troncature d'un fichier	• Identifiant de fichier • Adresse cible	*vfs_truncate()*
drop_caches()	Vidage du page cache	(aucun)	*vfs_drop_cache()*
sync()	Synchronisation du système de fichier avec le média de stockage	(aucun)	*vfs_sync()*

Table 4.5 – Liste des appels systèmes et paramètres associés représentés par le modèle de charge.

Dans la suite d'appels système représentant une charge d'E/S donnée, les appels sont ordonnés dans le temps. Concrètement, une application du modèle de charge d'E/S consiste à représenter la suite d'appels dans un fichier contenant un appel par ligne. Un exemple est présenté ci-dessous :

```
# Les commentaires sont préfixés par '#'
```

```
# Specifications du format :
# <temps (ms)>;<appel systeme>;<param. 1>;<param. 2>;...;<param. n>

# Creation d'un fichier d'id. 2 dont le nom fait 10 caracteres
10.0; create; 2; 10
15.0; open; 2; 0; 0; 0 # Ouverture du fichier d'id. 2
20.0; write; 2; 0; 16384 # Ecriture de 16 Ko à l'adresse 0
30.0; drop_cache # Vidage du page cache
40.0; read; 2; 0; 4096 # Lecture des 4 premiers Ko du fichier
55.0; close; 2 # Fermeture
```

Ce fichier est pris en entrée par le simulateur qui sera présenté au chapitre suivant.

c) VFS : modèles de performances et de consommation

On présente ici les modèles de performances et de consommation pour *vfs_read()* et *vfs_write()*. Le coût en temps d'exécution et énergie de read-ahead est inclus dans le modèle concernant *vfs_read()*. On a vu que chacune de ces deux fonctions est principalement constituée d'une boucle itérant sur le nombre de pages Linux à lire / écrire. L'overhead en performance et énergie est donc ici modélisé en fonction de ce nombre de pages Linux accédées. Contrairement aux fonctions du système de fichiers *xxx_write_begin()* / *end()* et *xxx_readpage*, qui ciblent une taille de données maximale d'une page Linux (4 Ko), *vfs_write()* et *vfs_read()* peuvent concerner une quantité de données très importante : on peut, par exemple, réaliser en un seul appel *read()* la lecture d'un fichier de 10 Mo, ce qui représente 2560 pages Linux de 4 Ko.

De plus, concernant *vfs_read()*, on verra dans la section traitant de l'extraction de paramètres que l'overhead par page est légèrement différent lorsque les pages Linux demandées sont (A) présentes dans le page cache au moment de la lecture ou (B) non présente dans le page cache. On modélise alors l'overhead de *vfs_read()* par page en deux termes, comme présenté ci-dessous :

Les modèles de performances et consommation au niveau VFS sont les suivants :

$$T_{VfsRead}(N_{p\_hit}, N_{p\_miss}) = N_{p\_miss} * (T_{FfsRead} + T_{Vfs\_Read\_Miss\_Overhead}) + N_{p\_hit} * T_{Vfs\_Read\_Hit\_Overhead} \quad (4.45)$$

$$T_{VfsWrite}(N_p) = N_p * (T_{FfsWriteBegin} + T_{FfsWriteEnd} + T_{Vfs\_Write\_Page\_Overhead}) \quad (4.46)$$

$$E_{VfsRead}(N_{p\_hit}, N_{p\_miss}) = N_{p\_miss} * (E_{FfsRead} + E_{Vfs\_Read\_Miss\_Overhead}) + N_{p\_hit} * E_{Vfs\_Read\_Hit\_Overhead} \quad (4.47)$$

$$E_{VfsWrite}(N_p) = N_p * (E_{FfsWriteBegin} + E_{FfsWriteEnd} + E_{Vfs\_Write\_Page\_Overhead}) \quad (4.48)$$

N_{p_hit} est le nombre de pages Linux concernées par une requête de lecture, et qui sont présentes dans le page cache au moment de cette requête. N_{p_miss} est le nombre de pages non présentes dans le cache, dont l'accès déclenche un appel au FFS. Concernant l'écriture N_p est le nombre de pages concernées par l'écriture.

4.2 VFS : méthodologie et exemple d'extraction de paramètres

Le but de l'extraction de paramètres au niveau VFS est de déterminer par la mesure les valeurs des différents overheads présentés ci-dessus.

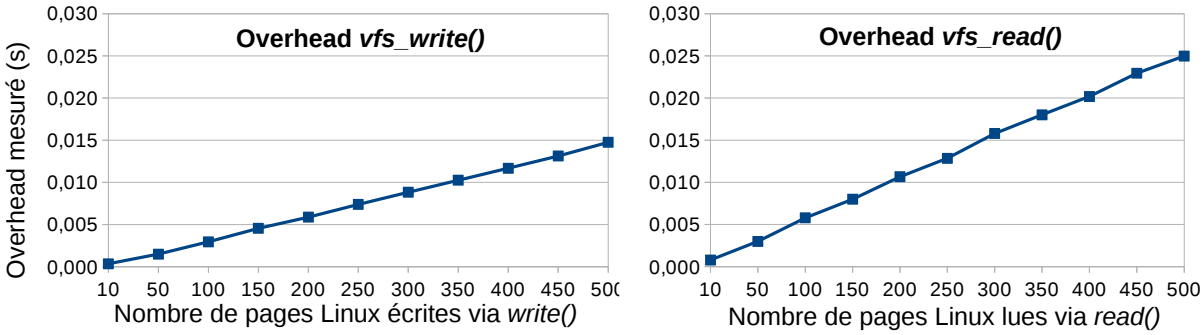


Figure 4.8 – Overheads mesurés pour *vfs_write()* (gauche) et *vfs_read()* (droite).

a) Extraction de paramètres VFS : performance

Écriture

• **Objectif :** extraction de $T_{Vfs\_Write\_Page\_Overhead}$.

• **Méthode :** pour mesurer l'overhead de VFS en écriture, on procède de manière suivante. Le but de l'expérience est d'observer l'évolution de l'overhead de *vfs_write()* en fonction du nombre de pages Linux écrites. Un micro-benchmark est lancé sur une partition flash nouvellement effacée de la carte Omap. Ce micro-benchmark effectue via l'appel système *write()* plusieurs écritures de pages séquentielles dans un fichier (créé vide au préalable). Comme VFS ramène les écritures via *write()* à la granularité des pages Linux, un unique appel à *write()* est réalisé. Le micro-benchmark est lancé plusieurs fois en variant la taille de la requête d'écriture, en nombre de pages Linux : un lancement est effectué pour 10 pages, les autres de 50 à 500 pages par pas de 50 pages. Pendant l'exécution du benchmark le temps d'exécution des fonctions *jffs2_write_begin()* et *jffs2_write_end()* est mesuré via Funcmon. Le temps d'exécution total de l'appel à *write()* est également mesuré. On obtient après traitement de la sortie de Funcmon une série de résultats dont un sous-ensemble est présenté dans la table 4.6.

Nombre de pages Linux écrites via l'appel à *write()*	**Temps d'exécution de l'appel à *write()* (s)**	**Somme des temps d'exécution de l'intégralité des appels à *jffs2_write_begin()* et *jffs2_write_end()* réalisés pendant l'appel à *write()* (s)**
10	0.00963	0.00928
50	0.04838	0.04689
100	0.09648	0.09352
...		
500	0.47948	0.46472

Table 4.6 – Résultats des micro-benchmarks concernant l'overhead de *vfs_write()*

Pour chaque résultat de benchmark, on soustrait alors le temps des appels au FFS du temps total de la requête *write()*. On obtient alors l'overhead de *vfs_write()* relatif à chaque exécution de benchmark. L'évolution de cet overhead en fonction du nombre de pages écrites est représentée sur la figure 4.8 à gauche. On peut voir que l'overhead évolue linéairement avec le nombre de pages Linux accédées, ce qui confirme la relation entre ces deux valeurs.

• **Résultats :** Pour calculer $T_{Vfs\_Write\_Page\_Overhead}$, on divise chaque overhead par le nombre de pages (de 10 à 500) correspondant, et on prend la moyenne de cette série de valeurs :

$$T_{Vfs\_Write\_Page\_Overhead} = 29.97\mu s \tag{4.49}$$

L'écart-type concernant cette moyenne par rapport à tous les tests réalisés est de $1.97\ \mu s$.

Lecture

• **Objectif :** extraction de l'overhead VFS en lecture.

• **Méthode :** on procède de manière similaire pour calculer l'overhead de *vfs_read()*. Avant le lancement de chaque micro-benchmark en lecture, read-ahead est desactivé et le page cache est vidé. L'évolution de l'overhead par rapport au nombre de pages lues, après soustraction des temps dus au FFS, est présenté à droite de la figure 4.8. L'overhead $T_{Vfs\_Read\_Miss\_Overhead}$ est calculé de la même manière que pour la fonction d'écriture. En relançant chaque micro-benchmark tout en s'assurant que les données à lire soient au préalable présentes dans le page cache, on obtient un overhead $Vfs\_Read\_Hit\_Overhead$ différent.

• **Résultats :**

$$T_{Vfs\_Read\_Miss\_Overhead} = 55.48\mu s \text{ (écart-type : } 8.3\mu s) \tag{4.50}$$

$$T_{Vfs\_Read\_Hit\_Overhead} = 39.74\mu s \text{ (écart-type : } 0.6\mu s) \tag{4.51}$$

b) Extraction de paramètres VFS : consommation

Les équations du modèle concernant l'énergie (équations 4.47 et 4.48) impliquent le fait de connaître les coûts en énergie des opérations au niveau FFS (par exemple $E_{Jffs2ReadPage}$). Or on a vu dans la section précédente que pour un certain nombre de limitations, nous ne pouvons les obtenir sur notre plate-forme matérielle. En revanche, les valeurs d'énergie des accès niveau pilote (MTD) ont précédemment été caractérisés. Les modèles de calcul de l'énergie pour *vfs_read()* et *vfs_write()* sont donc adaptés de manière suivante :

$$\begin{aligned} E_{VfsRead}(N_{hit}, N_{miss}) = {} & E_{MTD} + N_{hit} * E_{Vfs\_Read\_Per\_Hit\_Overhead} \\ & + N_{miss} * E_{Vfs\_Read\_Per\_Miss\_Overhead} \end{aligned} \tag{4.52}$$

$$E_{VfsWrite}(N) = E_{MTD} + N * E_{Vfs\_Write\_Page\_Overhead} \tag{4.53}$$

Dans ces équations, N_{hit} est le nombre de pages demandées par *vfs_read()* se trouvant dans le page cache, N_{miss} est le nombre de pages Linux à charger via le FFS. N est le nombre de pages Linux concernées par l'opération d'écriture. E_{MTD} correspond à l'énergie calculée par les modèles niveau pilote, concernant l'intégralité des opérations flash réalisées pendant l'exécution de *vfs_read()* et *vfs_write()*. Les occurrences et types de ces opérations flash sont bien entendu calculés par les différents modèles fonctionnels (VFS, FFS, pilote). Comme pour les performances, on modélise l'overhead en énergie des opérations de lecture et écriture VFS comme fonction du nombre de pages Linux lues et écrites. Comme détaillé dans l'extraction de paramètres ci-dessous, cet overhead contient l'énergie due à (A) la couche VFS et (B) la couche FFS, car il est obtenu par soustraction avec l'énergie consommée au niveau pilote. Le but de l'extraction de paramètre est donc de déterminer $E_{Vfs\_Read\_Hit\_Overhead}$, $E_{Vfs\_Read\_Miss\_Overhead}$ et $E_{Vfs\_Write\_Page\_Overhead}$.

On propose de déterminer ces valeurs de manière relativement similaire à ce qui est réalisé pour le temps d'exécution : en soustrayant à des mesures d'énergie au niveau VFS les mesures correspondant au niveau pilote, déterminée via le modèle niveau pilote présenté en début de chapitre. Dans les paragraphes suivants on présente l'extraction de paramètre réalisée pour *vfs_write()* sur la carte Omap.

• **Objectif :** extraction de l'overhead des écriture en énergie pour le niveau VFS.

• **Méthode :** une série de micro-benchmarks est lancée sur la carte Omap. Chaque micro-benchmark cible une partition JFFS2 nouvellement effacée, sur laquelle un fichier a été créé, vide à la base. Les benchmarks écrivent séquentiellement via *write()* dans le fichier une taille de données de 5 Mo. Chaque micro-benchmark effectue dans une boucle un nombre de requête *write()* variable : 10240, 5120, 2560, 1280, 640 et 320. La taille des données écrites par chaque appel à *write()* est fixée pour chaque benchmark en fonction du nombre d'appels, pour obtenir une taille totale de fichier de 5 Mo en fin d'expérience. Les tailles sont donc respectivement 512, 1024, 2048, 4096, 8192 et 16384 octets. La

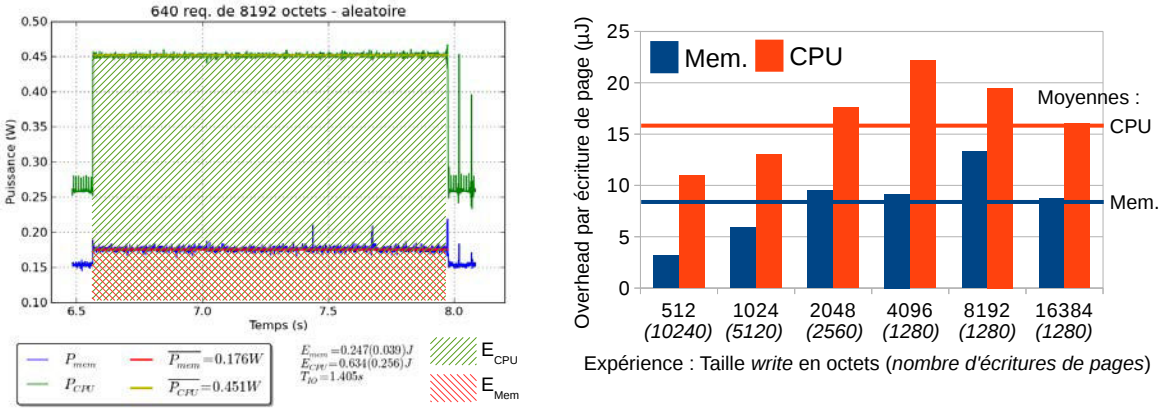


Figure 4.9 – A gauche : mesure de puissance et calcul de l'énergie consommée pendant l'un des micro-benchmark lancés dans le cadre de l'extraction de paramètres pour *vfs_write()*. **A droite** : overhead calculé pour *vfs_write()* via l'exécution des différents micro-benchmarks

consommation est mesurée via la plate-forme Open-PEOPLE durant l'exécution de chaque benchmark. On calcule à partir des résultats l'énergie consommée par les E/S, suivant la méthode décrite dans le chapitre 2 (voir équation 2.5 page 54). Un exemple de consommation mesurée pour l'expérience effectuant 640 requêtes de 8192 octets chacune est présenté à gauche sur la figure 4.9. En parallèle, on utilise les modèles fonctionnels et de consommation présentés jusqu'ici pour calculer l'énergie consommée au niveau pilote par chaque micro benchmark.

Pour chaque lancement de micro-benchmark, on soustrait à l'énergie totale des E/S mesurées ($E_{Measured}$ dans l'équation ci-dessous) l'énergie correspondant aux accès flash E_{MTD}, calculée via l'exécution des modèles précédemment décrits. Cela nous donne pour chaque benchmark l'énergie correspondant à l'intégralité des overheads pour tous les appels à *vfs_write()* ($E_{Total\_Overhead}$). Pour chaque benchmark cette valeur est alors divisée par *le nombre d'écritures de pages Linux* ($N_{Linux\_Pages\_Written}$) réalisées par le benchmark. Le nombre d'écritures de page Linux correspond au nombre d'appels à *jffs2_write_begin()* et *jffs2_write_end()* effectués au cours de l'exécution du benchmark. Par exemple, pour l'expérience réalisant 10240 requêtes *write()* de 512 octets, le nombre d'écritures de pages est 10240. Certes, dans le cas de ce benchmark les 8 premières écritures ciblent toutes la même page (page d'indice 0 dans le fichier), mais il y a néanmoins 8 appels aux fonctions d'écriture VFS. Pour l'expérience réalisant 640 écritures de 8192 octets, il y a 1280 (2 fois 640) appels au FFS (on rappelle que *vfs_write()* divise les requêtes au niveau de granularité d'une page Linux). Pour chaque benchmark on a donc :

$$E_{Total\_Overhead} = E_{Measured} - E_{MTD} \tag{4.54}$$

$$E_{Vfs\_Write\_Page\_Overhead} = E_{Total\_Overhead} / N_{Linux\_Pages\_Written} \tag{4.55}$$

• **Résultats :** à droite de la figure 4.9 on peut voir les résultats pour chacun des benchmarks. La carte Omap disposant de plusieurs points de mesures, on présente l'overhead mesuré sur le composant CPU et le composant mémoire. A noter que l'overhead sur le composant mémoire (RAM + flash sur la carte Omap) est théoriquement uniquement dû à la mémoire vive car on a soustrait des valeurs mesurées la consommation au niveau pilote qui comprend celle de la mémoire flash. La figure montre que l'overhead calculé est assez variable suivant les benchmarks : entre 3 et 13 μJ pour la mémoire, et entre 10 et 22 μJ pour le CPU. Néanmoins ces valeurs sont globalement faibles par rapport à l'énergie consommée par une écriture de page flash (81 μJ, voir table 4.4 page 140), on choisit donc de prendre la moyenne pour tous les benchmarks pour déterminer l'overhead : 8.28 μJ pour la mémoire (écart-type 3.4 μJ) , et 16.5 μJ pour le CPU (écart-type 4.1 μJ).

On procède de manière similaire pour *vfs_read()*.

• **Objectif :** extraction de l'overhead des lectures en énergie pour le niveau VFS.

• **Méthode :** des micro-benchmarks effectuant des lectures séquentielles via *read()* sur un fichier JFFS2 de 50 Mo sont lancés, chaque benchmark effectuant un nombre de lectures différent dans une boucle : 1500, 3000, 6000 et 12000. La taille des données lues par chaque appel à *read()* est fixée pour tous les benchmarks à 4096 octets, car on rappelle que (1) *vfs_read()* découpe les requêtes en pages Linux et (2) même si l'applicatif ne demande à lire qu'un sous-ensemble d'une page Linux, la page entière est lue. La lecture est une opération plus rapide que l'écriture (à quantité de données égales), les nombres de pages lues sont donc ici supérieurs aux nombres de pages écrites lors de l'extraction de paramètres en écriture (*vfs_write()*). Les benchmarks sont lancés deux fois, lors de la première passe le page cache est vidé avant le lancement des lectures. Lors de la seconde passe on s'assure que les données à lire sont présentes dans le page cache avant les appels à *read()*. Dans le premier cas, les accès flash sont tracés (il n'y en a aucun dans le cas de la seconde passe de chaque benchmark) et les modèles niveaux pilote sont utilisés pour trouver les valeurs d'overhead qui nous intéressent.

• **Résultats :** on détermine $E_{Vfs\_Read\_Per\_Miss\_Overhead}$ à 18.16 μJ (CPU, écart-type 0.7 μJ) et 11.91 μJ (mémoire, écart-type 0.4 μJ) ; et $E_{Vfs\_Read\_Per\_Hit\_Overhead}$ à 6.16 μJ (CPU, écart-type 0.07 μJ) et 4.37 μJ (mémoire, écart-type 0.07 μJ). Les valeurs d'énergies supérieures en cas de cache miss dans le page cache correspondent au sur-coût engendré par l'appel au FFS effectué dans ce cas.

c) Extraction de paramètres VFS : conclusion

La table 4.7 résume les résultats de l'extraction de paramètres niveau VFS. Les overheads en énergie sont divisés en overhead relatifs (A) à la mémoire et (B) au CPU, du fait des deux points de mesure utilisés sur la carte Omap.

Performances (μs)		**Consommation** (μJ)			
Paramètre	**Valeur**	**Paramètre**	**CPU**	**Mémoire**	**Total**
$T_{Vfs\_Write\_Page\_Overhead}$	29.97	$E_{Vfs\_Write\_Page\_Overhead}$	16.5	8.28	24.78
$T_{Vfs\_Read\_Miss\_Overhead}$	55.48	$E_{Vfs\_Read\_Miss\_Overhead}$	18.16	11.91	30.07
$T_{Vfs\_Read\_Hit\_Overhead}$	39.74	$E_{Vfs\_Read\_Hit\_Overhead}$	6.16	4.37	10.53

Table 4.7 – Résultats de l'extraction de paramètre au niveau VFS

On rappelle que les valeurs d'overhead en énergie contiennent également l'overhead correspondant à l'exécution du FFS, alors que les valeurs de temps ne contiennent que l'overhead de la couche VFS.

5 Ensemble de modèles pour la représentation d'une infrastructure multi-puces complexe de type SSD

Comme énoncé précédemment, on souhaite que le simulateur, dans lequel sont implémentés les modèles présentés ici, puisse également simuler des systèmes à base de FTL. Dans le domaine de l'embarqué, on peut notamment penser à l'utilisation assez répandue de cartes SD comme stockage secondaire, ou au développement important des puces eMMC. Pour ce faire, on propose d'augmenter et de raffiner le jeu de modèles présentés dans ce chapitre suivant les points ci-dessous :

— On ajoute un nouvel ensemble de modèles fonctionnels représentant l'implémentation d'un système de gestion flash à base de FTL. Cet ensemble peut se substituer aux modèles fonctionnels correspondant à un système à base de FFS (VFS, FFS, pilote MTD), comme présentés dans ce chapitre. Ce nouvel ensemble de modèles permet de décrire les différentes fonctionnalités d'une FTL. De plus, dans de nombreux systèmes à base de FTL (en particulier les SSD), on retrouve un buffer de mémoire vive qui tamponne les données lues et écrites dans le périphérique. Un

modèle fonctionnel est également défini pour ce buffer. Suivant les mêmes principes que pour les modèles concernant les FFS, des modèles de performances et de consommation concernant les évènements relatifs à la couche FTL doivent être définis. Ici on ne présente pas de modèles de performance / consommation concernant les évènements de la couche de gestion de type FTL ;

— Ce nouvel ensemble implique la définition d'un nouveau type de modèle de charge. En effet son utilisation implique la représentation d'un périphérique de type bloc. Une charge niveau bloc doit être utilisée en entrée ;
— Les modèles structurels relatifs au composant matériel flash doivent être raffinés pour permettre la description d'une architecture complexe de puces flash organisées de manière similaire à ce que l'on peut trouver dans un SSD : une hiérarchie de pages, blocs, plans, LUNs, et canaux ;
— Le modèle opérationnel doit lui-même être raffiné pour supporter les opérations dites *avancées* qui vont de pair avec le modèle structurel complexe ;
— Les modèles de performances / consommation relatifs au composant flash doivent enfin être raffinés pour intégrer les opérations avancées.

L'intégration de ces modèles dans ce qui a été présentée jusqu'ici est présenté sur la figure 4.10. Cette figure sera utilisée pour illustrer chacun des nouveaux types de modèles présentés dans les sections ci-dessous.

Dans la pratique, un jeu de modèles concrets correspondant à une FTL à base de traduction par page a été réalisé, et implémenté dans le simulateur présenté au chapitre suivant. Ces modèles concrets sont très simples et représentent une étude de faisabilité concernant les modèles orientés FTL.

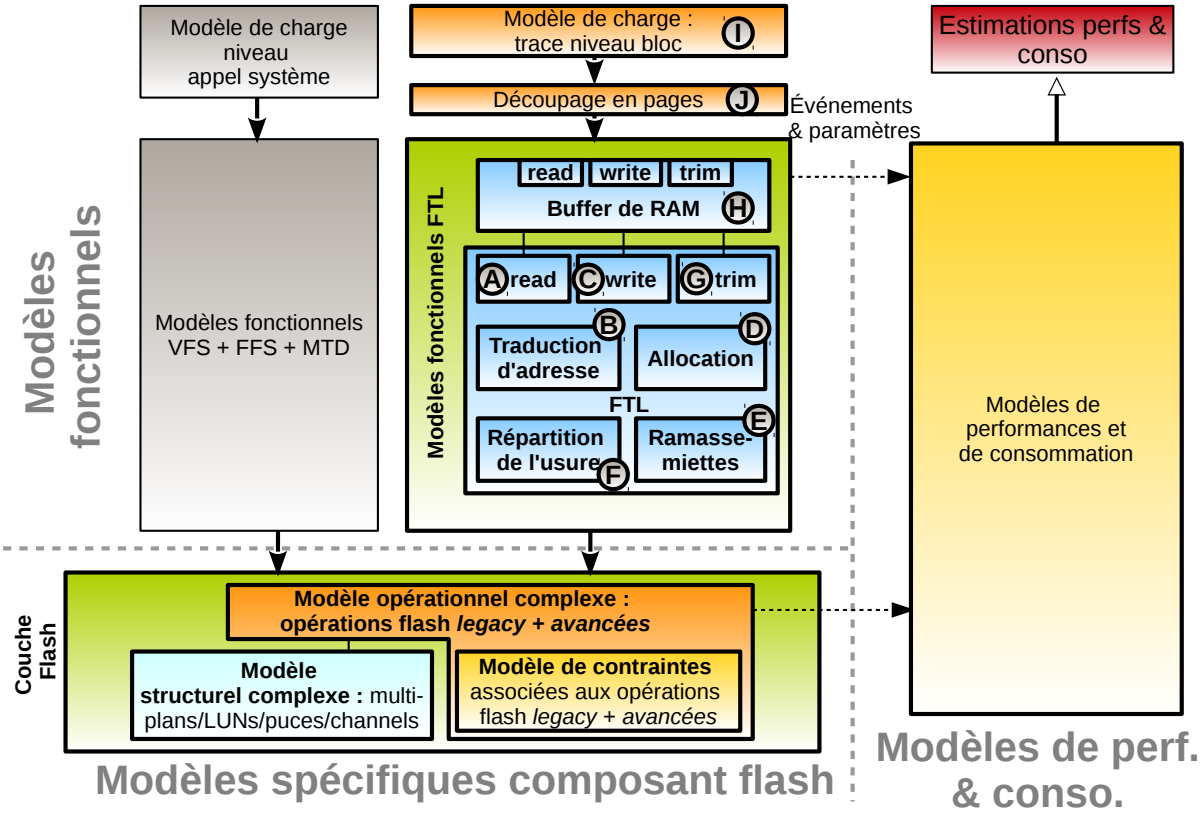


Figure 4.10 – Interactions entre l'intégralité des modèles définis dans le cadre de ce travail de thèse.

Dans les sections suivantes on présente les modèles abstraits qui définissent les règles de représentation d'un système à base de FTL dans notre infrastructure. Par souci d'espace le modèle fonctionnel réalisé correspondant à la FTL à base de traduction par page est présenté en annexe.

5.1 Ensemble de modèles fonctionnels représentant un système à base de FTL

On fait l'hypothèse qu'un système de stockage à base de mémoire flash de type FTL décrit via les modèles présentés ici est un périphérique de type bloc. C'est le cas des implémentation actuelles des systèmes à base de FTL (Bjørling et coll., 2010b). Un modèle fonctionnel relatif à une FTL doit être capable de fournir des algorithmes spécifiant le traitement des opérations niveau bloc suivantes : la lecture et l'écriture de données, ainsi que la commande *TRIM*. La commande TRIM (Serial ATA International Organization, 2011) est une commande ajoutée à des standards d'interfaces d'E/S de type bloc tels que ATA, SCSI ou eMMC. Cette commande permet à l'applicatif de notifier la FTL que certaines données en flash ne sont plus nécessaires et qu'elles peuvent être invalidées. Cela évite par la suite de coûteuses passes du ramasse-miettes qui, sans TRIM, déplacerait de nombreuses données invalides du point de vue de la couche applicative mais considérées comme valides par la FTL qui elle n'a pas cette information. Le cas typique d'usage est la suppression d'un fichier.

Dans les paragraphes ci-dessous on tente d'énoncer pour chaque opération (lecture, écriture, *TRIM*) les procédés globaux communs à toute implémentation de FTL. Bien entendu, en pratique, le déroulement de ces opérations est très dépendant du modèle de FTL utilisé et de son implémentation.

La *lecture* (A sur la figure 4.10) est gérée de manière relativement simple : le mécanisme de traduction d'adresse (B) est consulté pour déterminer la page flash physique correspondant à l'adresse logique demandée par l'applicatif. La page physique est ensuite lue. L'implémentation de la traduction d'adresse est très dépendante du modèle de FTL, comme vu au premier chapitre (voir pages a) à b)).

Dans le cadre de *l'écriture de données* (C sur la figure 4.10), la première étape consiste à sélectionner la page flash libre qui sera effectivement écrite. Il s'agit de l'*allocation* (D). C'est un choix non trivial, en particulier dans les systèmes de type SSD mettant en œuvre plusieurs niveaux de parallélismes, comme indiqué dans Hu et coll. (2011). Une fois la page choisie, elle est écrite et les structures de traduction d'adresse (B) sont mises à jour. Lors d'une requête d'écriture, le ramasse-miettes (E) peut être lancé si certains seuils (par exemple un seuil limite d'espace libre) sont atteints. Les stratégies d'écriture (allocation) ainsi que le comportement du ramasse-miettes peuvent selon le modèle de FTL considéré prendre en compte des critères de répartition de l'usure (F).

Lors de la réception d'une commande *TRIM* (G), la FTL invalide les données concernées. Cette opération est tout à fait dépendante du modèle de FTL considéré.

De nombreux systèmes à base de FTL font usage, tout comme les FFS, d'un *ramasse-miettes en arrière plan*, recyclant des données invalides en espace libre pendant les temps d'inactivité du périphérique de stockage. On a vu dans l'état de l'art que les simulateurs actuels ne permettent pas l'implémentation d'un tel système. Le simulateur proposé au chapitre suivant offre une telle fonctionnalité.

De nombreux systèmes, notamment les SSD, présentent un *buffer de mémoire vive* (H) implanté au sein du périphérique qui tamponne les données lues et / ou écrites dans le but d'améliorer les performances générales (Kim et Ahn, 2008; Park et coll., 2006a; Boukhobza et coll., 2011). Ce buffer est situé en amont de la FTL.

5.2 Modèle de charge niveau bloc

Les modèles représentant un périphérique de type bloc, un nouveau modèle de charge d'E/S doit être utilisé (I sur la figure 4.10). On propose d'utiliser le format de traces d'E/S niveau bloc dit *ascii*, popularisé par le simulateur de disques durs *DiskSim* (Bucy et coll., 2008), et utilisé dans de nombreux simulateurs orientés FTL (Microsoft Research, 2009; Gupta et coll., 2009; Hu et coll., 2011). Ce format se présente sous la forme d'une liste de requêtes décrites comme suit :

```
# <temps d'arrivée en ms> <identifiant de disque> <adresse de la req.>
     <taille de la req.> <type de req. (0 = write, 1 = read)>
0.000000 0 303567 7 0
```

```
26.214000 0 303574 7 0
117.964000 0 303581 7 0
```

Les unités d'adresse et de taille de requête sont spécifiées en *secteurs* (généralement 512 octets). Une couche supplémentaire de traitement de la charge pour ramener ces unités au niveau de la page flash (J) est nécessaire.

5.3 Modèles structurel et opérationnel correspondant à une architecture multi-puce complexe supportant les commandes avancées

a) Modèle structurel raffiné

La structure hiérarchique flash présentée précédemment, contenant uniquement les notions de page et de bloc, est raffinée en rajoutant les niveaux hiérarchiques suivants :

— Un ensemble de blocs est contenu dans un *plan* ;
— Un ou plusieurs plans sont contenus dans une *unité logique* (LUN) ;
— Les LUN sont organisées en *canaux*.

b) Modèle opérationnel raffiné

Un modèle flash opérationnel complexe est proposé. Ce modèle contient les opérations *legacy* (déjà présente dans le modèle présenté dans ce chapitre) et *avancées*. Les opérations avancées, décrites au chapitre 1 (voir page b)) sont les commandes suivantes :

— La lecture et l'écriture en *mode cache* ;
— Les lectures / écritures / effacements *multi-plans* ;
— Les opérations *entrelacées entre plusieurs LUN* ;
— Les opérations profitant du *parallélisme pur multi-canaux* ;
— Les *combinaisons* de plusieurs des commandes citées ci-dessus, on peut par exemple réaliser une écriture en mode cache sur deux plans d'une même LUN en même temps (*multi-plane cache write*). D'autres opérations combinées complexes sont également possibles.

Le modèle flash opérationnel associé à chacune de ces opérations consiste en (A) une description de la manière dont l'état des pages flash est modifié par l'exécution de l'opération et (B) une série de contraintes quant à l'utilisation de cette opération. Ci-dessous on présente deux exemples de modèles opérationnels pour opérations avancées : l'opération d'écriture en mode cache, et l'opération de lecture multi-plans. La description détaillée de toutes les opérations avancées modélisées est présentée en annexe A (voir page 209) de cette thèse.

Exemple de modèle opérationnel : écriture en mode cache On rappelle que l'écriture en mode cache sous-entend l'introduction au sein d'un plan d'un cache buffer, de la taille d'une page flash, situé entre le page buffer (de même taille) et la matrice de transistor NAND. En mode cache, plusieurs pages flash physiquement contiguës sont écrites. Le cache buffer permet de pipeliner (recouvrement temporel) deux opérations : (1) l'écriture d'une page depuis le cache buffer dans la matrice de transistor NAND et (2) le transfert de la page à écrire suivante sur le bus d'entrée sortie depuis l'hôte vers le page buffer.

Une opération d'écriture en mode cache est définie par une adresse P_{Start} représentant la première page flash à écrire, et un nombre de pages n à écrire. Les contraintes associées sont les suivantes : (A) l'ensemble de pages accédées doit être contenu au sein d'un même plan, (B) n doit être supérieur ou égal à 2, (C) toutes les pages à écrire doivent être libres et (D) la contrainte d'écriture séquentielle dans un bloc doit être respectée pour tous les blocs concernés par l'écriture.

L'exécution d'une écriture en mode cache modifie l'état des pages flash concernées comme décrit dans l'algorithme 18.

Algorithme 18 : Modèle opérationnel de l'écriture en mode cache

```
1 Fonction CacheWrite(P_Start, n)
      entrées : P_Start page de départ, n nombre de pages à écrire en séquentiel à partir de P_Start
2     pour P_current = P_Start jusqu'à P_Start + n − 1 faire
3         P_current.state = occupied;
4     fin
```

Exemple de modèle opérationnel : lecture multi-plans La lecture multi-plans consiste à lire en parallèle plusieurs pages, chacune présente dans un plan différent d'une même puce. Plus précisément, les transferts de données depuis la matrice de transistors NAND vers le page buffer au sein de chaque plan sont réalisés en parallèle. Les différentes pages sont alors transférées une par une depuis les page buffers sur le bus d'E/S de la puce. De multiples contraintes s'appliquent à cette opération (et aux opérations multi-plans en général) :

1. Les pages ciblées doivent chacune être dans un plan différent d'une même LUN ;
2. L'adresse de la page à lire dans chaque plan est représentée par un couple (*indice de bloc dans le plan*, *indice de page dans le bloc*). Toutes les pages ciblées par une opération multi-plan doivent avoir le même couple d'adresses (*bloc*, *page*) dans le plan ciblé. Cette contrainte d'adressage forte propre aux opérations multi-plans limitte la hausse de performances apportées par ce type d'opération (Jung et coll., 2012) ;
3. L'opération *multi-plans* est effectuée sur *tous* les plans d'une même LUN.

Les 3 contraintes ci-dessus font que, pour caractériser une opération de lecture multi-plans, il suffit d'un indice de LUN ciblée L, ainsi que d'un indice de bloc et un indice de page dans ce bloc, représentant les pages qui seront lues en parallèle dans chaque plan de la LUN L.

Cette opération effectue des lectures et ne modifie pas l'état des pages de la mémoire flash ciblée.

5.4 Modèles de performances et de consommation pour architecture flash complexe

L'intégralité des modèles de performances et de consommation concernant les architectures flash complexes est présentée en annexe. On présente ici un exemple pour l'écriture en mode cache.

Écriture en mode cache : modèle de performances et de consommation On modélise le temps d'exécution d'une opération d'écriture en mode cache, concernant n pages flash, grâce aux sous-opérations précédemment présentées : T_{TON} et T_{IO}. La figure 4.11 représente le déroulement d'une telle opération. Si le système peut effectivement effectuer un recouvrement temporel (pipeline) des deux sous-opérations, deux dépendances viennent limiter le parallélisme : la dépendance (A) représente le fait qu'une page doit être transférée dans son ensemble depuis l'hôte sur le bus d'E/S dans le page buffer, avant de pouvoir être passée dans le cache buffer (1 cycle d'horloge, temps négligeable non modélisé) pour transfert vers la matrice NAND. La dépendance (B) représente le fait que le page buffer doit être libre avant de commencer un transfert depuis l'hôte sur le bus d'E/S vers ce page buffer. On

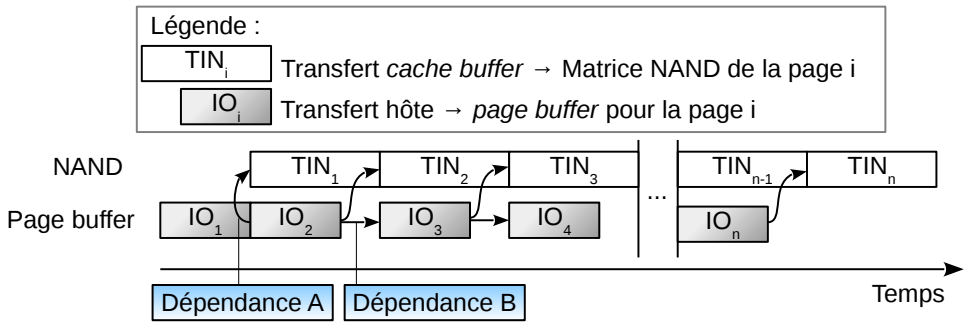


Figure 4.11 – Schéma du déroulement d'une opération de type écriture en mode cache

peut ainsi modéliser le temps d'exécution de l'opération d'écriture en mode cache comme suit :

$$T_{CacheWrite}(addr_1, addr_2, ..., addr_n) = T_{IO} + \sum_{i=2}^{n} max(T_{TIN_i}, T_{IO}) + T_{TIN_n} \quad \text{avec } n \geq 2 \tag{4.56}$$

Dans cette équation $addr_i$ représente l'adresse de la i-ème page dans un ensemble écrit en mode cache. T_{IO} représente le temps de transfert d'une page sur le bus. T_{TIN_i} représente le temps d'écriture d'une page d'adresse i dans la matrice de transistors NAND. A noter que ce temps peut être variable selon l'adresse de la page : on a vu que celà est possible dans les systèmes à base de puces MLC (Jung et coll., 2012; Grupp et coll., 2009). Cette caractéristique est prise en compte dans les modèles proposés.

L'énergie consommée est modélisée de manière simple : l'énergie consommée par chaque sous-opération se voit modélisée comme la multiplication de la puissance constatée pendant cette sous-opération par son temps d'exécution. L'énergie totale correspond alors à la somme des énergies de toutes les sous-opérations constituant l'opération d'écriture en mode cache :

$$E_{CacheWrite}(addr_1, addr_2, ..., addr_n) = \sum_{i=1}^{n} (P_{TIN} * TIN_i + P_{IO} * T_{IO}) \quad \text{avec } n \geq 2 \tag{4.57}$$

Lecture multi-plan : modèle de performances et de consommation La figure 4.12 page 164 représente le déroulement d'une opération mutli-plans : les pages sont lues en parallèles dans les différents plans, puis envoyées en séquentiel (dépendance A sur la figure) sur le bus d'E/S.

On modélise les performances de cette opération de manière suivante : On définit T_{P_i} comme étant le temps d'une opération de lecture d'une page dans le plan i.

$$\begin{aligned} T_{P_1}(addr_1) &= TON_{addr_1} + IO \\ T_{P_i}(addr_i) &= max(T_{P_{i-1}}, TON_{addr_i}) + IO \end{aligned} \tag{4.58}$$

$addr_i$ est l'adresse de la page à lire dans le plan i. Ainsi :

$$T_{MultiPlaneRead}(addr_1, addr_2, ..., addr_n) = max_i(T_{P_i}) \quad \text{avec } n \geq 2 \tag{4.59}$$

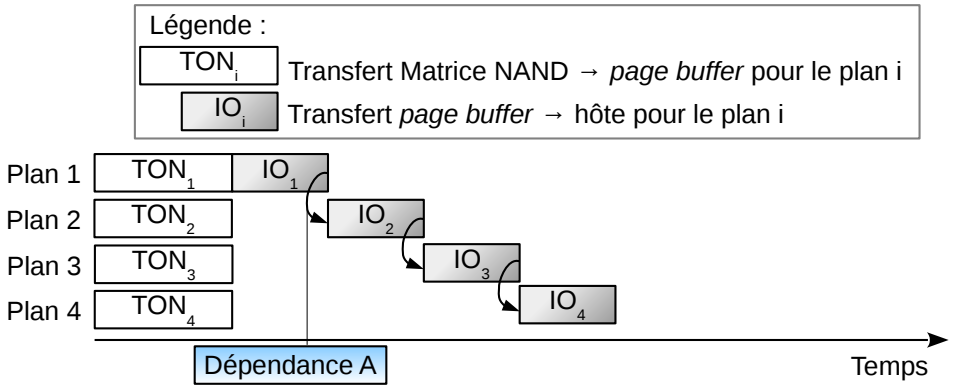


Figure 4.12 – Schéma du déroulement d'une opération de type lecture multi-plans

n est le nombre de plans participants à l'opération multi-plan. La consommation d'une opération de lecture multi-plans est quant à elle modélisée comme suit :

$$E_{MultiPlaneRead}(addr_1, addr_2, ..., addr_n) = \sum_{i=1}^{n}(PTON * TON_{addr_i}) + n * PIO * IO \quad \text{avec } n \geq 2 \tag{4.60}$$

Plusieurs études observent, pour une même puce, des variations dans les valeurs de TIN (et dans une moindre mesure de TON) dans le cadre des puces de type MLC (Jung et coll., 2012; Grupp et coll., 2009). Il faut noter que toutes les équations des modèles de performances (et de consommation) des opérations avancées prennent effectivement en compte le fait que TIN et TON peuvent être variables.

L'intégralité des autres opérations avancées à été modélisée de manière similaire aux exemples présentés ici : ce travail est inclus en annexe A page 209. Ces modèles n'ont pas fait l'objet d'une extraction de paramètres, et n'ont donc pas été validés. L'une des raison derrière cela est le fait qu'il n'est pas aisé d'extraire des informations sur le fonctionnement interne, les performances et la consommation des systèmes à base de FTL. Il s'agit en effet de systèmes propriétaires et fermés (*boîtes noires*). Une validation / extraction de paramètre pourrait néanmoins être réalisée sur des systèmes ouverts comme la plate-forme *OpenSSD* (OpenSSD contributors, 2014), avec l'aide d'instruments de mesure précis.

Les modèles relatifs aux systèmes à base de FTL, en particulier les modèles de performances et de consommation pour opérations flash avancées, sont implémentés dans le simulateur présenté au chapitre suivant. De plus une FTL basique de type traduction par page est implémentée. Hormis le modèle de performances et consommation relatif au composant flash, il n'y a pas de modèles concrets de performances / consommation réalisés pour la couche de gestion de cette FTL.

6 Conclusion

Plusieurs ensembles de modèles ont été présentés dans ce chapitre. Ces modèles permettent de décrire un système de stockage à base de mémoire flash. Des modèles de différents types sont définis, permettant de représenter les multiples paramètres qui impactent les performances et la consommation d'un système de stockage à base de mémoire flash. Ces différents types de modèles sont complémentaires et interagissent de diverses manières. Les types de modèles définis sont le suivants :

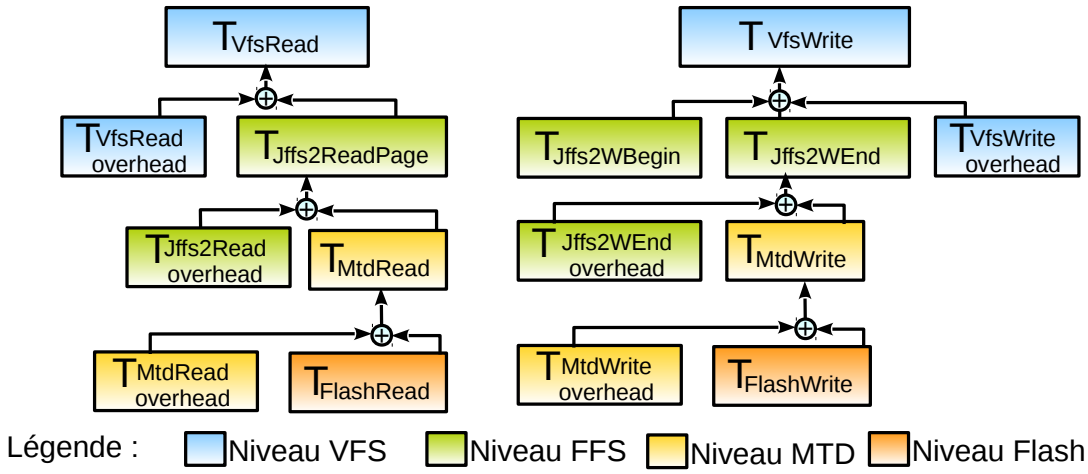


Figure 4.13 – Vision simplifiée de l'arborescence de modèles de performances (même principe pour les modèles de consommation)

1. Les modèles *fonctionnels* permettent de (A) représenter les différents algorithmes de la couche de gestion, (B) maintenir l'état du système, et (C) calculer les occurrences d'évènements dont on souhaite déterminer les performances et la consommation via les modèles de performances et de consommation. L'intégralité des niveaux de la pile de gestion des E/S sous Linux est prise en compte : les niveaux VFS, FFS, et pilote NAND ;
2. Les modèles de *performances* et de *consommation* permettent de calculer ces métriques concernant les différents évènements calculés par les modèles fonctionnels représentant la couche de gestion. Le composant matériel flash fait l'objet d'une modélisation relativement fine, en particulier du fait de son impact fort sur les performances : les temps d'exécutions et valeurs d'énergies pour chaque opération flash supportée par la ou les puce(s) flash sont modélisés. Concernant les composants RAM et CPU exécutant les algorithmes de la couche de gestion, leur impact est représenté sous la forme d'un overhead présent dans les différentes équations composant les modèles de performances et de consommation ;
3. Les modèles *structurels* et *opérationnels* viennent compléter la représentation fine du composant mémoire flash. Le modèle structurel décrit les paramètres architecturaux d'une ou plusieurs puces flash présentes dans un système de stockage. Les modèles opérationnels représentent les différentes opérations qu'il est possible d'appliquer à l'ensemble de puces, ainsi que la manière dont ces opérations modifient l'état des pages flash. De plus, ces modèles incorporent également une définition des contraintes relatives aux différentes opérations flash. Le maintient de l'état des pages flash, ainsi que la représentation des contraintes des opérations sont nécessaires. Ene ffet on souhaite que l'infrastructure dans laquelle les modèles sont implémentés, le simulateur, puisse elle-même se présenter comme une plate-forme de développement et de prototypage de nouveaux systèmes de gestions pour mémoire flash ;
4. Les modèles de *charges d'E/S* permettent de représenter la charge applicative appliquée au système de stockage et traitée par la couche de gestion.

L'approche adoptée pour la réalisation des modèles de performances et de consommation est la suivante : à chaque niveau (VFS / FFS / pilote / flash), le temps ou l'énergie d'une opération correspond au temps ou à l'énergie passé au niveau inférieur plus un overhead relatif à l'opération courante. On peut ainsi voir les modèles comme une arborescence, présentée sur la figure 4.13. A noter qu'il s'agit d'une vision simplifiée des modèles actuels :

— On rappelle que les caches présents à différents niveaux peuvent modifier le traitement d'une requête d'E/S : par exemple un cache hit dans le page cache lors d'une lecture ne déclenche

pas d'appels aux niveaux inférieurs ;
— Le temps et la consommation de *jffs2_write_begin()* peut potentiellement inclure le temps et la consommation d'une lecture de page Linux si la page concernée par l'écriture n'est pas présente dans le page cache ;
— Le ramasse-miettes n'est pas représenté, en particulier son impact sur le temps et la consommation de *jffs2_write_end()* lorsque la quantité d'espace flash libre est critique ;

Pour chaque modèle de performances et de consommation, une méthodologie d'extraction de paramètres est présentée. L'extraction de paramètres permet, en réalisant des micro-benchmarks, d'obtenir un profil de performances et de consommation correspondant à un environnement logiciel et matériel donné. Ce profil permet par la suite d'obtenir des estimations relatives à cet environnement grâce aux modèles de performances et de consommation. Dans ce chapitre, on a présenté un exemple de profil correspondant à la carte Omap exécutant le FFS JFFS2 sous Linux 2.6.37. Certaines limitations peuvent rendre difficile l'extraction de paramètres, par exemple la granularité de mesure des outils utilisés, ou la difficulté d'accès de certaines fonctions au cœur d'un système d'exploitation. On a présenté dans ce chapitre certaines techniques de mesures de performances et de consommation pour contourner ces limitations. Par exemple, on peut effectuer un nombre important d'opérations dans une boucle pour déterminer une puissance moyenne, ou encore intégrer un overhead en énergie au niveau de représentation supérieur, comme cela a été fait pour l'overhead JFFS2 intégré au niveau VFS. La validation des modèles sera présentée au chapitre suivant.

La technique d'extraction de paramètres est constituée d'un ensemble de micro-benchmarks opérants à plusieurs niveaux du système d'exploitation. Cette technique est ré-utilisables sur toutes les plate-formes matérielles supportant Linux. Dans ce travail de thèse, l'extraction de paramètres à été appliquée au FFS JFFS2. Pour appliquer la méthodologie de modélisation à d'autres FFS (en particulier YAFFS2 et UBIFS), il est nécessaire de (1) construire un modèle fonctionnel pour le FFS concerné et (2) appliquer la technique d'extraction de paramètres sur la plate-forme matérielle concernée.

Les modèles présentés ici peuvent couvrir l'ensemble du spectre d'application des systèmes de stockage à base de mémoires flash actuels : les systèmes à base de FFS, et également à base de FTL. Pour supporter les systèmes de type FTL les modèles sont raffinés :

— Les modèles relatifs au composant flash sont augmentés pour supporter (A) une structure flash complexe multi-plans/LUNs/canaux et (B) les opérations *avancées* qui vont de pair avec une telle structure, en particulier les opérations qui mettent en œuvre le parallélisme potentiel que l'on peut trouver dans des périphériques comme les SSD. Des modèles de performances et de consommation théoriques relatifs à ces opérations avancées ont également étés présentés ;
— Un modèle de charge simple pour périphérique de type bloc est utilisé ;
— Des indications sur la construction des modèles fonctionnels relatif à la couche FTL sont également données dans ce chapitre.

Réutilisabilité des modèles niveau FFS Les modèles suivants sont réutilisables dans le cas du développement d'un nouveau modèle fonctionnel de FFS (par exemple pour JFFS2 ou YAFFS2) :

— Les modèles fonctionnels et les modèles de performances / consommation pour VFS et MTD : ces couches sont utilisées par tous les FFS sous Linux ;
— Les modèles correspondant au composant flash : modèles structurel, opérationnel, de performances et de consommation ;
— Le modèle de charge d'E/S : ce dernier est générique car il cible le niveau appel système.

L'intégralité des modèles fonctionnels, le modèle de charge d'E/S, ainsi que les modèles structurel et opérationnel flash sont réutilisable dans le cas où l'on prend en considération une nouvelle plate-forme matérielle (autre que la carte Omap). La méthodologie d'extraction de paramètre présentée ici peut être appliquée sur une nouvelle plate-forme pour construire les modèles de performances et de consommation représentant son profil par rapport à ces métriques.

Les modèles de performances et consommation niveau pilote, ainsi que la méthodologie et les résultats relatifs à la carte Omap ont étés présentés dans une conférence (Olivier et coll., 2013a). Une présentation générale, relativement simplifiée, des interactions entre les modèles proposés dans ce chapitre a elle été présentée sous forme de poster (Olivier et coll., 2013b).

L'ensemble des modèles présentés dans ce chapitre sont intégrés dans un simulateur, présenté au chapitre suivant.

CHAPITRE 5

Estimation des performances et de la consommation par la simulation des systèmes de stockage à base de mémoire flash

Dans ce chapitre, on présente en premier lieu un simulateur, nommé OpenFlash*, développé dans le cadre de cette thèse, qui implémente les modèles proposés au chapitre précédent. Le simulateur, écrit en language C++, permet d'utiliser les modèles pour obtenir des estimations concernant les performances et la consommation d'un système de stockage à base de mémoire flash. Il s'agit d'un simulateur à évènements discrets prenant une trace d'E/S en entrée. Cet outil est présenté dans les trois premières sections de ce chapitre : en premier lieu, on rappelle les grandes lignes de l'état de l'art sur les simulateurs de systèmes de stockage à base de mémoire flash et l'on situe notre contribution, par rapport à ces travaux. Dans cette première section on effectue également une présentation générale d'OpenFlash, en décrivant globalement les différents modules logiciels le composant. Ces modules sont par la suite décrits en détail dans les deux sections suivantes : dans la deuxième section on présente les modules ayant trait à la gestion interne et aux entrées et sorties du simulateur. Il s'agit de la gestion de la charge en entrée, de la configuration d'une simulation, de la gestion du temps et des évènements dans le simulateur, et les différents résultats disponibles en sortie après une simulation. On s'intéresse ensuite en section 3 à l'intégration des modèles fonctionnels, de performances et de consommation au sein du simulateur.*

La section 4 traite de la validation des modèles et de leur implémentation dans le simulateur. On commence par présenter la méthodologie globale adoptée dans cette phase de validation. Ensuite, on présente une méthodologie de validation concernant (A) les modèles fonctionnels, (B) les modèles de performances et (C) les modèles de consommation. On donne également les résultats de cette méthodologie de validation appliquée aux modèles réalisés pour la carte Omap exécutant JFFS2 sous Linux. Les chiffres montrent que dans la majeure partie des cas, l'erreur d'estimation du simulateur par rapport à des mesures en environnement réel reste sous la barre des 10%.

Sommaire

1 Positionnement dans l'état de l'art et présentation générale du simulateur *OpenFlash*

1.1 Rappel sur l'état de l'art

Après la réalisation de l'état de l'art en début de document (voir chapitre 2 page 69) sur les simulateurs, les manques suivants ont étés observés :

— Il n'existe pas, à notre connaissance, de simulateur concernant les systèmes de stockage à base de mémoire flash gérée par FFS. La plupart des simulateurs actuels ciblent les systèmes de types FTL. Il s'agit effectivement d'une thématique de recherche relativement populaire aujourd'hui, du fait de la multiplicité de ce type de système ;
— Les différents simulateurs étudiés dans l'état de l'art ne proposent pas de support pour les opérations en arrière plan, en particulier les opérations de ramasse-miettes asynchrones. Ce type de processus est pourtant très présent dans des systèmes comme les SSD et les FFS ;
— On a vu que l'état initial (quantité d'espace libre / valide / invalide) influençait fortement les résultats de simulation. Les simulateurs présentés ne gèrent pas ou peu la mise en place d'un état initial pré-simulation. Dans la plupart des simulateurs c'est à l'utilisateur de mettre en place un état initial correct (phase appelée *warm-up* ou *pre-conditionning*).

On note également les conclusions suivantes :

— Pour la plupart des simulateurs ciblant des périphériques de type bloc, la granularité de modélisation adoptée est au niveau *couche de gestion* + *flash* ;
— Certains simulateurs proposent également la possibilité de prototyper de nouveaux mécanismes de couche de gestion (FTL). Cela permet de valider des concepts, et également de comparer plusieurs systèmes de gestion entre eux. ;
— Le rôle de la majeure partie des simulateurs est de proposer une exploration de l'espace de conception d'un système à base de mémoire flash en permettant à l'utilisateur de faire varier un certain nombre de paramètres relatifs aux modèles implémentés dans ces simulateurs. Des estimations concernant les performances et, pour certains simulateurs, la consommation sont disponibles en sortie ;
— Les modèles représentant l'ensemble des puces flash simulées sont plus complets dans certains simulateurs que dans d'autres. Les modèles complets implémentent (A) le support des architectures flash avancées (notions de plans / LUNs / canaux), (B) le support des opérations flash avancées (toutes les opérations non *legacy*) et (C) les modèles de performances et de consommation qui vont de pair avec ces opérations avancées. Les modèles simples ne permettent pas de représenter les systèmes de stockage à base d'architecture flash avancées.

Basé sur ces points, il est évident qu'aucun simulateur existant ne peut être ré-utilisé dans le cadre de ce travail de thèse, compte tenu des objectifs fixés. La granularité de simulation adoptée par la plupart des simulateurs et le non support de systèmes de types FFS rendent la simulation de tels systèmes très difficile, pour ne pas dire impossible.

On décide donc, dans ce travail de thèse, de développer un simulateur dédié. On souhaite qu'OpenFlash puisse combler les manques présents au niveau des simulateurs de systèmes de stockage à base de mémoire flash actuels, tout en reprenant les points positifs et importants implémentés par certains d'entre eux.

1.2 Positionnement de la contribution dans l'état de l'art

Dans cette sous-section on situe OpenFlash par rapport aux travaux existants : en reprenant chacun des points cités dans la sous-section précédente, on montre comment la proposition présentée ici tente d'y répondre.

Support des systèmes de type FFS Le simulateur proposé dans ce travail de thèse supporte la représentation et l'exécution de modèles décrivant de la mémoire flash embarquée gérée sous Linux via un système de fichiers dédiés aux mémoires flash. Des estimations concernant les performances et la consommation relatives au traitement d'une charge d'E/S applicative sont disponibles en sortie. OpenFlash implémente des modèles représentant l'intégralité de la couche de gestion dans un contexte de type FFS. Le simulateur implémente également les interactions entre cette couche de gestion et (A) l'applicatif (trace d'E/S) et (B) le composant mémoire flash. La couche de gestion "pure" (le FFS) étant fortement couplée avec des éléments relatifs au système d'exploitation comme VFS, le page cache, ou encore le mécanisme read-ahead, ces derniers sont également modélisés dans le simulateur. Un exemple de représentation de *JFFS2* et des couches associées (VFS et MTD) est implémenté dans OpenFlash, et ce dernier propose une infrastructure de classes permettant une intégration relativement aisée pour d'autres types de FFS.

Support des systèmes à base de FTL Si dans cette thèse l'accent est mis sur les FFS, l'outil proposé supporte également la simulation de systèmes à base de FTL. Un exemple d'implémentation fonctionnelle basique d'une FTL à base de *traduction par page* est proposé dans le simulateur. Dans l'état actuel des choses, des estimations concernant uniquement les performances et la consommation relatives aux accès flash sont disponibles après une simulation de type FTL. On peut imaginer la construction de modèles concernant les performances et la consommation de contrôleurs de systèmes à base de FTL, ou des buffers de mémoire vive comme on peut trouver dans les SSD.

Granularité de modélisation On peut considérer que la granularité de modélisation (voir l'état de l'art page 69) pour une simulation de type *FFS* est située au niveau *système d'exploitation*. En effet, on considère dans ce cas la mémoire flash, le système de fichiers, le système de fichier virtuel et la charge provenant directement de l'applicatif. Concernant une simulation de type FTL, la granularité de modélisation est clairement au niveau *couche de gestion + flash*, car seulement un périphérique de type bloc est modélisé.

Fonctionnalités du simulateur proposé : estimation et développement de couches de gestion On distingue deux buts principaux lors de l'utilisation du simulateur proposé. Premièrement, on souhaite qu'un usager puisse utiliser des modèles pré-établis pour obtenir des estimations concernant les performances et la consommation pour un système donné, par exemple lorsque l'utilisateur ne possède pas le système physique ou ne dispose pas d'appareils de mesures adaptés. Deuxièmement, l'utilisateur doit pouvoir développer de nouveaux systèmes de gestion au sein d'OpenFlash, que ce soit pour représenter le comportement de couches de gestion existantes, ou encore prototyper un nouveau modèle de couche de gestion. Ce faisant il est alors possible d'utiliser le simulateur pour effectuer des études comparatives entre plusieurs implémentations de couches de gestion.

Support des opérations en arrière plan De par la définition des mécanismes de gestion du temps et des évènements dans le simulateur, cet outil supporte la description et la simulation d'opérations en arrière plan. Il s'agit d'évènements qui ne sont pas présents dans la trace, mais qui sont insérés par la couche de gestion elle-même de manière asynchrone par rapport aux requêtes contenues dans la trace d'E/S en entrée. Ce sont typiquement des opérations de ramasse-miettes. Contrairement aux simulateurs qui ne supportent que la description d'opérations de ramasse-miettes synchrones (autrement dit qui retardent la requête d'E/S les ayant déclenchées), les opérations en arrière plan sont asynchrones et peuvent avoir lieu pendant les temps morts entre les E/S de la trace. Elles impactent la consommation et les performances.

Finesse de modélisation du composant mémoire flash Le simulateur supporte la modélisation d'une structure flash avancée, ainsi que des opérations avancées. Les modèles de performances et de consommation relatifs à ces opérations sont définis à une granularité moyenne, comme présenté au chapitre précédent (voir page 132).

Gestion de l'état de départ avant la simulation Dans l'état actuel des choses le simulateur proposé ne supporte pas la mise en place d'un état initial pré-simulation. Il s'agit d'un problème relativement complexe. Une discussion et des pistes de recherche sont néanmoins données un peu plus loin dans ce chapitre.

OpenFlash est en développement depuis mars 2013. Il s'agit d'un outil développé en C++. Les concepts de *classes* et *objets* introduits par la couche *orientée-objet* de C++ apportent une facilité non négligeable pour décrire des systèmes de stockage à base de mémoire flash. De plus, des concepts comme l'*héritage* et le *polymorphisme* sont très utiles à l'outil proposé dans le sens où l'on souhaite pouvoir permettre à un utilisateur de développer dans l'outil de nouvelles couches de gestion flash. Pour ce qui est du nombre de fichiers et de lignes de code[1], on présente ci-dessous la sortie de l'outil *cloc* (*count lines of code*) appliqué au répertoire des sources du simulateur :

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                             67           1807           1379           6926
C/C++ Header                    73            807            600           2347
CMake                            1             22             35             16
-------------------------------------------------------------------------------
SUM:                           141           2636           2014           9289
-------------------------------------------------------------------------------
```

1.3 Présentation générale d'OpenFlash

Dans Jain (2008), l'auteur définit les 4 principaux types de simulations utiles dans le cadre de l'étude d'un système informatique : l'*émulation*, la simulation *Monte-Carlo*, la simulation *à base de trace* et la simulation *à évènements discrets*. L'émulation est inadaptée à notre travail car elle consiste à décrire un système matériel dans les moindres détails. La simulation Monte-Carlo est utilisée pour décrire un système qui n'évolue pas en fonction du temps, elle est donc également inadaptée.

Le simulateur présenté ici est *à base de trace* : en effet, suivant les règles posées par la description du modèle de charge d'E/S (présenté au chapitre précédent), la série de requêtes d'E/S appliquées lors d'une simulation du système de stockage modélisé est représentée comme une suite d'évènements ordonnés dans le temps. De plus, le simulateur est *à évènements discrets*. Ce terme s'oppose à la simulation continue dans laquelle le simulateur examine le système modélisé de manière régulière à chaque pas de temps, pour déterminer si l'état du système doit changer ou pas : c'est par exemple le cas pour les simulateurs de modèles de composants matériels décrits en VHDL, dits *cycle-accurate*. Le simulateur présenté ici est dit à évènements discrets car c'est l'arrivée des différents évènements représentés dans la trace qui déclenche leur traitement. Si un temps important est présent entre deux évènements de la trace, cela n'impacte en rien le temps d'exécution de la simulation car OpenFlash traite les évènements les uns après les autres indépendamment du temps les séparant. En d'autres termes, une simulation lancée via le simulateur est *cadencée par la trace*.

1. Il n'y a pas de code généré automatiquement dans les sources du simulateur

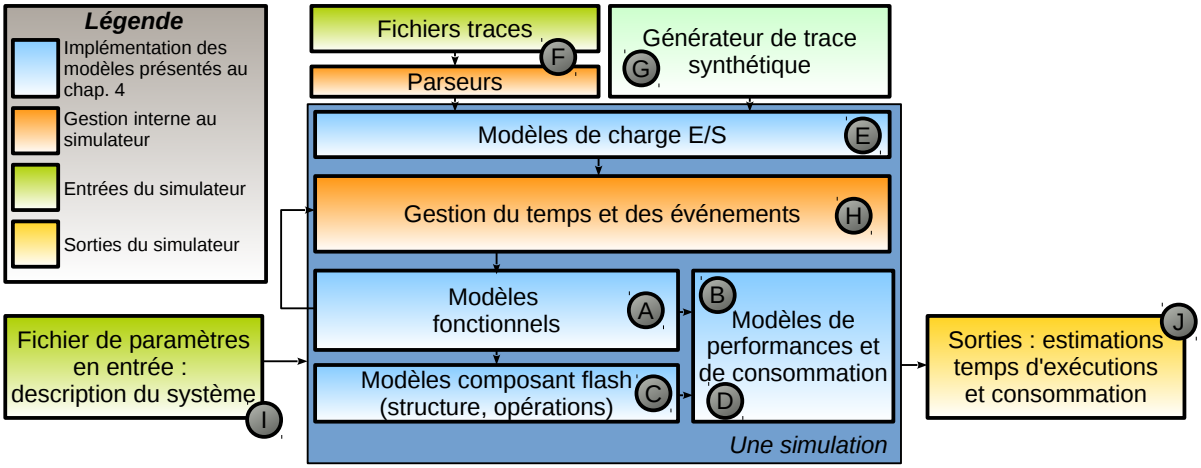


Figure 5.1 – Différents éléments mis en œuvre lors de l'exécution d'une simulation grâce à l'outil présenté ici.

Lors de l'exécution d'une simulation, le principe général de fonctionnement de l'outil est le suivant : les *évènements* traités par le simulateur sont les requêtes applicatives soumises au système de stockage simulé, c'est à dire les appels à *vfs_read()*, *vfs_write()* et les autres appels systèmes gérés (pour les systèmes FFS), ou les requêtes niveau bloc *read*, *write* et *TRIM* (pour les systèmes FTL). Le temps et la consommation sont calculés à la granularité d'un évènement, même s'il est possible d'obtenir des informations concernant les occurrences des sous-opérations composant l'évènement. Le traitement d'un évènement consiste en l'appel de la fonction correspondante au niveau de l'implémentation des modèles fonctionnels. Les modèles sont alors exécutés pour déterminer (1) le temps d'exécution de l'évènement, (2) l'énergie consommée due à l'évènement, et (3) la manière dont l'évènement modifie l'état du système simulé.

La figure 5.1 représente les différents éléments du simulateur mis en œuvre lors de l'exécution d'une simulation. Ces éléments sont présentés rapidement ici, et leur implémentation est décrite en détail dans les deux sections suivantes.

On retrouve au cœur d'OpenFlash l'implémentation des modèles présentés au chapitre précédent : les modèles fonctionnels (couche de gestion), de performances et de consommation, ainsi que les modèles propres au composant matériel flash (A, B, C et D sur la figure 5.1). Pour rappel, les interactions entre ces modèles ont été détaillées au chapitre précédent.

On retrouve également les modèles de charge d'E/S (E). La série d'évènements représentant la charge est décrite dans un fichier de trace passé en entrée du simulateur et parsée lors de la simulation (F). Le simulateur est également écrit de manière à pouvoir développer facilement un générateur de trace synthétique (G, encore non développé), qui permet de produire une charge d'E/S à partir d'une description générale du comportement de cette charge.

Le simulateur comporte une horloge qui permet de gérer le temps, ainsi qu'un ordonanceur permettant de diriger les évènements d'E/S vers la couche de gestion (H). Ces évènements doivent être ordonnancés car ils proviennent de deux sources : (1) la trace en entrée et (2) la couche de gestion qui insère des évènements additionnels dits *asynchrones* (par rapport au fait que c'est la trace qui cadence la simulation). Par exemple, l'occurence d'une passe de ramasse-miettes asynchrone n'est pas présente dans la trace applicative, mais elle est calculée par la couche de gestion et insérée dans la liste d'évènements à traiter.

Outre la trace, le simulateur prend en entrée un fichier de paramètres (I) permettant de décrire le système à simuler : il s'agit des paramètres des modèles, et de multiples options relatives au simulateur lui-même. En fin de simulation les résultats sont disponibles en sortie (J).

Les deux sections suivantes décrivent l'implémentation des composants cités ci-dessus. La section 2 décrit l'implémentation des composants pour la gestion interne du simulateur ainsi que les entrées

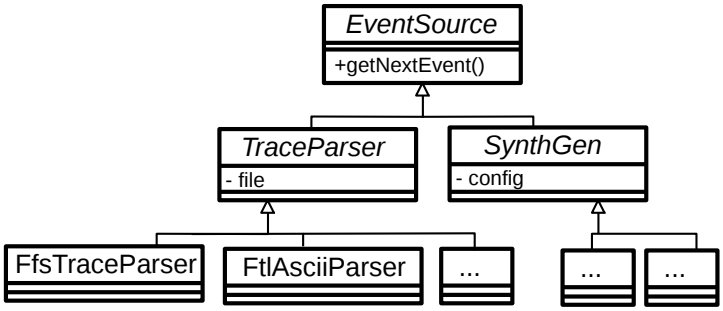


Figure 5.2 – Schéma partiel des classes implémentant la gestion de la trace en entrée du simulateur

et sorties du simulateur. La section 3 décrit l'implémentation des modèles présentés au chapitre précédent : modèles fonctionnels, de performances et de consommation, et modèles propres au composant flash.

2 OpenFlash : gestion interne, entrées et sorties

2.1 Modèle de charge et traitement de la trace en entrée

On a vu au chapitre précédent que les modèles de charge d'E/S pour la simulation de systèmes à base de FFS étaient différents de ceux utilisés lors de la simulation de systèmes de type FTL. Au niveau de la gestion de la trace en entrée, on utilise les concepts d'héritage et de polymorphisme offerts par C++ pour gérer ces différences de la manière la plus transparente possible pour l'utilisateur. La figure 5.2 illustre les classes impliquées dans la gestion de la trace en entrée. A noter que les schémas de classes présentés dans ce chapitre ne représentent pas de l'UML strict et que de nombreux attributs et méthodes présents dans le code sont omis sur les schémas pour des raisons de lisibilité.

Une simulation instancie de manière globale (singleton) un objet de type abstrait *EventSource* qui représente la source d'évènements pour la simulation. La méthode la plus importante de cette classe est *getNextEvent( )*. Il s'agit d'une méthode abstraite qui doit être implémentée dans les classes dérivées. Cette fonction est appelée par l'ordonnanceur, à chaque fois que le traitement d'un évènement est terminé et qu'il faut passer au suivant.

Une source d'évènements peut être soit un fichier contenant la trace (*TraceParser*), soit un générateur synthétique (*SynthGen*). Il s'agit également de types abstraits. Il n'existe actuellement pas d'implémentation de générateur synthétique dans le simulateur ; seul le support de fichiers de traces est proposé. Deux classes concrètes sont présentes. Elles correspondent aux deux formats de trace présentés au chapitre précédent : le format du niveau *appels systèmes* pour la simulation de systèmes de type FFS (*FfsTraceParser*) et le format du niveau bloc pour les systèmes de types FTL (*FtlAsciiParser*).

L'avantage d'utiliser une hiérarchie d'héritage telle que présentée ici est qu'il est très facile d'ajouter un nouveau format de trace sous forme de fichier, ou d'écrire un ou plusieurs générateur(s) synthétique(s).

2.2 Gestion du temps et des évènements

a) Gestion du temps

Le temps dans OpenFlash est représenté par une variable globale de type flottant. Sa valeur est incrémentée en fonction (A) des temps d'arrivée des requêtes (évènements) composant la trace ou insérées par la couche de gestion et (B) du temps d'exécution de ces évènements.

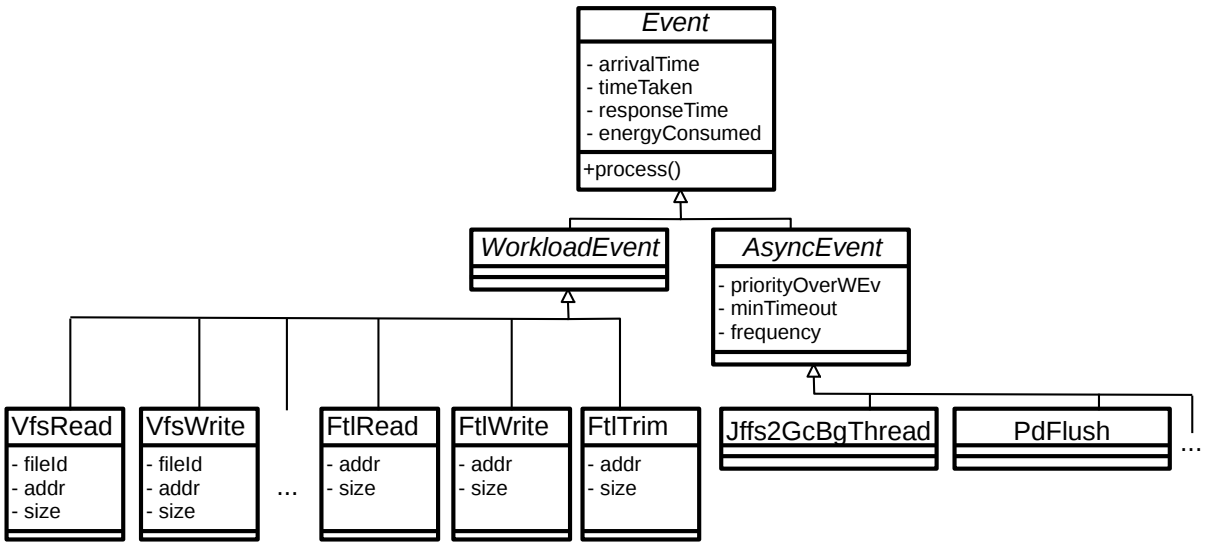


Figure 5.3 – Schéma partiel des classes représentants les différents types d'évènements abstraits et concrets utilisés dans le simulateur.

b) Types d'évènements

La hiérarchie de classe gérant les évènements est illustrée sur la figure 5.3. Tout évènement (classe *Event*) possède lors de sa création un temps d'arrivée. Deux autres variables d'instance permettent de stocker le temps d'exécution ou temps de traitement (*timeTaken*), le temps de réponse (*responseTime*) et l'énergie (*energyConsumed*) consommée par l'évènement une fois traité par le simulateur. Le temps de traitement correspond au temps d'exécution de l'évènement à partir du début de son traitement par le système de stockage simulé. Le temps de réponse est défini comme le temps entre l'arrivée de l'évènement dans le système et la fin de son traitement. Il est potentiellement plus long que le temps de traitement lorsque le début du traitement d'un évènement est postérieur à sa date d'arrivée dans le système, par exemple lorsque l'évènement précédent est toujours en cours de traitement. Tout évènement concret implémente la méthode *process()*, qui décrit l'exécution de l'évènement dans le simulateur. Il s'agit d'un appel aux modèles fonctionnels. Comme énoncé précédemment, un évènement peut être une requête applicative provenant de la trace (*WorkloadEvent*) ou un évènement inséré par la couche de gestion (*AsyncEvent*).

Comme indiqué sur la figure 5.3, les évènements provenant de la trace correspondent aux appels systèmes relatifs au stockage (mode FFS) comme *vfs_read()*, *vfs_write()* et les autres appels systèmes modélisés, ou encore les requêtes de type bloc (mode FTL). Chacun de ces évènements possède des attributs qui lui sont propres, par exemple un évènement de type *VfsRead* possède comme attribut un identifiant du fichier cible, une adresse cible et une taille de données à lire. L'implémentation de la fonction *process()* pour cet évènement consiste à appeler le modèle fonctionnel de *vfs_read()* en lui passant les paramètres correspondants. Le modèle fonctionnel calcule alors la manière avec laquelle l'évènement modifie le système et fait appel aux modèles de performances et de consommation pour renseigner les attributs *timeTaken* et *energyConsumed*.

Les évènements insérés par la couche de gestion, dits *asynchrones*, correspondent à des évènements calculés par cette couche de gestion et non présents dans la trace. Les évènements asynchrones possèdent des attributs permettant de les représenter de manière fine. L'attribut *priorityOverWEv* (pour *priority Over Workload EVent*) est un booléen qui, si activé, permet de spécifier que l'évènement asynchrone est prioritaire au moment de son insertion dans le système sur tout évènement en attente d'ordonnancement provenant de la trace. L'attribut *minTimeout* permet de spécifier un temps d'inactivité minimum après lequel l'évènement asynchrone peut s'exécuter. Finalement, l'attribut *frequency*

permet de définir un évènement asynchrone périodique : une fois traité par le simulateur, l'évènement est automatiquement ré-inséré dans le système à une date future égale au temps actuel plus la période de l'évènement. Actuellement, le simulateur implémente les évènements correspondant au thread du ramasse-miettes JFFS2, et au démon *pdflush* de Linux. *pdflush* déclenche avec JFFS2 une éviction (*flush*) régulière du write buffer (par défaut toutes les 5 secondes) pour éviter la perte de données en cas de démontage brutal. Plus d'informations sur l'implémentation du thread ramasse-miettes sont données dans la sous-section d) ci-dessous.

c) Ordonnancement des évènements

La boucle principale du simulateur traite séquentiellement les évènements provenant de la trace les uns après les autres, comme indiqué dans l'algorithme 19 ci-dessous. Les évènements provenant de la trace sont récupérés un par un (ligne 2) et passés en paramètres à la fonction *ProcessWorkloadEvent()* (ligne 3) qui correspond au cœur de l'ordonnanceur. Les évènement provenant de la trace étant traités de manière séquentielle, on peut dire que la politique d'ordonnancement est *premier arrivé, premier servi* (*first-come, first-served*).

Algorithme 19 : Boucle principale du simulateur

```
   données : eventSource : source d'évènement comme décrite en section précédente, scheduler : ordonnanceur du
             simulateur
1  tant que il reste des événements provenant de la trace à traiter faire
2  |  WorkloadEvent e = eventSource.getNextWorkloadEvent();
3  |  scheduler.processWorkloadEvent(e);
4  fin
```

On rappelle qu'un simulateur à évènements discrets est cadencé par l'arrivée des requêtes d'E/S contenues dans la trace. Le cœur de l'ordonnancement est ainsi réalisé dans la méthode *processWorkloadEvent()*, illustrée par l'algorithme 20. Le principe de base est le suivant : l'ordonnanceur contient une liste d'évènements asynchrones à exécuter. A chaque fois qu'un évènement synchrone (venant de la trace applicative) doit être exécuté, l'ordonnanceur compare le temps actuel (qui correspond à la date de fin de traitement du dernier évènement synchrone) et le compare avec le temps d'arrivée de l'évènement synchrone courant. Cela donne le temps libre entre deux requêtes applicatives pour potentiellement exécuter un ou plusieurs évènements asynchrones. Une fois les évènements asynchrones traités, l'évènement synchrone est lui-même traité, et le simulateur passe à l'évènement synchrone suivant avec un nouvel appel à la méthode *processWorkloadEvent()*. Le calcul du temps libre est réalisé à la ligne 2 de l'algorithme 20. À la ligne 4, un appel à une fonction auxiliaire *getNextAsyncEvent()* est réalisé pour déterminer le prochain évènement asynchrone à traiter en fonction du temps libre. Si aucun évènement asynchrone ne peut être traité (ou qu'il n'y a pas d'évènement asynchrone), le simulateur passe au traitement de l'évènement synchrone (lignes 5 à 7 et 23). Si un évènement asynchrone peut être traité, le simulateur définit le temps global à la date d'arrivée de cet évènement, et décrémente le temps libre en fonction de cette date d'arrivée (lignes 8 à 11). À la ligne 13, l'évènement asynchrone est exécuté. Son temps d'exécution et l'énergie consommée sont calculés via la fonction *process()*, et tout impact sur l'état du système est pris en compte. Le temps global et le temps libre sont alors mis à jour (lignes 14 et 15) en fonction du temps d'exécution de l'évènement. Si l'évènement en question est périodique, il est ré-inséré dans la liste d'évènements asynchrones à traiter (lignes 16 à 18). Lorsqu'il n'y a plus d'évènement asynchrone arrivé avant l'évènement synchrone courant, ce dernier est traité. Le temps global est mis à jour en fonction de son temps d'arrivée (lignes 20 à 22). L'évènement est exécuté (ligne 23) et le temps global est mis à jour en fonction du temps d'exécution de l'évènement (ligne 24).

Depuis la couche de gestion, il est possible d'ajouter des évènements asynchrones dans la liste

Algorithme 20 : Cœur de l'ordonancement : fonction *processWorkloadEvent*

```
1  Fonction processWorkloadEvent(we)
       entrées  : we : évènement à traiter provenant de la trace, de type WorkloadEvent
       données : time variable globale représentant le temps dans le simulateur
2      FreeTime = we.getArrivalTime() - time;
3      tant que vrai faire
4          ae = getNextAsyncEvent(FreeTime); // prochain évènement asynchrone
5          si ae == null alors
6              break ;
7          fin
8          si ae.getArrivalTime() > time alors
9              FreeTime = FreeTime - ae.getArrivalTime();
10             time = ae.getArrivalTime(); // mise à jour du temps global
11         fin
12         time = time + ae.getMinTimeOut();
13         ae.process(); // calcul du temps d'exécution et de l'énergie consommée
14         FreeTime = FreeTime - ae.getExecutionTime();
15         time = time + ae.getExecutionTime();
16         // mise à jour du temps global
17         si ae est périodique alors
18             ré-insérer ae dans la liste des évènements aynchrone, avec un temps d'arrivée égal à time +
               ae.getFrequency();
19         fin
20     fin
21     si we.getArrivalTime() > time alors
22         time = we.getArrivalTime(); // mise à jour du temps global
23     fin
24     we.process(); // calcul du temps d'exécution et de l'énergie consommée
25     time = time + we.getExecutionTime();
```

d'évènements asynchrones à traiter au niveau de l'ordonnanceur. C'est ainsi qu'est implémenté le comportement du thread ramasse-miettes de JFFS2 décrit dans la sous-section suivante.

d) Implémentation du thread ramasse-miettes asynchrone de JFFS2

A chaque réveil du thread ramasse-miettes JFFS2, une passe de ramasse-miettes est effectuée. Dans le noyau, le cœur de l'implémentation de ce thread correspond à une boucle infinie qui effectue le travail décrit, grossièrement, en pseudo-code dans l'algorithme 21.

Algorithme 21 : Représentation simplifiée de l'exécution du thread ramasse-miettes de JFFS2

```
1  tant que vrai faire
2      schedule();
3      si un signal a été reçu alors
4          Lancer une passe de ramasse-miettes;
5      fin
6  fin
```

La fonction *schedule()*, appelée de cette manière, correspond à une mise en sommeil du thread (ce dernier rend volontairement la main sur le processeur). Ce thread noyau n'est donc pas réveillé périodiquement, mais s'exécute de manière sporadique : son exécution est conditionnée à la réception d'un signal. Le signal est envoyé depuis le code de JFFS2. A chaque fois qu'une nouvelle node JFFS2 doit être écrite, comme vu dans les chapitres précédents, une réservation d'espace flash est réalisée. Au moment de cette réservation, JFFS2 vérifie les niveaux d'espaces libres et invalides. Si ces niveaux

atteignent certains seuils, définis, entre autres, par rapport à la taille de la partition, le signal est envoyé et une passe de ramasse-miettes est effectuée au prochain réveil du thread noyau. Il faut généralement plusieurs passes, donc plusieurs réveils du thread, pour recycler un ou plusieurs blocs et faire repasser les seuils concernés à un niveau ne déclenchant plus d'appel au ramasse-miettes en arrière plan. Il faut, de plus, savoir que ce thread possède une priorité relativement faible. Ainsi, il se passe un temps non-prédictible entre l'envoi du signal et l'exécution effective de la passe de ramasse-miettes correspondante. La même réflexion peut être faite concernant le temps entre deux passes du ramasse-miettes asynchrone. par conséquent, les occurrences d'exécution des passes de ramasse-miettes dues au thread noyau JFFS2 sont relativement difficiles à prévoir et donc à modéliser. Il n'est notamment pas possible de les représenter par un évènement purement périodique.

Après l'exécution d'une suite d'expériences déclenchant des appels au ramasse-miettes en arrière plan, et des mesures de temps via Funcmon, on peut constater que le temps entre deux réveils du thread ramasse-miettes dépend (1) des caractéristiques de la charge d'E/S, en particulier les temps d'inter-arrivées ; et (2) de la charge du système d'exploitation (nombre de tâches exécutées sur le CPU). Quantifier l'impact de ces deux éléments sur le temps entre deux réveil du ramasse-miettes en arrière plan est complexe. On considère le cas d'une charge d'E/S avec des temps d'inter-arrivées nuls (E/S arrivant aussi vite que possible) et une charge système minimale. Dans ce cas on constate que la distribution des temps entre deux appels du thread ramasse-miettes s'apparente à une distribution exponentielle.

Le thread ramasse-miettes est donc modélisé dans le simulateur de manière suivante : au niveau de l'implémentation du modèle fonctionnel de JFFS2, lors de la réservation d'espace flash les seuils d'espaces libres et invalides sont vérifiés tout comme dans l'implémentation réelle de JFFS2. S'il s'avère qu'un réveil du thread ramasse-miettes est nécessaire, ce réveil est représenté par l'insertion d'un évènement asynchrone (sporadique) dans la liste maintenue au niveau de l'ordonnanceur. Au prochain temps de pause entre les E/S provenant de la charge applicative, une passe de ramasse-miettes sera effectuée. A la fin de cette passe, si les seuils nécessitent toujours l'intervention du ramasse-miettes, un nouvel évènement asynchrone est inséré à une date dans le futur. Cette date est calculé à partir de la génération d'une valeur aléatoire basée sur une distribution exponentielle, sommée au temps actuel.

Il s'agit là d'une modélisation assez grossière. Pour décrire de manière fine les temps de réveil du thread ramasse-miettes, il faudrais quantifier précisément l'impact de la charge système sur ces valeurs, par exemple observer l'évolution du temps entre les réveils du ramasse-miettes en arrière plan en fonction du pourcentage d'occupation CPU. Il faudrait alors intégrer la notion d'occupation CPU au simulateur.

e) Quelques mots sur *pdflush*

Le démon *pdflush* est un processus noyau dont la responsabilité est d'évincer des données *dirty* depuis le page cache vers le média de stockage, lors de l'utilisation du mécanisme de *write-back* du page cache. Comme énoncé précédemment, JFFS2 ne fait pas usage de ce mécanisme. Néanmoins, *pdflush* a un léger impact sur le comportement de JFFS2 : lors du réveil de pdflush : le write buffer de JFFS2 est écrit en flash, même si ce dernier n'est pas plein.

pdflush se réveille de manière périodique : par défaut cet intervalle est fixé à 5 secondes. Sous Linux, cette valeur peut être consultée ou modifiée en lisant ou en écrivant le fichier suivant :

```
/proc/sys/vm/dirty_writeback_centisecs
```

pdflush est modélisé dans le simulateur sous la forme d'un évènement asynchrone périodique, dont la période est configurable et est fixée par défaut à 5 secondes. Ce démon est activé automatiquement lors d'une simulation de type FFS.

2.3 Entrée de paramètres et fichier de configuration

OpenFlash prend deux entrées : (A) la trace, décrite précédemment, et (B) un fichier de paramètres contenant :

— Les paramètres des multiples modèles implémentés dans l'outil ;

— Des options relatives au simulateur lui-même.

La librairie *libconfig* (Lindner, 2012) est utilisée pour traiter le fichier de paramètres. Cette librairie définit un format structuré pour représenter des ensembles relativement simples de données.

Les paramètres sont structurés en blocs. Ci-dessous on peut voir un exemple du bloc correspondant aux paramètres du modèle structurel du composant mémoire flash (bloc `flash_layer`) et un sous ensemble du bloc correspondant à la couche fonctionnelle (`functional_model`) :

```
flash_layer =
{
  pages_per_block = 64;
  blocks_per_plane = 800;
  planes_per_lun = 1;
  luns_per_channel = 1;
  channels = 1;

  page_size_bytes = 2048;
  oob_size_bytes = 64;
  channel_width_bits = 8;
}

functional_model =
{
  functional_mode = "ffs"; // Mode de simulation - ffs ou ftl
  ffs =
  {
    type = "JFFS2";
    jffs2 =
    {
      check_after_erase = false; // options de fonctionnement de JFFS2
      read_check = false;

      base_readpage_timing_overhead = 30.0; // overheads niveau FFS
      base_write_end_timing_overhead = 50.0;
      base_write_begin_timing_overhead = 5.7;

      bg_thread = true; // options relatives au thread ramasse-miettes
      bg_thread_inter_arrival_exponential_rate = 0.00005685356;
      bg_thread_inter_arrival_exponential_base = 10000;
    }
  }

  vfs =
  {
    /* etc. */
  }
```

```
  /* etc. */
}
```

L'intégralité des paramètres des modèles présentés au chapitre précédent est présente dans le fichier de paramètres. On peut voir ci-dessus les paramètres du modèle structurel flash. Les paramètres des modèles de performances et de consommation sont également présents (valeurs de temps des accès flash, et overheads à tous les niveaux). On peut aussi retrouver des options relatives aux couches de gestion : pour VFS, la taille fixée du page cache, la fréquence d'exécution de *pdflush*, l'activation ou la désactivation de read-ahead ; pour JFFS2, l'activation ou la désactivation du thread ramasse-miettes, des paramètres relatifs à sa fréquence d'exécution, etc.

D'autres paramètres sont également présents comme la valeur d'une graine pour toutes les générations de nombre aléatoires réalisés au sein du simulateur (pour la reproduction à l'identique d'une simulation), le mode d'utilisation (FTL ou FFS), le type de trace en entrée et le chemin vers le fichier correspondant. Tous les paramètres étant présents dans le fichier décrit ici, au lancement le simulateur prend uniquement en argument le chemin vers le fichier de paramètres :

```
$ ./OpenFlash fichier_parametres.conf
```

2.4 Gestion des erreurs

OpenFlash comporte un système simple de gestion des erreurs. On définit une *erreur* comme étant un problème empêchant le déroulement correct d'une simulation (par exemple le non respect d'une contrainte forte relative à une commande flash qui indique un problème d'implémentation de la couche fonctionnelle). Lors d'une erreur, la simulation s'arrête immédiatement. En cours de développement, par exemple lors du prototypage d'un système de gestion flash, le simulateur exécute l'instruction `assert(0)`. Cela permet lors de l'utilisation d'un débogueur comme *gdb* d'indiquer un problème tout en maintenant le contexte d'exécution du programme pour inspecter les variables et méthodes de l'environnement. En cours d'utilisation (non-développement), par exemple lors du lancement de simulations comparatives sur plusieurs systèmes de gestion, le programme se contente de quitter en indiquant le problème, tout en libérant la mémoire allouée.

On définit également la notion d'*avertissement*, qui indique un problème potentiel mais ne perturbant pas *a priori* le déroulement de la simulation. Le logiciel se contente d'indiquer un message sur la sortie d'erreur et poursuit son exécution.

Les notions d'erreurs et d'avertissement, couplées aux modèles représentant les contraintes des opérations flash supportées par le simulateur, sont une aide pour l'intégration de couches de gestion flash existantes, ou le prototypage de nouvelles couches de gestion.

2.5 Sorties d'OpenFlash

Les estimations d'énergie et de performances pour une simulation de type FFS sont présentées sous forme de fichiers disponibles en fin de simulation. Un fichier est créé par niveau modélisé (VFS, FFS, pilote NAND MTD, et mémoire flash). Chaque fichier contient la liste des évènements calculés au niveau correspondant, classés dans l'ordre temporel. Pour chaque évènement le temps d'exécution et l'énergie consommée sont indiqués. Ces fichiers de résultats sont au format *CSV* (Comma Separated Values), ce qui les rend facilement traitables, par exemple pour une représentation sous la forme d'un graphique. Le fichier correspondant au niveau de granularité le plus haut (VFS) contient également des informations générales sur les résultats, comme par exemple l'énergie totale consommée durant la simulation, le temps d'exécution moyen par appel système, etc.

Ci-dessous, on présente un exemple d'une partie d'un fichier de résultats correspondant au niveau VFS. Ces résultats sont issus de l'exécution des modèles de la carte Omap, les valeurs d'énergie sont ainsi divisées entre le composant mémoire (RAM + flash) et le CPU.

```
# VFS Statistics
# ================
# ------------------------------
# Total energy at VFS level (J) :
# ------------------------------
#  - CPU : 1.08679
#  - Mem : 0.407379
# ----------------------------
# Mean energy per syscall (J) :
# ----------------------------
#  - CPU : 0.00338562
#  - Mem : 0.00126909
# --------------------------------
# Mean exec. time per syscall (s) :
# --------------------------------
# 0.00771392
# ----------------------------
# Total time at VFS level (s) :
# ----------------------------
# 2.47617
# -------------------------------------
# Detailed timing / energy per syscall:
# -------------------------------------
# syscall_type ; syscall_execution_time ;
#  syscall_energy_on_memory_chip ; syscall_energy_on_cpu
vfs_write;1.17353e+06;0.190588;0.508264
vfs_write;4393.5;0.000732665;0.00195409
vfs_write;3982;0.000662299;0.00176769
vfs_write;4393.5;0.000732665;0.00195409
vfs_write;4393.5;0.000732665;0.00195409
vfs_write;3982;0.000662299;0.00176769
vfs_write;3982;0.000662299;0.00176769
vfs_write;3570.5;0.000591932;0.00158128
# etc ...
```

Il est possible de spécifier dans le fichier de paramètres quels niveaux de résultats (VFS / FFS / MTD / flash) on souhaite obtenir en fin de simulation. Outre les estimations de temps d'exécution et d'énergie, un fichier de résultats contenant des informations sur les opérations flash est également disponible. Ce fichier contient en particulier le nombre d'opérations flash réalisées, ainsi que des informations concernant la répartition de l'usure : il s'agit d'une liste donnant pour chaque indice de bloc physique simulé le nombre d'effacements subis par le bloc correspondant.

2.6 Réflexion sur l'état initial avant simulation

L'état initial d'un système de stockage à base de mémoire flash, que ce soit en simulation ou en environnement réel (benchmarking), influence de manière potentiellement importante les résultats de simulation ou d'évaluation de performances (Bjørling et coll., 2010b; Smith, 2009; Olivier et coll., 2012a). Il est donc très important de définir avant une simulation, de manière précise, l'état initial du système.

Du point de vue de nos modèles, qui présentent une granularité de modélisation relativement fine pour les algorithmes de la couche de gestion ainsi que l'état des pages de la flash simulée, on propose

de définir l'état initial comme divisé entre les deux états suivants :

1. L'état de chacune des pages flash simulées, *libre* ou *occupée* ;
2. L'état de la couche de gestion :
   (a) Les méta-données indiquant la validité ou l'invalidité des données contenues dans les pages occupées ;
   (b) Les informations de traduction d'adresse (FTL) ou d'indexation des fichiers (FFS) ;
   (c) D'autres paramètres spécifiques au modèle de couche de gestion considéré.

La mise en place de l'état initial avant une simulation s'appelle la phase de *warm-up*. Dans un simulateur complexe, il ne suffit pas, pour définir un état initial, de prendre en compte uniquement la mémoire flash (par exemple en définissant une quantité données de pages dans l'état *occupé* avant la simulation). Il faut prendre également en considération la couche de gestion.

Au niveau des simulateurs existants, le warm-up est réalisé de différentes manières. Dans *Flashsim* (Gupta et coll., 2009; Kim et coll., 2009b), pendant la phase de warm-up, la trace en entrée est exécutée une première fois, chaque requête de cette trace étant traitée comme une requête d'écriture. Par la suite les statistiques du simulateur sont remises à 0 et la simulation est lancée : la trace est alors reexécutée. Il s'agit d'une méthode non-optimale car le fait de jouer la trace une fois avant la simulation ne permet absolument pas de spécifier un état initial de manière détaillée. De plus, il s'agit d'une opération qui double le temps d'exécution de la simulation, ce qui est gênant lors de simulation de traces de tailles importantes. Dans *SsdSim* (Hu et coll., 2011), une quantité donnée d'espace invalide est créé en flash avant la simulation, et les méta-données de l'unique modèle de FTL supporté (traduction par page) sont mises à jour avec ces informations. La technique de warm-up est donc fortement couplée à l'algorithme de couche de gestion utilisé, et peu efficace dans le sens ou il s'agit uniquement d'espace invalide. On peut néanmoins noter que dans ce cas le temps de warm-up ne dépend plus de la taille de la trace.

Dans l'état actuel des choses, le simulateur proposé ici ne supporte pas la mise en place d'un état initial. Néanmoins, on envisage d'implémenter la phase de warm-up de la manière décrite ci-dessous.

Comme le prouve l'implémentation de Hu et coll. (2011), la méthode de warm-up dépend totalement de la couche de gestion utilisée. On considère donc que le warm-up doit être réalisé au niveau de la couche de gestion elle-même. La couche de gestion a une visibilité directe sur le composant mémoire flash et peut donc modifier l'état des pages. De plus la couche de gestion a, bien entendu, une visibilité sur ses propres méta-données ce qui lui permet de les modifier.

Cette phase de warm-up doit être aussi rapide que possible, ainsi il serait intéressant que l'état initial soit "chargé" automatiquement de manière indépendante de la trace. Cela est notamment utile lors de l'utilisation de générateurs de trace synthétiques, avec lesquels la trace n'est parfois pas connue avant le lancement de la simulation elle-même. On propose d'ajouter aux classes abstraites représentant un FFS ou une FTL une méthode pour effectuer le warm-up. Cette méthode est elle-même abstraite et doit être redéfinie au niveau de toute couche de gestion concrète implémentée au sein du simulateur. Elle prend en entrée des paramètres venant de l'utilisateur, représentant les quantités d'espace flash (A) libre et occupé, (B) valide et occupé, ou (C) invalide désirées en début de simulation. On peut également imaginer d'autres paramètres spécifiques à des modèles de couches de gestion donnés, ainsi que des méthodes de chargement automatique d'états de départ pré-définis.

2.7 Réutilisabilité du simulateur, des modèles et des méthodes d'extraction de paramètres

Une grande partie des modèles actuellement implémentés dans le simulateur est ré-utilisable dans le cas ou l'on souhaiterais y compléter l'outil avec des modèles relatifs à un FFS existant (par exemple YAFFS2 ou UBIFS), ou encore y prototyper une nouvelle couche de gestion, FFS ou FTL.

VFS et le pilote NAND MTD sont des couches génériques sous Linux : elles sont utilisées par tous les FFS supportés par cet OS. Ainsi les modèles relatifs à ces couches sont ré-utilisables directement. Il en est de même pour les modèles relatifs à la mémoire flash. Que le FFS considéré soit un FFS existant ou un prototype pour une nouvelle proposition, le travail d'un utilisateur souhaitant implémenter un FFS dans le simulateur peut être divisé en plusieurs étapes :

1. Il faut construire une série de modèles fonctionnels représentant :
   — la manière dont les accès flash sont déterminés en fonction de la charge d'E/S appliquée par VFS : il s'agit des types d'opérations réalisées et des pages ou blocs ciblés par ces opérations. C'est l'équivalent du découpage en nodes et de la stratégie d'écriture pour notre cas d'étude JFFS2 ;
   — l'éventuelle présence de caches de RAM et leur fonctionnement, qui peut fortement impacter le traitement d'une charge d'E/S donnée, et les performances / la consommation associées. Pour JFFS2 il s'agit par exemple du buffer d'écriture.
2. La présence de mécanismes spécifiques identifiés dans la première phase peut nécessiter la définition de nouveaux modèles de performances et de consommation, relatifs auxdits mécanismes ;
3. Il est nécessaire d'effectuer une nouvelle phase d'extraction de paramètres :
   — L'interfaçage avec VFS imposant la définition par un FFS de fonctions d'accès standardisées (par exemple *xxx_readpage()* ou encore *xxx_write_end()*, *xxx* étant le nom du FFS), les méthodes d'extraction de paramètres présentées dans ce chapitre sont dans le cadre de ces fonctions directement réutilisables ;
   — Si de nouveaux modèles de performances / consommation spécifique au FFS considéré on été développés, alors il est nécessaire de définir les méthodes d'extraction de paramètres concernants ces noveaux modèles.

Dans le code du simulateur, la partie où il faut intervenir lors de l'implémentation des modèles relatif à un nouveau FFS est isolée des couches supérieures et inférieures (VFS et pilote NAND), ce qui facilite grandement le développement et réduit les risques d'erreur.

Concernant la partie FTL, la partie des modèles représentant le buffer de mémoire vive (voir au chapitre précédent la figure 4.10 page 159) est, dans le simulateur, découplée des modèles fonctionnels de la FTL elle même. Ainsi, lorsque le simulateur implémentera plusieurs algorithmes de FTL et de gestion de buffer de mémoire vive, il sera possible d'utiliser chaque algorithme de buffer avec chaque algorithme de FTL.

De manière générale, à moyen terme, on souhaite que le simulateur puisse implémenter une bibliothèque d'algorithmes de FFS, FTL, et de modèles de performances / consommations associés. Lors de l'implémentation d'un nouveau mécanisme de gestion dans le simulateur, il devient alors possible de comparer les performances et la consommation de ce mécanisme avec les systèmes précédemment présents dans l'outil.

3 Intégration des modèles dans OpenFlash

3.1 Modèles fonctionnels

Les algorithmes des modèles fonctionnels sont organisés dans un ensemble de classes qui correspondent aux différents niveaux de modélisation fonctionnelle, comme présenté sur la figure 5.4. Dans une simulation il y a au niveau global un objet de type *Vfs* représentant le système de fichiers virtuel, un objet de type abstrait *FlashFileSystem* (par exemple un objet concret *Jffs2*) représentant le FFS, et un objet représentant le pilote NAND (MTD) de type *NandDriver*. Il existe également une série d'objets représentant les implémentations des modèles opérationnels et structurels correspondant au composant mémoire flash, qui seront décrits en section 3.3 plus loin.

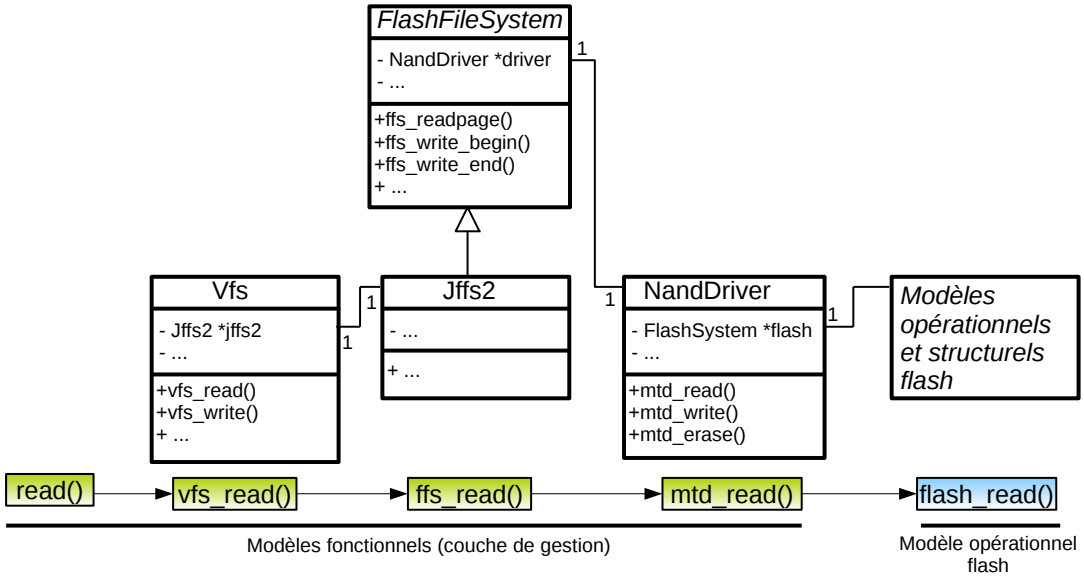


FIGURE 5.4 – Implémentation des modèles fonctionnel pour une couche de gestion de type FFS

Les objets composant la couche fonctionnelle sont inter-connectés pour représenter la pile de gestion des E/S sous Linux : VFS à une visibilité (c'est un pointeur) sur le FFS, qui voit le pilote, qui lui-même peut accéder à la représentation de la mémoire flash. Chaque classe implémente des méthodes correspondant aux différents algorithmes présentés au chapitre précédent : par exemple l'objet VFS implémente les méthodes *vfs_read()* et *vfs_write()*. Sur la figure 5.4 on présente la manière avec laquelle un appel *read()* de la trace est traité par la couche fonctionnelle : l'appel est réceptionné par la couche VFS (*vfs_read()*). Cette fonction itère sur les pages Linux à lire et exécute potentiellement une ou plusieurs passes de *read-ahead*. Lorsque des données doivent être lues depuis le FFS, l'objet VFS fait appel à *ffs_readpage()* pour chaque page Linux à lire. L'implémentation JFFS2 de cette fonction fait appel à *mtd_read()* pour effectuer des lectures de pages flash. L'objet pilote fait lui-même appel aux fonctions du modèle opérationnel flash pour lancer les opérations de lecture de pages.

L'intégration de la couche FFS sous forme d'une classe abstraite, dont il faut faire hériter tout type de système de fichiers concret, rend plus aisée la représentation et le prototypage de FFS existants et nouveaux au sein du simulateur. Une approche similaire est adoptée pour le modèle de couche fonctionnelle de type FTL.

3.2 Modèles de performances et de consommation (non flash)

Les modèles de performances et de consommation qui ne sont pas relatifs à la mémoire flash sont constitués des valeurs d'overhead présentées au chapitre précédent. Dans l'état actuel des choses ils sont implémentés au sein des méthodes correspondant au modèle fonctionnel. L'ensemble des méthodes du modèle fonctionnel impliquées dans le traitement d'un évènement provenant de la trace (un appel système dans le cas FFS) utilisent un objet destiné à contenir : (A) le temps d'exécution de l'appel système et (B) l'énergie consommée lors de l'exécution de l'appel système. Cet objet, de type *PpcVal* pour *performance and power consumption value* contient donc les champs suivants :

1. *timeTaken* : temps d'exécution ;
2. *eCPU* : énergie consommée sur la puce processeur ;
3. *eMem* : énergie consommée sur la puce mémoire (RAM + flash).

Cet objet est assez spécifique à la carte Omap car l'énergie consommée est subdivisée entre les puces CPU et mémoire (on rappelle qu'il s'agit des deux points de mesure de consommation considérés

sur cette plate-forme matérielle). Il est facilement adaptable (création d'un champ par point de mesure) à d'autres systèmes en fonction des points de mesures disponibles.

On rapelle que les modèles de performances et de consommation fonctionnent sur le principe suivant : a un niveau de modélisation donné (VFS, FFS, pilote), le temps d'exécution et l'énergie consommée sont égale à la somme entre (A) le temps / l'énergie du niveau inférieur et (B) un overhead correspondant au niveau courant. Des objets de type *PpcVal* sont déclarés dans l'implémentation des modèles correspondant à chaque niveau de modélisation. La méthode correspondant au traitement d'une E/S à un niveau donné incrémente les champs d'un objet *PpcVal* en fonction des temps d'exécutions et valeurs d'énergies consommées à ce niveau. Cet objet est alors renvoyé au niveau supérieur qui le somme avec un l'overhead relatif à son propre niveau. Par exemple, on a vu au chapitre précédent que le modèle fonctionnel pour *vfs_write()* itère sur chacune des pages Linux touchées par l'écriture. La manière dont le temps d'exécution et la consommation sont gérés à ce niveau est présentée sur l'algorithme 22.

Algorithme 22 : Intégration des modèles de consommation

```
1  PpcVal result;
2  pour toute page Linux p touchée par l'écriture faire
3  |   // Traitement fonctionnel niveau VFS ...
4  |   // Calcul temps et consommation du niveau inférieur (FFS)
5  |   result += ffs->ffs_write_begin(p);
6  |   result += ffs->ffs_write_end(p);
7  |   // Ajout des overhead pour le niveau courant (VFS)
8  |   result.timeTaken += T_Vfs_Write_Page_Overhead;
9  |   result.eCpu += E_Vfs_Write_Page_Overhead_CPU;
10 |   result.eMem += E_Vfs_Write_Page_Overhead_Mem;
11 fin
12 retourner result
```

Dans cet algorithme on peut voir qu'un objet de type *PpcVal* est déclaré (ligne 1) et renvoyé en fin de fonction. Cet objet contient le temps d'exécution et l'énergie consommée par l'appel système. Aux lignes 2 à 8 la fonction itère sur chacune des pages Linux touchées par l'écriture. Le modèle fonctionnel de niveau inférieur (fonctions *ffs_write_begin()* et *ffs_write_end()*) est appelé (lignes 4 et 5). Ces fonctions renvoient un objet de type *PpcVal*, qui correspond au temps d'exécution et à l'énergie consommée par le niveau FFS et les niveaux inférieurs (pilote et flash). L'opérateur C++ "+=" est surchargé pour permettre d'additionner facilement des valeurs de type *PpcVal* (il s'agit simplement de l'addition des trois composantes). Les résultats sont stockés dans l'objet *result* courant. Les overheads sont alors ajoutés lignes 6 à 8, dans la boucle car on rappelle que l'overhead correspondant à *vfs_write()* est un overhead par page Linux touchée. Suivant ce principe, l'objet *PpcVal* du niveau le plus haut (VFS) contient l'estimation de temps d'exécution et énergie consommée par l'appel système. Les objets *PpcVal* relatifs aux autres niveaux peuvent également être consultés pour obtenir des estimations aux niveaux inférieurs.

La manière dont ces modèles de performances et de consommation sont intégrés fait qu'ils sont actuellement fortement couplés avec l'implémentation fonctionnelle. Une évolution possible du simulateur serait de découpler les modèles de performances et de consommation dans des classes dédiées, possédant des interfaces auxquelles feraient appel le modèle fonctionnel lorsqu'il est nécessaire de calculer une quelconque valeur de temps d'exécution ou d'énergie.

3.3 Modèles relatifs au composant flash

Comme vu au chapitre précédent, le composant matériel puce flash a fait l'objet d'une modélisation fine. Dans cette sous-section, on présente l'implémentation de ces modèles dans le simulateur.

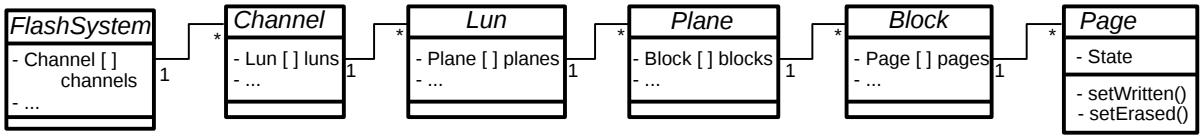


FIGURE 5.5 – Classes gérant le modèle structurel flash

a) Modèles structurels et opérationnels relatifs au composant flash

Modèles structurels flash La figure 5.5 représente les classes gérant le modèle structurel du composant flash : le type *FlashSystem* représente l'ensemble des puces flash présentes lors d'une simulation (éventuellement une seule). Il comporte un tableau d'éléments *Channel* qui représentent les canaux. Les canaux contiennent des LUN (*Lun*), contenant des plans (*Plane*), contennant des blocs (*Block*) qui eux mêmes contiennent des pages (*Page*). Une page possède un état (voir au chapitre précédent la figure 4.4 page 131). L'état de la page peut être modifié par le modèle opérationnel, en appelant les méthodes présentes au niveau d'une page : *setWritten* (change l'état de la page de *libre* en occupée) et *setErased* (passe l'état de la page à *libre*).

Pour ce qui est de l'adressage d'une entité dans une représentation telle que celle présentée ici, on utilise des tuples définis comme suit : (*indice de canal*, *indice de LUN dans le canal*, *indice de plan dans la LUN*, *indice de bloc dans le plan*, *indice de page dans le bloc*). Par exemple (0, 1, 2, 239, 62) représente la 63^{è} page du 240^{è} bloc contenu dans le 3^{è} plan de la 2^{è} LUN du premier canal (les indices démarrent à 0). Une classe C++ *Address* est définie suivant ce modèle.

Pour une simulation de type FFS on ne simule, en général, qu'une seul puce flash (c'est sur ce principe que sont basés la plupart des modèles présentés au chapitre précédent) : pour ce faire il suffit de décrire un système avec un unique canal et une seule LUN, et de décrire les niveaux inférieurs (nombre de plans, blocs, pages) en fonction de la puce à représenter.

Modèle opérationnel flash Le modèle opérationnel complexe présenté brièvement au chapitre précédent (et dépeint plus en détail en annexe) est implémenté dans le simulateur. S'il est vrai que les modèles concernant les FFS ne font appel qu'aux opérations flash de base (dites *legacy*), cela n'empêche pas la compatibilité avec le modèle complexe : il suffit de ne pas utiliser les opérations avancées proposées par ce dernier. C'est le cas de l'implémentation fonctionnelle du pilote MTD, qui ne fait appel qu'aux opérations legacy : lecture, écriture et effacement.

La figure 5.6 présente la hiérarchie de classes permettant la représentation des multiples commandes flash supportées dans le simulateur. Le type *FlashCommand* (A sur la figure 5.6) est un type abstrait. Tout type de commande pouvant être appliqué sur la représentation de(s) puce(s) flash simulée(s) hérite de *FlashCmd*. Ce type centralise un certain nombre d'attributs communs à tout type de commande flash. Il présente également des méthodes abstraites à être redéfinies dans les classes dérivées. Ces méthodes sont présentées ci-après.

Héritant directement de *FlashCommand*, on retrouve le type concret *MultiChannelCmd* (B), représentant une commande multi-canaux, et le type abstrait *UniChannelCmd* (C). Une commande de type *UniChannelCmd* s'applique sur un unique canal, une commande *MultiChannelCmd* contient donc une série de commandes *UniChannelCmd* représentant les différentes sous-commandes exécutées en parallèle, chacune sur un canal différent. Les sous-commandes composant une opération multi-canaux peuvent donc être de tout type sauf *MultiChannelCmd*. Héritant de *UniChannelCmd*, on retrouve le type concret *MultiLunCmd* (D) représentant une commande LUN-entrelacée, et le type abstrait *UniLunCmd* qui représente une commande s'appliquant sur une seule LUN. Les mêmes principes que pour les opérations multi-canaux sont appliqués.

Héritant du type *UniLunCmd* on retrouve une série de types concrets représentant le reste des commandes flash : les commandes legacy (*LegacyRead*, *LegacyWrite* et *LegacyErase*), en mode cache

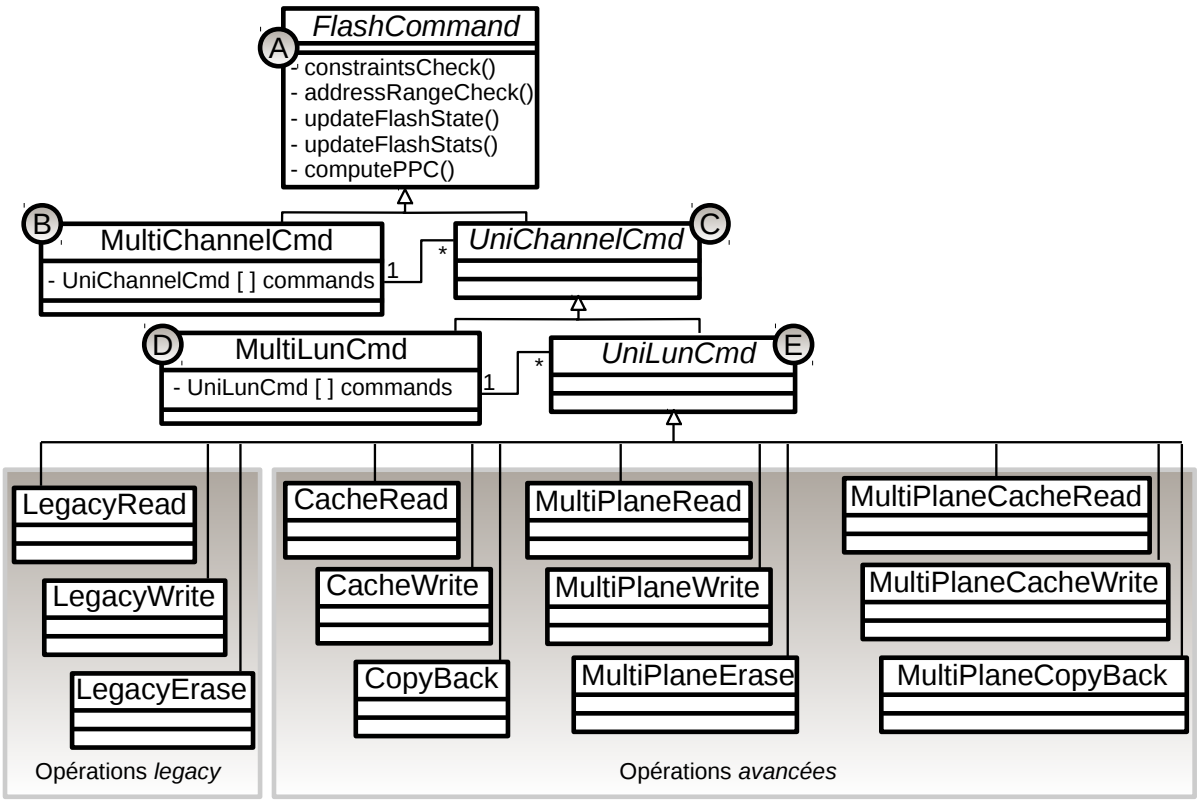


Figure 5.6 – Hiérarchie de classes pour la gestion du modèle opérationnel du composant flash

(*CacheRead* et *CacheWrite*), copy-back (*CopyBack*), et multi-plans (*MultiPlaneRead*, *MultiPlaneWrite*, etc.).

Comme mentionné plus tôt, la classe abstraite *FlashCommand* possède 5 méthodes abstraites qui doivent être redéfinies dans toute classe concrète dérivée. Ces méthodes permettent de définir les contraintes relatives à l'opération, la manière avec laquelle elle modifie l'état du système lors de son exécution, et la manière dont sont calculées les performances et la consommation relatives à cette commande. Comme on peut le voir sur la figure 5.6, ces méthodes sont les suivantes :

1. *addressRangeCheck* est une méthode qui vérifie que l'intégralité des composants (canaux / LUNs / plans / blocs / pages) adressés sont bien situés dans l'espace d'adressage flash simulé. Par exemple pour une commande de type *LegacyWrite*, cette méthode vérifie que l'indice du canal adressé est inférieur au nombre de canaux simulés, idem pour la LUN, le plan, le bloc et la page ;
2. *constraintsCheck* vérifie l'intégralité des contraintes relatives à la commande. Dans notre cas d'exemple avec *LegacyWrite*, la méthode vérifie (1) que la page adressée est bien dans l'état *libre* (contrainte *effacer avant d'écrire*) et (2) si l'indice de la page dans le bloc contenant est différent de 0, le système vérifie également que la dernière page écrite dans le bloc concerné est la page d'indice précédent la page actuelle (contrainte d'écriture séquentielle au sein d'un bloc) ;
3. *updateFlashState* réalise la mise à jour de l'état du système : pour *LegacyWrite*, il s'agit de changer l'état de la page ciblée en *occupée* ;
4. *updateFlashStats* met à jour des statistiques sur les occurrences des opérations au niveau flash ;
5. *computePPC* (pour *compute Performance and Power Consumption*) implémente les modèles de performances et de consommation propres au composant flash. Ces modèles sont présentés dans la sous-section b) suivante.

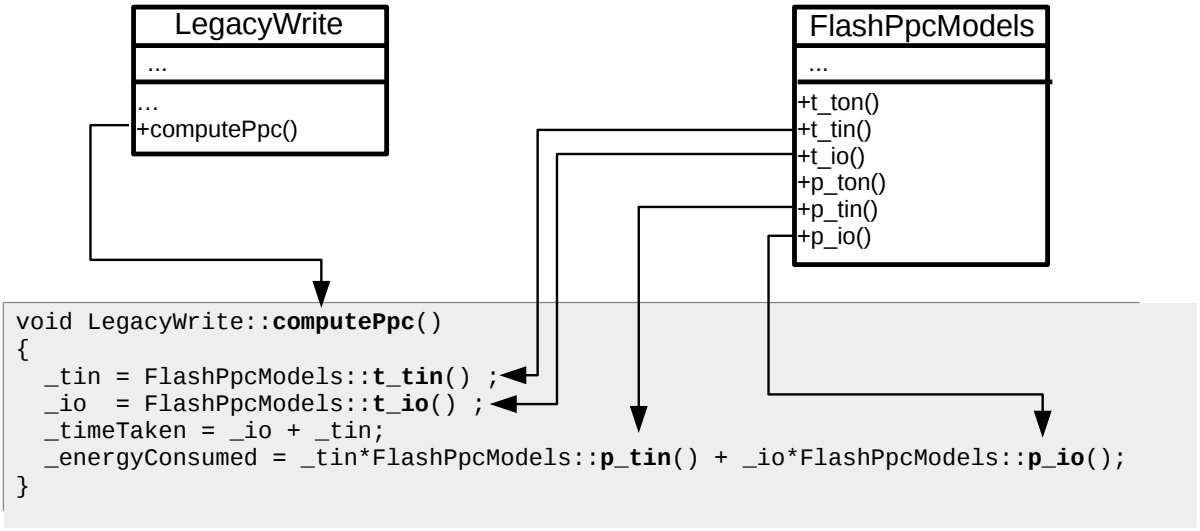


FIGURE 5.7 – Calcul des temps d'exécution et de l'énergie consommée par les opération flash : exemple pour *legacy write*

Ces méthodes sont exécutées dans l'ordre lors de l'application de la commande sur la représentation du (des) composant(s) flash simulés. Ces méthodes vérifient des informations contenues dans la représentation du composant flash (par exemple l'état d'une page donnée), chaque commande a donc une visibilité sur la représentation de l'état de la mémoire flash (objet *FlashSystem*) via un pointeur.

Les vérifications de contraintes relatives aux opérations flash sont une aide au développement de couches de gestion flash dans le simulateur.

b) Modèles de performances et de consommation relatifs au composant flash

Les modèles de performances et de consommation sont implémentés au sein de la fonction *computePpc*, redéfinie par tout type concret de commande flash. Les modèles de performances et de consommation avancés (présentés pour la plupart en annexe A page 209) sont également implémentés. On rappelle que ces modèles calculent le temps d'exécution et l'énergie consommée par les opérations flash en combinant les valeurs suivantes : T_{TON}, T_{TIN}, T_{IO}, P_{TON}, P_{TIN} et P_{IO}. C'est à l'utilisateur de fournir ces valeurs en entrée du simulateur, dans le fichier de paramètres. Ces valeurs peuvent être fournies sous diverses formes :

- Simplement sous forme de *constantes*, par exemple $T_{TON} = 200\mu s$;
- Sous la forme d'une valeur aléatoire, entre une borne minimale et maximale spécifiée par l'utilisateur (réalisé à chaque accès à la valeur) ;
- Sous la forme d'une valeur aléatoire basé sur une distribution normale, ou encore exponentielle, dont les paramètres sont spécifiés par l'utilisateur ;
- Pour T_{TON} et T_{TIN} il est possible de spécifier que leurs valeurs sont fonction de l'indice de la page adressée dans le bloc. Cela permet de représenter le fait que dans les puces MLC on peut constater une variation des latences en lecture et écriture en fonction de l'indice de la page lue ou écrite dans le bloc la contenant.

Toutes les générations de valeurs aléatoires basées sur des distributions sont effectuées en utilisant la bibliothèque scientifique GNU GSL (Gough, 2009). Le calcul des valeurs T_{TON}, T_{TIN}, etc. est confié à une classe dédiée qui propose une méthode par valeur à calculer. L'implémentation des modèles de performances et consommation flash au sein des fonctions *computePpc()* de chaque type concret d'opération flash fait appel à cette classe. La figure 5.7 illustre un exemple de calcul des estimations pour une opération de type *écriture legacy*.

Modèles de performances et de consommation flash simplifiés On a vu au chapitre précédent qu'il ne nous est pas possible d'utiliser directement les modèles de performances et de consommation avancé, pour la raison suivante : on ne dispose pas d'instruments de mesures suffisamment précis pour mesurer les temps d'exécution et les énergies consommées par les sous-opérations flash comme T_{TON}, P_{TIN}, etc. On utilise donc comme base les modèles réalisés au niveau pilote qui eux sont basés sur des mesures réelles. On paramètre alors une simulation de manière décrite ci-dessous.

Dans cette phase de validation, on définit le niveau pilote comme le niveau d'estimation de performances et de consommation le plus bas. On définit les valeurs de temps et d'énergie relatives aux sous-opérations flash (T_{TON}, P_{TIN}, etc.) comme égales à 0, ils n'ont donc pas d'impact sur les résultats de simulation. Pour chacun des types d'opérations flash de base, les estimations au niveau pilote sont basées sur les temps et l'énergie mesurés au chapitre précédent (voir table 4.4 page 140). Dans le cas d'un cache hit dans le buffer de lecture MTD, le temps est déterminé comme l'overhead en lecture au niveau pilote définit au chapitre précédent (voir équation 4.21 page 138). L'énergie est déterminée de manière similaire (voir équations 4.30 et 4.30 page 139). Il faut noter que, bien entendu, la méthode présentée ci-dessus est une manipulation destinée spécifiquement à la validation du simulateur, et non représentative d'une simulation standard.

4 Validation

Dans cette section on présente la méthodologie et les résultats de validation des modèles relatifs à la couche de gestion de type FFS, présentés au chapitre précédent. On se concentre sur les modèles associés à une couche de gestion aux niveaux pilote, FFS et VFS. On retrouve à ces niveaux les modèles *fonctionnels*, de *performances* et de *consommation*.

Concernant la consommation, on présente au niveau global *VFS + JFFS2 + pilote* une validation des modèles de consommation. Ce niveau global englobe tous les niveaux inférieurs, il s'agit donc d'une première étape pour la validation de l'ensemble du simulateur mis en œuvre lors de cette phase.

Au niveau VFS on présente également une validation fonctionnelle du modèle relatif à read-ahead. En effet il s'agit d'un algorithme relativement complexe, situé à haut niveau dans la pile de modèles fonctionnels : une erreur dans l'implémentation de cet algorithme pourrait se propager et potentiellement impacter fortement la précision des estimations en sortie.

Au niveau *JFFS2 + pilote*, on propose une validation des modèles de performances. On se concentre sur les fonctions de lecture et d'écritures de données dans les fichiers : *jffs2_readpage()*, *jffs2_write_begin()* et *jffs2_write_end()*. Comme énoncé précédemment il est difficile d'obtenir des mesures de consommation correspondant uniquement à la couche JFFS2. Les modèles de consommation sont donc validés au niveau supérieur (VFS). Les modèles fonctionnels ne font pas l'objet d'une phase de validation dédiée. Néanmoins, on peut considérer que le fait d'obtenir en sortie du simulateur une bonne précision pour l'estimation de performances constitue un élément de validation de l'implémentation fonctionnelle, du fait du fort couplage entre d'une part les modèles fonctionnels et, d'autre part, les modèles de performances et de consommation. De plus, il faut noter que certaines métriques relatives à la validation de performances permettent de s'assurer du bon fonctionnement des modèles fonctionnels JFFS2. Il s'agit par exemple du nombre d'écritures de pages flash réalisées durant un test, présenté dans les résultats dans cette section.

Concernant le *niveau pilote*, on peut observer que les caractéristiques de la régression linéaire utilisée lors de la construction du modèle de performances constituent en elles mêmes une validation du modèle (voir table 4.2 au chapitre précédent, page 137). On voit également au niveau des mesures de consommation que la puissance est très stable pour un type d'accès donné. Pour calculer l'énergie consommée on multiplie cette puissance par le temps d'exécution, lui-même considéré valide car venant du modèle de performances. Pour ces raisons on ne présente pas de validation spécifique au niveau pilote dans cette section.

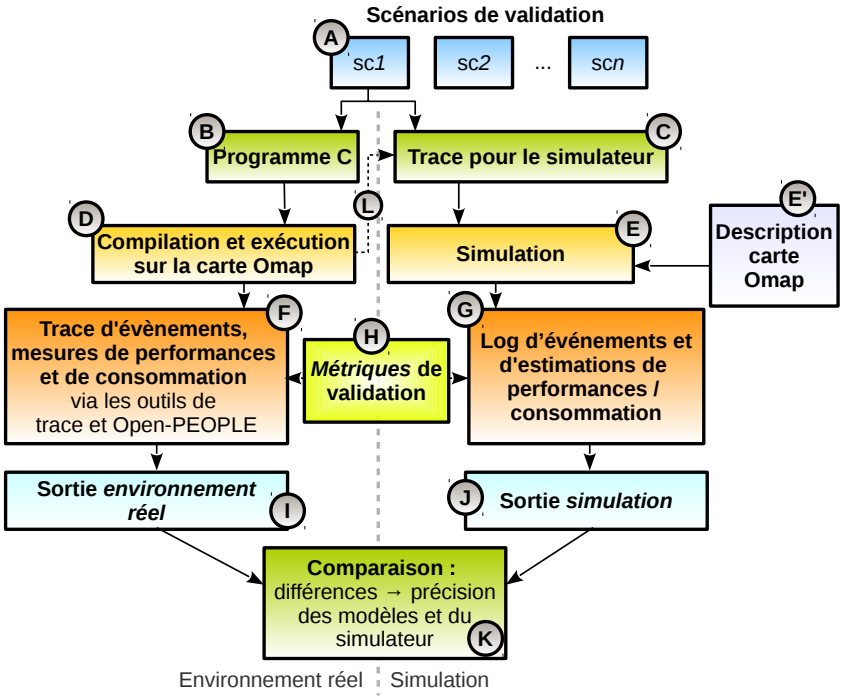


FIGURE 5.8 – Méthodologie de validation

Pour les raisons énoncées précédemment, les modèles concernant une couche de gestion de type FTL ne sont pas validés. Cela est vrai en particulier pour les modèles de performances et de consommation des opérations flash avancées (non *legacy*), qui restent dans l'état actuel au stade théorique, bien qu'ils soient implémentés dans OpenFlash.

4.1 Méthodologie de validation

OpenFlash est utilisé pour valider les modèles présentés au chapitre précédent. Pour les différentes expériences réalisées dans le cadre de la validation, la méthodologie adoptée est identique. Cette méthodologie est illustrée sur la figure 5.8.

Des *scénarios de validation* (A sur la figure 5.8) sont définis. Il s'agit de descriptions de programmes effectuant des requêtes d'E/S sur des fichiers. Pour chaque scénario, sa description est traduite (1) en un programme C compilé pour la carte Omap (B) et (2) en une trace pouvant être passée en entrée du simulateur (C). Le programme est alors exécuté sur la carte Omap (D). La trace est également passée en entrée du simulateur (E), qui prend de plus une description de la carte Omap en entrée (E', valeurs pour les paramètres des modèles implémentés).

Lors de l'exécution du programme C sur la carte Omap, les outils de trace présentés au chapitre 3 ainsi que la plate-forme Open-PEOPLE sont utilisés pour tracer de multiples métriques de performances (F), de consommation, et d'occurrences d'évènements. On les appelle *métriques de validation* (H). Les métriques de validation sont également récoltées au niveau de la simulation (G) : on récupère les estimations de performances et de consommation, et le simulateur est configuré pour afficher en sortie un journal des évènements relatifs aux métriques de performances. On obtient alors une sortie *en environnement réel* correspondant à l'ensemble des mesures de métriques de validation pour le scénario testé, et une sortie *en simulation* correspondant à l'équivalent simulé. Ces valeurs sont comparées, et la différence constatée détermine l'erreur d'estimation des modèles et la validité de leur implémentation dans le simulateur.

A noter que pour la constitution de la trace, certaines informations doivent être récupérées depuis l'exécution du programme C en environnement réel (L sur la figure 5.8). Il s'agit notamment des adresses

de requêtes aléatoires, déterminées via la fonction *rand()* dans le programme C et effectivement connues uniquement à l'exécution. Cela permet d'obtenir une trace en entrée de simulation la plus proche possible de ce qui se passe en environnement réel.

Pour l'intégralité des simulations réalisées dans le cadre de la validation, la description de la carte Omap présente dans le fichier de configuration est définie de manière suivante :

— Les paramètres du modèle structurel flash sont basés sur les informations de la fiche technique (Micron Inc., 2009) de la puce flash de la carte Omap : une taille de page de 2 Ko + 64 octets OOB, 64 pages par bloc, 800 blocs au sein d'un unique plan dans une LUN, elle-même située dans un canal. Le nombre de 800 blocs correspond à 100 Mo de données utilisateur : il s'agit de la taille de la partition utilisée dans le cadre de la validation. Même si la puce flash fait 256 Mo (2048 blocs), JFFS2 n'a de visibilité que sur cette partition ;

— Les paramètres des modèles de performances et consommation sont fixés aux valeurs déterminées lors de l'extraction de paramètres ;

— Comme expliqué plus haut, la fréquence de réveil de *pdflush* est fixée à 5 secondes, et le temps entre chaque passe du ramasse-miettes asynchrone lorsque les seuils d'espace libre / invalide justifient son exécution est défini comme présenté précédemment (voir section 2.2.2.d) page 177).

4.2 Validation des modèles de consommation : VFS, JFFS2 et pilote

Comme indiqué plus haut la validation concernant la consommation est réalisée au niveau général, c'est à dire au niveau VFS + FFS + pilote. On se concentre sur les opérations de lecture et d'écriture de données dans les fichiers.

a) Scénarios de validation

Lecture On valide les modèles de consommation au niveau global en définissant des scénarios mettant en lumière les éléments impactant cette métrique : le mode d'accès (séquentiel / aléatoire), le page cache, read-ahead et le buffer de lecture MTD. On définit en lecture des scénarios dans lesquels un programme effectue des lectures sur un fichier JFFS2 créé séquentiellement page par page. Le fichier est de grande taille (50 Mo) pour pouvoir supporter un nombre important de requêtes de lecture. On définit les deux scénarios suivants, destinés à valider les modèles selon l'impact du page cache :

1. Lecture après vidage du page cache ;
2. Lecture du fichier dont les données sont présentes dans le page cache au moment du test (le fichier est lu une première fois avant le test).

Chaque test est lancé effectuant 1500, 3000, 6000 et 12000 requêtes de lecture, (A) en séquentiel et (B) en aléatoire. On valide ainsi l'évolution des résultats en fonction du nombre de requêtes réalisées. Comme vu précédemment, effectuer avec JFFS2 des lectures séquentielles permet de profiter de l'effet de cache offert par le buffer de lecture MTD. Au contraire en lecture aléatoire le buffer n'est pas utilisé. Read-ahead impacte également les performances et la consommation de manière négative. Cela est particulièrement vrai en lecture aléatoire, on s'attend donc à une baisse de performances et une augmentation de la consommation en aléatoire.

Écriture On définit en écriture deux scénarios de test dont le but est de mettre en œuvre les éléments impactant la consommation : il s'agit principalement, dans le cadre de l'écriture, du write buffer de JFFS2.

1. Écriture séquentielle dans un fichier JFFS2 initialement vide ;
2. Écriture aléatoire dans un fichier JFFS2 de 5 Mo, créé avant chaque test par écriture séquentielle page Linux par page Linux.

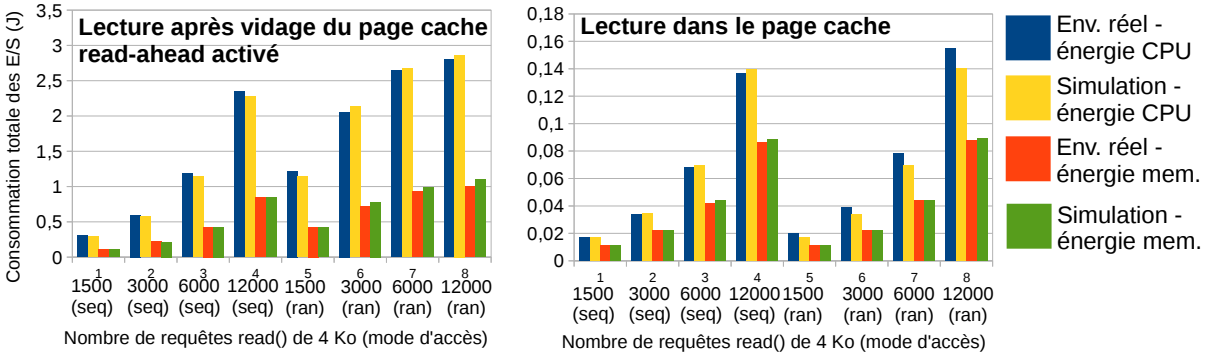


FIGURE 5.9 – Validation des modèles de consommation en lecture au niveau global : énergie dues aux E/S sur les puces CPU et mémoire, concernant la lecture dans un fichier non présent (gauche) et présent (droite) dans le page cache.

Pour chaque scénario, on fait varier la taille et le nombre de requêtes parmi les couples suivants : 12040 requêtes de 512 octets, 5120 requêtes de taille 1024 octets, 2560 requêtes de taille 2048, 1280 de taille 4096, 640 de taille 8192 et 640 de taille 16384 octets. Ainsi, on fixe la taille totale écrite à 5 Mo pour chaque exécution. L'écriture séquentielle dans un fichier vide est une opération relativement classique. Concernant les accès aléatoires, ils ne sont pas alignés sur des pages. On rappelle que VFS découpe une requête d'écriture située sur une frontière de page en deux appels en écriture au FFS. L'expérience déclenche donc des appels au FFS de tailles variées, qui sont tamponnés par le write buffer de JFFS2 lorsque cette taille est suffisamment petite.

b) Métriques de validation

En lecture et en écriture, on s'intéresse à la consommation totale des E/S pendant le test. OpenFlash intègre le profil de consommation de la carte Omap, et permet notamment la différenciation de l'énergie consommée sur la puce mémoire et la puce CPU, ainsi que la représentation de la valeur de la puissance à vide pour chacune des puces. Les mesures de puissance sur Open-PEOPLE sont traitées de manière à ne retenir que la consommation pendant les accès aux E/S, clairement définies dans chaque expérience par un plateau comme vu sur les courbes de consommation présentées aux chapitres précédents. Pour un test donné, chaque échantillon de puissance est multiplié par la période correspondant à la fréquence d'échantillonnage de l'instrument de mesure de consommation (1 kHz, soit une période de 1 ms) pour obtenir l'énergie consommée. Cette valeur d'énergie est comparée à la somme des énergies estimées en simulation pour tous les appels à *vfs_read()* et *vfs_write()* effectués dans un test.

c) Résultats

Lecture La figure 5.9 présente la comparaison des énergies totales dues aux E/S mesurées et estimées, pour les scénarios en lecture 1 et 3 présentés ci-dessus. Il s'agit de la lecture d'un fichier non présent dans le page cache, et d'un fichier présent dans le page cache. On peut voir que les estimations sont très proches des valeurs mesurées. La consommation due aux E/S croît avec (A) le nombre de requêtes et (B) le mode d'accès. On voit également que lire un fichier depuis le page cache, c'est à dire en RAM, est beaucoup plus économe en énergie que la lecture depuis la mémoire flash.

Pour inspecter plus en détail la précision du simulateur et des modèles implémentés, on peut se référer à la table 5.1. Cette table présente l'erreur d'estimation du simulateur par rapport aux mesures réelles, pour chaque scénario, nombre de pages Linux lues, et mode d'accès. On peut voir que de manière générale, l'erreur reste sous la barre des 10 %.

Read-ahead	Page cache vidé avant expérience	Mode d'accès	Lectures de pages Linux	Mesures réelles (J)		Simulation (J)		Erreur (%)	
				CPU	Mem.	CPU	Mem.	CPU	Mem.
on	oui	Séq.	1500	0.307	0.112	0.291	0.108	**5.27**	**3.31**
on	oui	Séq.	3000	0.596	0.217	0.576	0.214	**3.36**	**1.13**
on	oui	Séq.	6000	1.183	0.427	1.146	0.427	**3.12**	**0.01**
on	oui	Séq.	12000	2.352	0.848	2.281	0.850	**3.03**	**0.24**
on	oui	Aléa.	1500	1.218	0.426	1.146	0.417	**5.93**	**2.06**
on	oui	Aléa.	3000	2.059	0.72	2.137	0.78	**3.79**	**8.33**
on	oui	Aléa.	6000	2.647	0.931	2.673	0.989	**0.97**	**6.28**
on	oui	Aléa.	12000	2.805	1.005	2.864	1.096	**2.10**	**9.09**
off	oui	Séq.	1500	0.282	0.101	0.284	0.106	**0.86**	**4.97**
off	oui	Séq.	3000	0.572	0.205	0.569	0.212	**0.56**	**3.42**
off	oui	Séq.	6000	1.139	0.407	1.137	0.424	**0.13**	**4.18**
off	oui	Séq.	12000	2.262	0.805	2.275	0.847	**0.57**	**5.22**
off	oui	Aléa.	1500	0.616	0.218	0.635	0.235	**3.04**	**8.01**
off	oui	Aléa.	3000	1.112	0.398	1.161	0.432	**4.41**	**8.64**
off	oui	Aléa.	6000	1.84	0.657	1.957	0.735	**6.35**	**11.85**
off	oui	Aléa.	12000	2.663	0.959	2.813	1.078	**5.63**	**12.43**
on	non	Séq.	1500	0.017	0.011	0.017	0.011	**2.70**	**0.64**
on	non	Séq.	3000	0.034	0.022	0.035	0.022	**4.18**	**0.64**
on	non	Séq.	6000	0.068	0.042	0.070	0.044	**2.71**	**5.43**
on	non	Séq.	12000	0.137	0.086	0.140	0.089	**1.96**	**2.98**
on	non	Aléa.	1500	0.02	0.011	0.017	0.011	**15.00**	**0.00**
on	non	Aléa.	3000	0.039	0.022	0.034	0.022	**12.82**	**0.00**
on	non	Aléa.	6000	0.078	0.044	0.070	0.044	**10.51**	**0.45**
on	non	Aléa.	12000	0.155	0.088	0.140	0.089	**9.67**	**1.14**

Table 5.1 – Résultats de validation des modèles de consommation en lecture

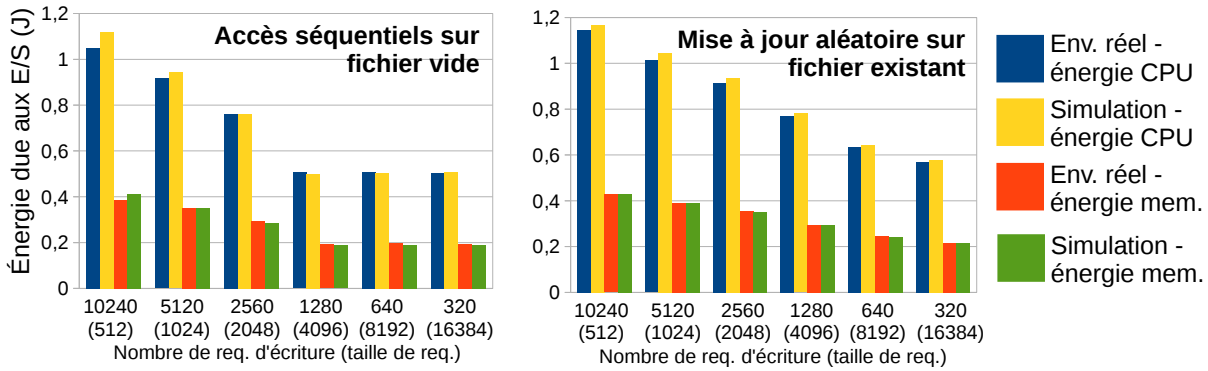


FIGURE 5.10 – Validation des modèles de consommation en écriture au niveau global : énergie dues aux E/S sur les puces CPU et mémoire, concernant des écritures séquentielles dans un fichier vide au départ (gauche) et des mises à jour aléatoires dans un fichier existant (droite).

Écriture On présente sur la figure 5.10 un comparatif de l'énergie totale due aux E/S pour les deux scénarios de test en écriture, pour chacun des couples (*nombre de requêtes réalisées, taille de requête*). Tout comme pour la lecture on constate que les estimations données par le simulateur sont très proches des mesures réalisées en environnement réel. On constate que lorsque la quantité de données écrites est fixée, écrire des paquets de données de taille importante est bien plus économe en énergie que l'écriture de petites quantités de données. L'énergie consommée due aux E/S est quasiment divisée par deux lorsque l'on effectue des écritures de taille 4096 Ko, par rapport à une taille de 512 octets. Cela est dû, entre autres, à la multiplication des overheads associés aux appels de fonctions de la pile de stockage qui sont plus nombreux lorsque l'on effectue 10240 écritures de 512 octets par rapport à 1280 écritures de 4096 octets. On constate également que passé 4096 octets, on n'observe pas d'amélioration de la consommation pour des tailles plus importantes. En sachant que VFS découpe les requêtes par page Linux (4 Ko), on peut en conclure que c'est l'overhead de JFFS2 qui influence le sur-coût en consommation constaté lors de requêtes de petite taille.

La table 5.2 présente le détail des mesures, estimations, ainsi que l'erreur associée aux modèles. On constate que dans le cas des écritures l'erreur est très faible. La quasi-totalité des expériences montre une erreur inférieure à 5%.

Scenario	Taille req.	Nombre req.	Mesures (J)		Simulation (J)		Erreur (%)	
			CPU	Mem.	CPU	Mem.	CPU	Mem.
Accès séquentiel sur fichier vide	512	10240	1.049	0.386	1.121	0.413	**6.85**	**6.97**
	1024	5120	0.917	0.351	0.942	0.351	**2.77**	**0.10**
	2048	2560	0.762	0.295	0.763	0.285	**0.13**	**3.23**
	4096	1280	0.506	0.192	0.501	0.188	**1.06**	**2.24**
	8192	640	0.507	0.199	0.515	0.189	**0.38**	**4.83**
	16384	320	0.505	0.194	0.518	0.190	**0.50**	**1.90**
Mises à jour aléatoires sur fichier existant	512	10240	1.145	0.427	1.167	0.429	**1.87**	**0.47**
	1024	5120	1.012	0.391	1.046	0.389	**3.27**	**0.63**
	2048	2560	0.916	0.356	0.934	0.349	**1.96**	**2.05**
	4096	1280	0.772	0.293	0.781	0.292	**1.14**	**0.26**
	8192	640	0.634	0.247	0.644	0.241	**1.63**	**2.33**
	16384	320	0.568	0.217	0.579	0.217	**1.82**	**0.10**

TABLE 5.2 – Résultats de validation des modèles de consommation en écriture

4.3 Validation des modèles fonctionnels : read-ahead

Valider la modélisation fonctionnelle de *read-ahead* est important car (A) il s'agit d'un algorithme relativement complexe et (B) une erreur dans cette modélisation présente à haut niveau (VFS) peut se propager aux niveaux inférieurs et fortement réduire la précision du simulateur. Dans les sous-sections suivantes on présente les scénarios, métriques, et résultats de validation.

a) Scénarios de validation

Les scénarios de validation sont des programmes effectuant un nombre variable d'appels systèmes read( ) sur un descripteur de fichier ouvert en lecture sur un fichier de 20 Mo. Le fichier est délibérément grand pour permettre l'exécution de scénarios avec un nombre important de requêtes sur des pages Linux distinctes, et ainsi assurer une validation exhaustive. Le fichier à lire est créé avant le lancement des scénarios. C'est un fichier comprenant des données aléatoires, créé par écriture séquentielle page par page. Avant chaque scénario le page cache est vidé. Dans chacun des programmes correspondants aux scénarios de test, le fichier est ouvert en lecture seule (drapeau `O_RDONLY` de la fonction *open( )*), et les appels à *read( )* sont effectués dans une boucle.

On définit neuf scénarios de validation, présentés dans la table 5.3. Dans cette table le terme *adresse* correspond à l'adresse logique à partir de laquelle la lecture est effectuée dans le fichier en octets. L'abréviation '@' signifie 'à l'adresse'.

Indice de scénario	**Description du scénario**	**Taille totale lue dans le fichier cible de 20 Mo**
1	Lecture @0, taille 4096 octets	4 Ko
2	Lecture @0, taille 512 octets	512 octets
3	Lecture @0, taille 20480 octets (5 pages Linux)	20480 octets
4	Lecture de l'intégralité du fichier, page Linux par page Linux	20 Mo
5	Lecture du fichier en 4 'flux' entrelacés : start0 = 0 ; start1 = taille_fichier / 4 ; start2 = taille_fichier / 2 : start3 = start1 +start2 ; for(i=0 ; i<(nb_pages_fichier), i++) read @(start0+i) 4096 octets // `flux 0` read @(start1+i) 4096 octets // `flux 1` read @(start2+i) 4096 octets // `flux 2` read @(start3+i) 4096 octets // `flux 3`	20 Mo
6	Lecture aléatoire dans le fichier de 5120 pages (4 Ko), alignées.	20 Mo
7	Lecture aléatoire dans le fichier de 5120 morceaux de 4 Ko, non alignés sur des pages Linux.	20 Mo
8	Lecture aléatoire dans le fichier de 500 morceaux de 512 octets, non alignés.	250 Ko
9	Lecture aléatoire dans le fichier de 500 morceaux de 20480 octets, non alignés.	environ 9.8 Mo

Table 5.3 – Scénarios de validation du modèle fonctionnel de read-ahead

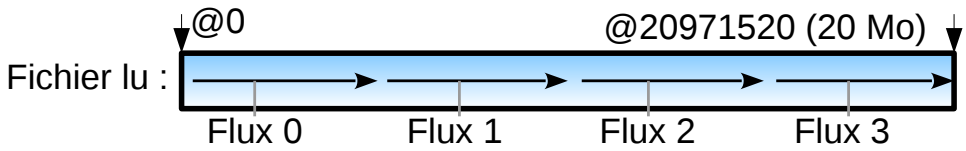


Figure 5.11 – Illustration de la lecture en flux réalisée par le scénario 5 (voir table 5.3)

Avec cette ensemble de scénarios on tente de couvrir un large spectre de modes d'accès en lecture par un programme à un fichier. En effet, ces scénarios sont utilisés pour valider le modèle fonctionnel de read-ahead, mais également les modèles de performances pour JFFS2. Les scénarios (sc.) 1, 2 et 3 effectuent un faible nombre de lectures en début de fichier : respectivement une lecture d'une page Linux (sc. 1), une lecture de 512 octets (sc. 2) et une lecture des 5 premières pages du fichier (sc. 3). L'objectif est là de tester la validité des modèles sur des accès volontairement très simples (faible nombre d'opérations). Le scénario 4 effectue une lecture de l'intégralité du fichier, séquentiellement, page Linux par page. La lecture séquentielle d'un fichier est une opération très répandue et il est logique de tester la validité des modèle face à ce type d'accès. Le scénario 5 lit l'intégralité du fichier en cinq flux entrelacés tel qu'illustré sur la figure 5.11. On sait que read-ahead peut adapter son comportement en fonction de flux de lecture passés (voir algorithme 15 page 152). Alors que les scénarios 1 à 5 font des accès séquentiels, les scénarios 6 à 9 effectuent des accès aléatoires. On a vu précédemment que le comportement de read-ahead, ainsi que les performances et la consommation, étaient impactés par le mode d'accès, séquentiel ou aléatoire. Le scénario 6 lit 5120 (nombre de pages Linux dans un fichier de 20 Mo) paquets de 4 Ko alignés sur les pages Linux du fichier. Le scénario 7 présente un même nombre et une même taille de requête, mais ces dernières ne sont pas alignées, ce qui signifie qu'une requête de 4 Ko peut chevaucher deux pages Linux du fichier cible. Enfin, au niveau des scénarios 8 et 9 on varie la taille de requête : ils effectuent chacun 500 requêtes de taille 512 octets (sc. 8) et 20480 octets (sc. 9).

b) Métriques de validation

Concernant read-ahead, les métriques de validation sont les suivantes :

1. Le *nombre de page cache hits* générés par le scénario de test. Un cache hit se produit lorsque le programme de test demande à lire une page du fichier cible, et que la page est déjà présente dans le page cache : il n'y a pas d'accès au stockage secondaire ;
2. La *liste contenant les indices des pages demandées par le programme de test qui débouchent sur un appel à la fonction noyau page_cache_async_readahead()*. La taille de cette liste correspond au nombre de passes read-ahead en mode asynchrone ;
3. Idem que ci-dessus, pour les appels à read-ahead en mode synchrone ;
4. Le *nombre total d'appels à read-ahead*, indépendamment du mode synchrone ou asynchrone. Il s'agit d'une valeur tracée indépendamment des deux valeurs ci-dessus, et elle n'est pas déterminée en faisant leur somme. Le nombre d'appels à read-ahead doit être égal à cette somme pour prouver la validité de l'implémentation ;
5. La liste contenant, pour chaque passe de read-ahead, (synchrone ou asynchrone), *les tailles de fenêtre calculée par chacune des passes* ;
6. La *liste des indices de pages Linux sur lesquelles read-ahead pose le* flag read-ahead ;
7. Le *nombre de pages Linux du fichier cible lues via le FFS* : il s'agit de l'intégralité des pages lues par read-ahead qui ne sont pas déjà dans le page cache au moment ou read-ahead décide de les lire ;
8. Le *nombre d'appels à vfs_read*, c'est le nombre d'appels système *read()* réalisés par l'applicatif.

c) Résultats

Les programmes C sont exécutés, et les traces correspondantes sont simulées comme spécifié dans la méthodologie présentée ci-dessus. On peut noter que pour l'intégralité des scénarios, les valeurs de toutes les différentes métriques de test sont totalement similaires en environnement réel et en simulation, ce qui prouve la validité du modèle fonctionnel de read-ahead. Les résultats validant le modèle à 100%, ils ne sont pas présentés ici mais en annexe B, page 225.

4.4 Validation des modèles de performances : JFFS2

Pour valider les modèles de performances de JFFS2, on se concentre sur les opérations de lecture et d'écriture de données dans des fichiers au niveau JFFS2 : *jffs2_readpage()*, *jffs2_write_begin()* et *jffs2_write_end()*.

a) Scénarios de validation

Lecture On a vu au chapitre 3 que la fragmentation avait un impact très important sur les performances en lecture de JFFS2. On reprend les scénarios 1 à 9 utilisés pour valider le modèle fonctionnel relatif à read-ahead. Chaque scénario est lancé deux fois : une fois sur un fichier de 20 Mo créé par écriture séquentielle page Linux par page Linux, une deuxième fois sur un fichier créé de même manière, puis mis à jour via 40960 écritures de 512 octets à des offset aléatoires, non alignés sur des pages Linux. Le deuxième fichier cible est donc fortement fragmenté. Ci-dessous, on fait référence au premier fichier en tant que *fichier créé séquentiellement* et au second comme *fichier créé aléatoirement*.

Écriture On définit 6 scénarios de validation concernant l'écriture. Ils sont présentés dans la table 5.4. Comme pour la lecture on tente de couvrir un nombre important de scénarios d'écriture dans un fichier par un programme : le scénario 1 effectue des écritures de taille une page Linux dans un fichier initialement vide, ce qui est une opération relativement classique. Le scénario 2 effectue également des écritures séquentielles, chacune de taille 512 octet. Cette taille est inférieure à la taille d'une page flash, le buffer d'écriture de JFFS2 pourra dans ce cas tamponner certaines écritures ce qui impacte les performances et la consommation. Les scénarios 3 à 6 effectuent des écritures aléatoires. Comme énoncé précédemment on ne s'intéresse pas au cas de "trous" de données dans un fichier (écriture à un offset supérieur à la taille du fichier) : les fichiers cibles doivent donc être créés au préalable. Ils sont créés par écriture séquentielle.

Au niveau VFS une requête d'écriture est découpée à la granularité de pages Linux avant d'être passée au FFS. Au niveau FFS, la taille écrite via *jffs2_write_begin()* et *jffs2_write_end()* dépend donc, entre autres, de l'alignement de la requête *write()* sur des pages Linux. On rappelle qu'une requête d'écriture située sur une frontière de page Linux est découpée par VFS en deux appels aux FFS : il est donc important de valider face à des requêtes alignées et non alignées, cela est fait via les scénarios 3 et 4. Le scénario 5 fixe la taille des requêtes à 512 octets pour valider l'impact du write buffer en écriture aléatoire.

On sait que le rôle de *jffs2_write_begin()* est de ramener dans le page cache la page concernée par une écriture, si jamais cette dernière ne s'y trouve pas. Concernant le scénario 6, on force le fait qu'aucune page touchée par une écriture ne se trouve dans le page cache en vidant ce dernier. On doit alors constater un impact sur les performances et la consommation de *jffs2_write_begin()*.

b) Métriques de validation

Lecture Concernant la lecture, on se base principalement sur le temps d'exécution de *jffs2_write_begin()*. On compare la moyenne des temps de réponse de cette fonction en environnement réel et en simulation, et ce pour chaque scénario et pour chaque mode de création du fichier cible (séquentiel /

Indice de scénario	Description	Taille totale écrite
1	5120 requêtes séquentielles de 4 Ko dans un fichier initialement vide	20 Mo
2	40960 requêtes séquentielles de 512 octets dans un fichier initialement vide	20 Mo
3	5120 requêtes aléatoires de taille 4 Ko alignées sur des pages Linux dans un fichier de 20 Mo créé en séquentiel	20 Mo
4	5120 requêtes aléatoires de taille 4 Ko non alignées sur des pages Linux dans un fichier de 20 Mo créé en séquentiel	20 Mo
5	40960 requêtes aléatoires de 512 octets non alignées dans un fichier de 20 Mo créé en séquentiel	20 Mo
6	Idem que le scénario 3, sauf que le page cache est vidé avant le lancement des écritures aléatoires	20 Mo

Table 5.4 – Scénarios de validation pour les performances de JFFS2 en écriture

aléatoire). La moyenne étant une métrique assez globale, on compare également l'écart type de l'ensemble des temps d'exécution des appels à *jffs2_write_begin()* constatés pour chaque scénario. On considère également, pour chaque scénario et mode de création du fichier cible, la distribution de l'ensemble des temps d'exécution des appels à la fonction JFFS2 de lecture de page.

Écriture On se base, pour l'écriture, sur les temps d'exécution de *jffs2_write_begin()* et *jffs2_write_end()*. On présente des moyennes par scénarios, ainsi que la distribution des temps d'exécution de ces deux fonctions par scénario. On compare également pour chaque scénario le nombre moyen d'écritures de pages flash constaté en environnement réel et en simulation. Concernant les scénarios 3 à 6, les métriques de performances ne sont mesurées qu'à partir du lancement des écritures aléatoires.

c) Résultats

Lecture La figure 5.12 présente les moyennes et écarts type des temps d'exécution de l'ensemble des appels à *jffs2_readpage()*, pour chaque scénario et pour chaque mode de création du fichier cible. On constate que les valeurs mesurées en environnement réel et les estimations obtenues par simulation sont très proches dans la majeure partie des scénarios. Seuls les scénarios 1 et 2 ciblant le fichier créé de manière aléatoire présentent une légère différence au niveau de la moyenne, et une différence plutôt importante pour ce qui est de l'écart type. On rappelle que ces scénarios effectuent chacun une seule lecture de petite taille (4 Ko pour le scénario 1, 512 octets pour le scénario 2), le nombre d'échantillons pour calculer la moyenne est donc trop réduit pour prendre en compte cette différence.

L'erreur relative à l'estimation concernant la moyenne des temps d'exécution de *jffs2_readpage()* reste sous la barre des 10% pour les scénarios 3 à 9. Les valeurs précises de l'erreur pour chaque scénario et mode de création du fichier cible sont présentées dans la table 5.5.

Pour avoir une idée plus fine de la précision des modèles, on peut s'intéresser à la distribution des temps d'exécution des appels à *jffs2_readpage()* dans chaque scénario. La figure 5.13 présente un exemple pour les scénario 6 et 7, et pour les deux modes de création de fichier cible : en aléatoire pour le scénario 6 et en séquentiel pour le 7. On peut constater visuellement que les estimations réalisées

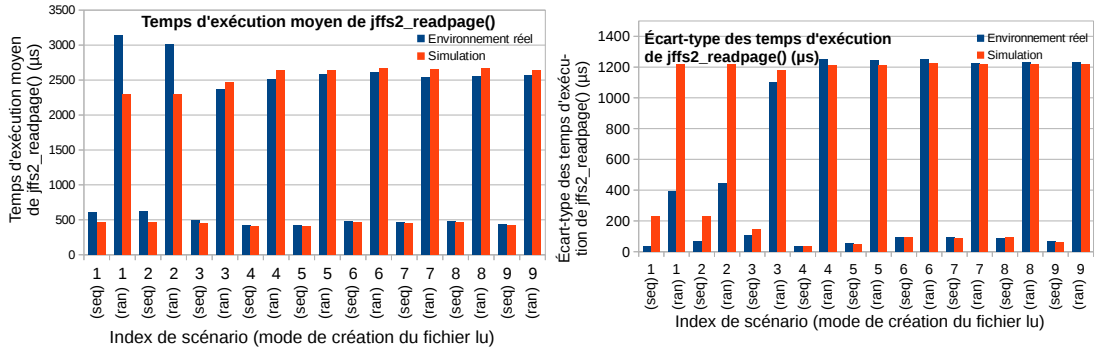


Figure 5.12 – Moyenne (gauche) et écart-type (droite) de l'ensemble des temps d'exécution des appels à *jffs2_readpage* pour les différents scénarios de test en lecture, sur fichier cible créé en séquentiel (par paquets de 4096 Ko) et en aléatoire (paquets de 512 octets).

Id. de scénario	Mode de création du fichier cible	Moyenne des temps d'exécution de *jffs2_readpage*		
		Env. réel (μs)	Simulation (μs)	Erreur (%)
1	Séquentiel	601.64	463.20	**23.01**
	Aléatoire	3143.38	2293.20	**27.05**
2	Séquentiel	625.44	463.20	**25.94**
	Aléatoire	3009.98	2293.20	**23.81**
3	Séquentiel	494.77	448.06	**9.44**
	Aléatoire	2373.99	2471.82	**4.12**
4	Séquentiel	418.63	402.64	**3.82**
	Aléatoire	2504.43	2632.66	**5.12**
5	Séquentiel	424.37	408.64	**3.70**
	Aléatoire	2586.74	2635.98	**1.90**
6	Séquentiel	483.34	469.14	**2.94**
	Aléatoire	2611.33	2668.36	**2.18**
7	Séquentiel	469.18	454.43	**3.14**
	Aléatoire	2535.35	2659.16	**4.88**
8	Séquentiel	476.11	462.34	**2.89**
	Aléatoire	2556.28	2675.57	**4.67**
9	Séquentiel	434.84	419.99	**3.42**
	Aléatoire	2569.34	2633.76	**2.51**

Table 5.5 – Calcul de l'erreur de l'estimation des performances de JFFS2 en lecture pour les scénarios de tests.

par le simulateur sont très proches des résultats en environnement réel. Ce constat est également vrai pour les distributions relatives aux autres scénarios, qui sont incluses en annexe C, page 233. Cela confirme la validité du modèle de performance en lecture.

Écriture La figure 5.14 présente la moyenne des temps d'exécution des appels à *jffs2_write_begin()* et *jffs2_write_end()*, pour chaque scénario en environnement réel et en simulation. Les écarts-types sont également présent sur la figure. Sur la figure 5.15 on peut voir le nombre total d'opérations d'écritures de pages flash par scénario, mesurées et simulées. On constate que, pour chaque scénario, les estimations sont très proches des mesures réalisées en environnement réel. L'erreur d'estimation est présentée dans la table 5.6.

Tout comme pour les lectures, on peut observer les distributions des temps d'exécutions de *jffs2_write_begin()* et *jffs2_write_end()* par scénario, pour déterminer plus en détail la précision des modèles. La figure 5.16 présente un exemple pour le scénario 5. Les estimations sont similaires aux résultats en environnement réel. Au niveau de *jffs2_write_begin()*, on peut noter un fort pourcentage

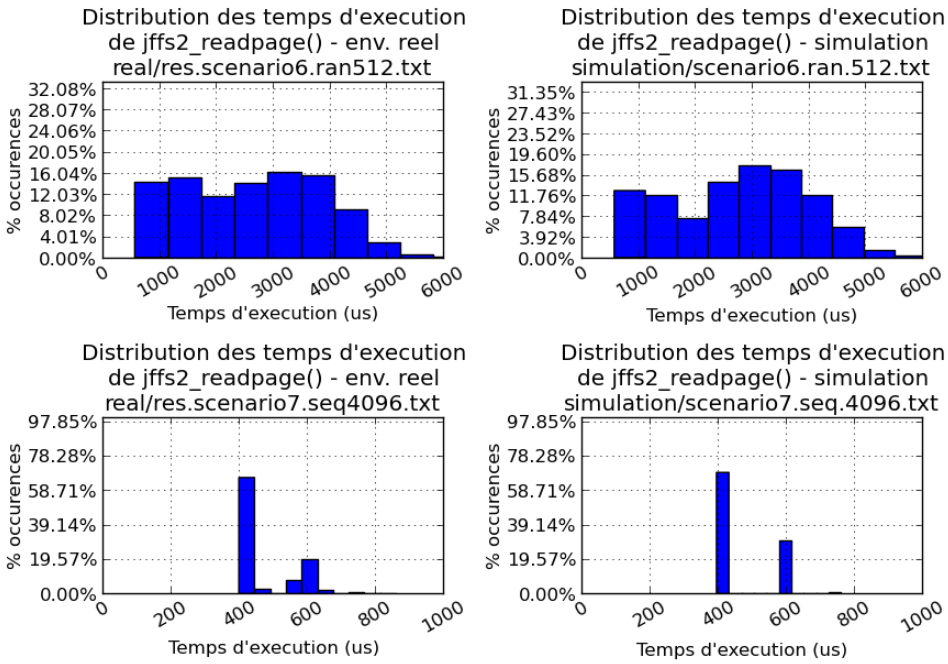


Figure 5.13 – Distribution des temps d'exécution de *jffs2_readpage()* pour le scénario 6 lancé sur fichier créé en aléatoire (haut) et le scénario 7 sur fichier créé en séquentiel (bas). Mesures en environnement réel à gauche, valeurs simulées à droite.

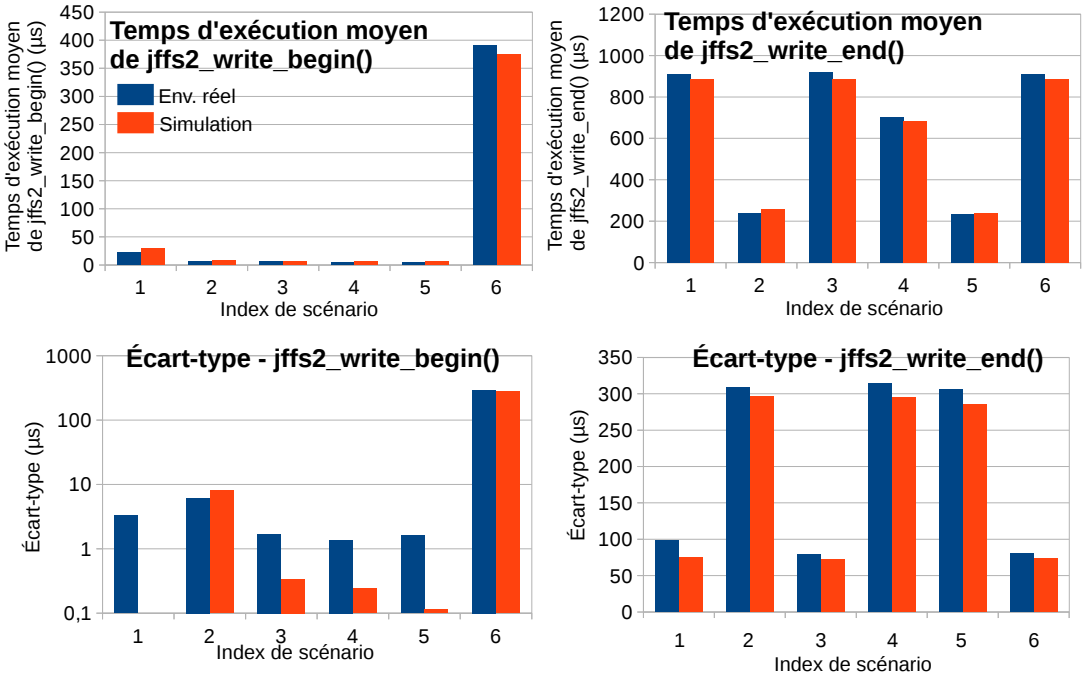


Figure 5.14 – Moyenne des temps d'exécution de *jffs2_write_begin()* (en haut à gauche), *jffs2_write_end()* (en haut à droite), et écarts-types pour ces moyennes, pour *jffs2_write_begin()* (en bas à gauche) et *jffs2_write_end()* (en haut à droite). La courbe traitant des écarts-types présente une échelle logarithmique en y.

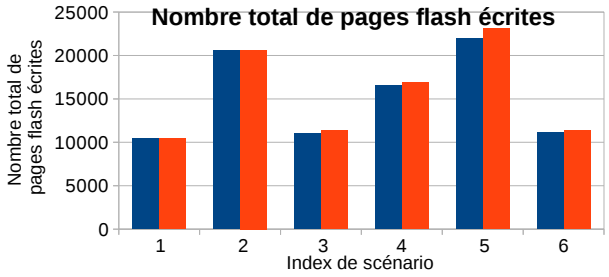


Figure 5.15 – Nombre d'écritures flash réalisées lors de chaque scénario.

Id. de scénario	Temps d'exécution moyen (μs)				(C) Nombre d'écritures de pages flash		Erreur (%)		
	(A) jffs2_write_begin()		(B) jffs2_write_end()						
	Env. réel	Simu.	Env. réel	Simu.	Env. réel	Simu.	(A)	(B)	(C)
1	22.5	30	911	887	10423	10432	**33.33**	**2.63**	**0.09**
2	6.4	8.7	240	257	20582	20604	**35.94**	**7.08**	**0.11**
3	5.7	5.7	922	886	10989	11348	**0.00**	**3.90**	**3.27**
4	5	5.7	701	682	16568	16887	**14.00**	**2.71**	**1.93**
5	4.89	5.7	233	240	21927	23118	**16.56**	**3.00**	**5.43**
6	391	374	910	887	11123	11387	**4.35**	**2.53**	**2.37**

Table 5.6 – Calcul de l'erreur de l'estimation des performances de JFFS2 en écriture pour les scénarios de tests.

d'erreur pour les scénarios 1 et 2. On rappelle que le temps d'exécution de cette fonction est très court et qu'il n'y a pas d'accès flash réalisés dans le cadre des scénarios 1 à 5. Ainsi, on ne s'inquiète pas de la différence de pourcentages.

Les graphiques présentant les distributions des temps d'exécution des fonctions *jffs2_write_begin()* et *jffs2_write_end()* relatives à chaque scénario en environnement réel et en simulations sont présentés en annexe C, page 233.

5 Conclusion

Dans ce chapitre, on a tout d'abord présenté l'une des contributions principales de cette thèse, un simulateur de systèmes de stockage à base de mémoire flash nommé *OpenFlash*. Ce simulateur reprend les points forts des simulateurs existants tel que le support d'architectures et d'opérations flash avancées, et ajoute de nouvelles fonctionnalités non présentes dans l'état de l'art actuel, en particulier le support des systèmes de fichiers dédiés aux mémoires flash, et la gestion d'évènements asynchrones comme les ramasse-miettes en arrière plan présents dans bon nombre de systèmes actuels. Outre les FFS, le simulateur supporte également la représentation de systèmes à base de FTL. Ainsi, il couvre l'intégralité des applications des systèmes de stockage à base de mémoire flash actuels. La totalité des modèles présentés au chapitre précédent sont implémentés au sein de l'outil proposé : les modèles fonctionnels, de performances et de consommation, les modèles de charge d'E/S, ainsi que les modèles relatifs au composant mémoire flash. On a présenté dans ce chapitre la manière dont les modèles sont implémentés dans l'outil.

Le premier rôle d'OpenFlash est d'obtenir des estimations concernant les performances et la consommation d'un système de stockage à base de mémoire flash à partir de deux entrées : La première entrée est une description du système à simuler via des paramètres représentant l'architecture du système, son profil de performances et de consommation, et des options relatives aux algorithmes de la couche de gestion flash. La deuxième entrée est une description de la charge d'E/S soumise au système simulé.

Le deuxième rôle du simulateur est de permettre de prototyper de manière aisée de nouveaux

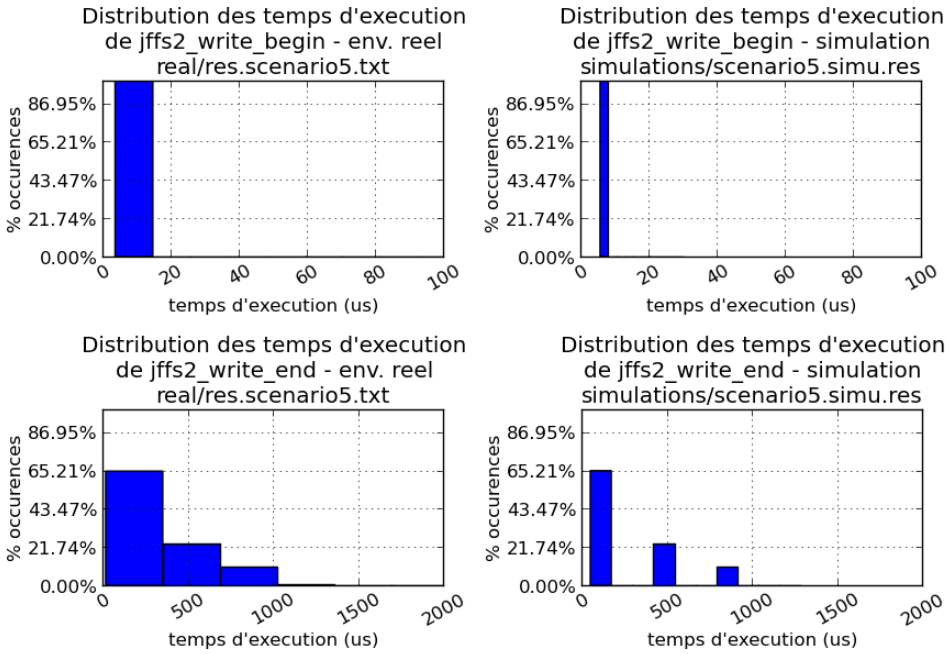


FIGURE 5.16 – Distributions des temps d'exécution de *jffs2_write_begin()* (haut), *jffs2_write_end()* (bas), en environnement réel (gauche) et en simulation (droite), concernant le scénario 5. La largeur des colonnes est fonction de la dispersion des données représentées par ces colonnes.

systèmes de gestion pour mémoire flash, de type FFS comme FTL. Le simulateur étant écrit en C++, on utilise certaines fonctionnalités offertes par la programmation orientée objet pour fournir une infrastructure de développement de couche de gestion. Un développeur qui souhaite prototyper une nouvelle couche de gestion dans cet outil est aidé par (A) la présence de classes abstraites lui indiquant les grandes lignes de définition de ses algorithmes, (B) la description fine des contraintes relatives aux opérations flash *legacy* et *avancées* et (C) le système de gestion d'erreurs et avertissements.

Dans l'état actuel des choses le simulateur ne supporte pas une définition automatique de l'état initial pré-simulation. L'état initial peut potentiellement impacter fortement les résultats de simulation. On a vu dans ce chapitre qu'il s'agit d'un problème complexe, l'exécution de la phase de *warm-up* étant dans les simulateurs actuels déléguée à l'utilisateur ou implémentée de manière non optimale. On a ici proposé des pistes de réflexion concernant la possibilité d'automatiser la mise en place d'un état initial de manière fine et exhaustive.

Dans la dernière section de ce chapitre, on a présenté la phase de validation des modèles présentés au chapitre précédent, concernant les FFS. Cette validation est réalisée à l'aide du simulateur. On s'est particulièrement intéressé au modèle fonctionnel du mécanisme de pré-chargement *read-ahead*, aux modèles de performances du FFS JFFS2, et aux modèles de consommation globaux intégrant l'ensemble de la pile de gestion des E/S sous Linux. La méthodologie de validation consiste à comparer les estimations données par le simulateur à des mesures réalisées en environnement réel (*métriques de validation)*, et ce dans de multiples cas d'application (*scénarios de validation*). Dans une grande majorité des cas les modèles implémentés au sein du simulateur présentent une erreur inférieure à 10 %.

Conclusion

Les performances et la consommation énergétique sont des métriques cruciales dans le domaine des systèmes embarqués. Dans cette thèse, on se concentre sur le stockage secondaire à base de mémoire flash NAND, largement utilisée dans ce domaine.

Les contributions apportées dans ce travail de thèse s'articulent autour d'une méthodologie de modélisation en trois étapes : (A) *l'exploration* correspondant à l'identification des éléments du système de stockage à base de flash impactant la consommation et les performances ; (B) *la modélisation* et l'*extraction de paramètres*, représentant respectivement la modélisation de l'impact relatif à chaque élément au sein d'un modèle, et la spécialisation de ce modèle par rapport à une plate-forme matérielle et logicielle réelle ; et enfin (C) l'implémentation de ces modèles dans un outil de *simulation*, pour l'obtention d'estimations concernant les performances et la consommation, ainsi que le prototypage de nouvelles couches de gestion flash.

Les contributions de ce travail de thèse sont les suivantes :

— L'identification des éléments impactant les performances et la consommation d'un système de stockage embarqué à base de FFS ;
— La représentation de l'impact de ces éléments au sein d'une suite de modèles de types variés ;
— Une série de modèles fonctionnels décrivant le fonctionnement d'une couche de gestion du stockage flash de type FFS : VFS, le pilote NAND, et un exemple de FFS, JFFS2 ;
— Une méthodologie d'extraction des paramètres relatifs à chaque modèle de performances et de consommation, pour construire un profil relatif à ces métriques pour plate-forme matérielle / logicielle donnée ;
— L'application de cette méthodologie à un cas d'étude concernant la carte *Mistral Omap3evm* exécutant Linux 2.6.37 ;
— Un simulateur implémentant les modèles, permettant d'obtenir des estimations concernant les performances et la consommation d'une application accédant au stockage secondaire de type flash. Le simulateur offre également une infrastructure pour le développement de nouvelles couches de gestion flash et l'évaluation de leur performances / consommation ;
— Un jeu d'outils génériques d'exploration des performances pour un système de type FFS, utilisés tout au long du processus de modélisation.

Bien que le travail présenté ici cible principalement les FFS, les systèmes à base de FTL sont également pris en compte : on présente les grandes lignes pour la définition fonctionnelle de couches de gestion de type FTL. Les modèles relatifs au composant mémoire flash sont adaptés pour supporter les architectures et opérations dites avancées présentes dans certains systèmes à base de FTL comme les SSD.

Éléments impactant les performances et la consommation dans un système de type FFS Les éléments suivants ont étés identifiés comme impactant ces métriques :

— Les algorithmes de la couche de gestion flash : il s'agit de la manière dont la charge d'E/S applicative est traitée. Au niveau FFS on identifie plusieurs niveaux de traitement. Il s'agit du système de fichiers virtuel (VFS), du FFS à proprement parler, et du pilote NAND. Au niveau VFS on note la présence du page cache et des algorithmes *read-ahead* et *write-back* qui y sont liés. Au niveau FFS, on note la manière dont les données sont réparties sur la flash, ainsi que

l'utilisation potentielle de tampons (par exemple le *write-buffer* JFFS2). Au niveau pilote on peut noter également la présence d'un buffer de lecture.

— Les caractéristiques de la charge d'E/S appliquée au système de stockage : nombre, type et taille des requêtes. Le mode d'accès (séquentiel ou aléatoire) présente également un impact considérable : les lectures aléatoires ont tendance à moins profiter des effets de cache, et les écritures aléatoires provoquent de la fragmentation ;

— L'état du système : il consiste en l'état des différents tampons, état des pages de la mémoire flash (quantité de données valides / invalides, et d'espace libre), ainsi que la fragmentation en flash des fichiers accédés ;

— Enfin, les profils de performances et de consommation des différents composants matériels utilisés dans le système de stockage : il s'agit pour la mémoire flash des temps d'exécutions et de l'énergie consommée lors des opérations de base. Pour les composants CPU et RAM, ils déterminent les performances et la consommation de la couche de gestion.

Au niveau de cette phase d'exploration, les mesures de consommation ont étés réalisées via la plate-forme de mesure à distance *OPEN-PEOPLE*. Concernant les mesures de performances, elles ont étés effectuées via une suite d'outils dédiés à l'exploration des performances des systèmes de type FFS. Ces outils ont également été utilisés en phase de modélisation et de simulation.

Modélisation des performances et de la consommation pour systèmes de fichiers dédiés aux mémoires flash Un jeu de modèles de différents types, ciblant les opérations de transfert de données dans des fichiers gérés par FFS a été proposé :

— Les modèles fonctionnels représentent les algorithmes de la couche de gestion. Ils traitent une représentation de la charge d'E/S en entrée, et maintiennent / mettent à jour l'état du système. Les modèles construits pour VFS et le pilote NAND MTD sont réutilisables, par exemple dans le cas de la construction d'un modèle fonctionnel pour un FFS autre que JFFS2 ;

— Les modèles de performances et de consommation représentent le profil de performance / consommation des éléments mémoire flash, CPU, et RAM. Pour construire un modèle de performances et de consommation correspondant à une plate-forme matérielle / logicielle donnée, une méthodologie d'extraction de paramètres est proposée ;

— Outre un modèle de performance / consommation, le composant flash est concerné par le modèle structurel représentant son architecture (nombre de pages par bloc, nombre de blocs par plan, etc.), et un modèle opérationnel représentant les opérations flash supportées, les contraintes associées, et la manière dont ces opérations modifient l'état des pages flash modélisées. Les modèles relatif au composant flash sont ré-utilisables, par exemple dans le cadre de simulation concernant la FTL ;

— Enfin, le modèle de charge représente la charge d'E/S soumise par l'applicatif au système de stockage. Dans le cas des FFS, cette charge est représentée au niveau de granularité des appels systèmes (notamment *read()* et *write()*). Ces modèles sont bien entendu ré-utilisables dans le cadre de simulations concernant d'autres types de FFS, lorsque ces derniers seront implémentés dans le simulateur.

Pour chaque type de modèle, un exemple concret est présenté, correspondant à la carte embarquée *Mistral Omap3evm*. Les modèles de performances et de consommation sont réalisés à parti de mesures.

On a également montré comment ces modèles peuvent être généralisés et modifiés pour s'adapter à un système de type FTL : ils permettent donc de couvrir toutes les applications des systèmes à base de mémoire flash actuels.

OpenFlash : un simulateur de systèmes de stockage à base de mémoire flash L'une des contributions principales de ce travail est un outil de simulation de systèmes de stockage à base de mémoires flash, ciblant (A) l'estimation des performances et de la consommation et (B) le prototypage de nouvelles

couche de gestions. Par rapport aux simulateurs pour mémoire flash existants, cet outil apporte les fonctionnalités nouvelles suivantes :

— Le support des systèmes de type FFS ;
— Le support des évènements asynchrones, en particulier les mécanismes de ramasse-miettes en arrière plan, présents dans de nombreux systèmes à base de mémoire flash actuels.

De plus, l'outil reprend les points forts des simulateurs actuels :

— Concernant la partie FFS, la possibilité d'intégration de modèles de performances et de consommation basés sur des mesures en environnement réel ;
— Le support des systèmes de type FTL ;
— Le support d'architectures flash avancées (multi plans / LUN / canaux), et des opérations avancées qui vont de pair avec ce type d'architecture complexes.

L'outil permet premièrement d'obtenir des estimations de performances et de consommation concernant l'utilisation d'un système de stockage à base de mémoire flash. Il prend une description du système en entrée (paramètres des modèles) : cette description concerne l'architecture du système, les algorithmes de la couche de gestion utilisée, et le profil du système concernant les performances et la consommation. L'outil prend également en entrée une description de la charge d'E/S soumise au système, au niveau des appels systèmes pour un FFS, et au niveau bloc pour une FTL.

Le simulateur, écrit en C++, offre de plus une infrastructure de développement relativement simple pour concevoir de nouvelles couches de gestion (FFS et FTL). Cela est notamment possible grâce à la présence de classes abstraites, définissant les grandes lignes de l'implémentation d'une couche de gestion, et dont il faut faire hériter les classes représentant cette implémentation. A moyen terme l'objectif est de disposer d'un catalogue de couches de gestions au sein du simulateur, pour pouvoir réaliser des études comparatives de performances et de consommation.

Le simulateur peut également être utilisé pour valider des modèles. On a ainsi présenté à l'aide de l'outil une validation des modèles de performances et de consommation relatifs à la carte Omap. La validation montre que l'erreur d'estimation des modèles reste, dans la majeure partie des cas, sous la barre des 10%.

Perspectives de travaux futurs

On peut diviser les perspectives de travaux futurs en deux catégories : les travaux relatifs aux modèles, et ceux concernant l'évolution du simulateur.

Évolution et raffinement des modèles

Niveau FFS - modèles existants

Validation des modèles JFFS2 Pour finaliser la validation des modèles relatifs à JFFS2 sur la carte Omap, il est nécessaire d'utiliser un macro-benchmark. La raison pour laquelle cela n'est pas présenté dans ce travail de thèse est la difficulté d'obtenir une trace pour une application à large échelle telle qu'un macro-benchmark (présenté plus en détail dans la section concernant le simulateur). Comme vu au chapitre 2, *postmark*, bien que non-spécifique à l'embarqué se pose en bon candidat pour une validation globale ;

Étude de l'impact des méta-données Dans ce travail on s'est concentré sur les transferts de données, principalement via les opérations de lecture et d'écriture dans les fichiers. Si l'on considère un système de fichier à grande échelle, contenant un nombre très important de fichiers et de répertoires, il est probable que l'impact des méta-données sur les performances et la consommation ne soit pas négligeable. On pense notamment à l'impact des structures d'indexation au niveau

FFS et VFS. Il est important de s'intéresser à cette problématique. En effet il est probable que dans ce genre de cas l'overhead aux niveaux VFS et FFS ne puisse être considéré comme une valeur constante ;

Étude plus fine de la consommation On a vu que la granularité des instruments de mesure de consommation peut rendre difficile les mesures à certains niveaux isolés (notamment le niveau JFFS2, par exemple pour une passe du ramasse-miettes). La possibilité de déclencher des mesures à granularité fine (par exemple démarrer / stopper la mesure directement depuis le code du noyau) permettrait la réalisation de modèles plus précis ;

Raffinement du modèle du page cache Le modèle actuel est très simple, et ne permet notamment pas de simuler un système sous forte pression mémoire. On peut imaginer faire évoluer le modèle du page cache, notamment en rendant la quantité de pages maximale stockées variable au cours d'une simulation.

Niveau FFS - nouveaux modèles

Considération d'autres FFS Pour étoffer le nombre de FFS supportés par le simulateur, il est nécessaire de modéliser le comportement d'autres types de FFS : on peut penser aux systèmes populaires UBIFS et YAFFS2, mais également aux propositions présentées dans l'état de l'art concernant les FFS au chapitre 2 ;

Automatisation de l'extraction de paramètres On projette de développer, sur la base des outils de trace des performances présentés au chapitre 3, une suite d'outils permettant d'automatiser l'extraction de paramètres de performances pour une plate-forme matérielle / logicielle donnée. Ces outils pourront par la suite être distribués, ce qui est très utile pour l'intégration dans le simulateur d'un catalogue de profils représentants de multiples plate-formes réelles.

Niveau FTL

Développement de nouveaux modèles de FTL Dans l'état actuel des choses, seule une FTL basique à base de traduction par page est modélisée. Il est important de modéliser le comportement d'autres modèles de FTLs, en particulier les propositions les plus populaires que l'on peut retrouver dans la littérature ;

Développement de modèles de performances / consommation pour FTL Au niveau FTL, le simulateur implémente uniquement un modèle théorique représentant seulement les performances et la consommation des accès à la mémoire flash. Il est nécessaire (A) de valider ce modèle théorique et (B) de construire des modèles de performances et de consommation pour les couches de gestion de type FTL ;

Évolution du simulateur

Concernant le simulateur, les perspectives d'évolution concernent l'amélioration des fonctionnalités existantes, ainsi que l'ajout de nouvelles fonctionnalités.

Amélioration des fonctionnalités existantes

Le découplage des overheads de performances / consommation des modèles fonctionnels Comme expliqué au chapitre 5, au niveau de l'implémentation l'ajout des overheads aux différents niveaux de la couche de gestion est réalisé dans le code correspondant aux modèles fonctionnels. Par souci d'évolutivité, il est nécessaire de découpler dans le simulateur ce qui correspond au calcul des performances et de la consommation du code relatif au modèles fonctionnels ;

L'optimisation des performances du simulateur lui-même Le simulateur a été développé dans le but d'être rapidement fonctionnel, et peu efforts ont été réalisés concernant les performances. L'outil supporte actuellement assez mal le passage à l'échelle (simulation sur un nombre de fichiers importants ou des fichiers de grande taille). Réduire le temps d'exécution d'une simulation est très important. Une première piste de travail serait de modifier l'indexation de nombreuses structures de données, réalisée actuellement via des tableaux, pour y introduire des arbres. C'est par exemple le cas de la liste des nodes composant un fichier JFFS2, qui impacte fortement le temps de simulation lorsqu'un fichier de grande taille est très fragmenté.

Ajout de nouvelles fonctionnalités

Prise en comtpe de l'état de départ Il est nécessaire d'implémenter la phase de *warm-up*, par exemple en intégrant au simulateur les propositions réalisées au chapitre 5 (voir page 181) ;

Développement d'un générateur de trace synthétique Il s'agit d'une fonctionnalité offerte par certains simulateurs existants. On peut imaginer le développement de générateurs synthétiques pour FFS (trace niveau appel systèmes) et FTL (niveau bloc). Concernant le niveau bloc il est potentiellement possible de ré-utiliser un générateur existant, comme c'est le cas pour *FlashSim* (Kim et coll., 2009b) qui ré-utilise le générateur de *DiskSim* (Bucy et coll., 2008). DiskSim est le simulateur de disques dur le plus utilisé ;

Amélioration de la documentation du simulateur Il s'agit d'une étape importante. La documentation de l'outil doit être réalisée en deux volets : une partie *utilisateur* pour la configuration et le lancement de simulations, une autre *développeur* pour le développement de nouvelles couche de gestion ;

Développement d'un enregistreur de trace niveau appel système On envisage d'adapter les outils de trace de performances développés dans le cadre de cette thèse pour qu'ils puissent procéder à l'enregistrement d'une trace compatible avec le format d'entrée du simulateur (en mode FFS) pour une application s'exécutant sur une plate-forme réelle. La difficulté de "rédiger" une trace complexe ou de grande taille à la main (rédaction d'un fichier au format présenté page 154) est la raison pour laquelle une validation via macro-benchmark n'est pas encore réalisée concernant les modèles.

ANNEXE A

Modèles opérationnels, de performances et de consommation pour une architecture flash complexe

1 Introduction

Dans cette annexe on présente l'intégralité des modèles opérationnels concernant les opérations avancées, présentés globalement au chapitre 4. On présente également les modèles de performances et consommation associés. Les équations présentées ici sont utilisées pour calculer les temps d'exécution et énergie consommée pour les opérations flash suivantes :

1. lecture et écriture *legacy*
2. copy-back ;
3. lecture et écriture en mode cache ;
4. lecture, écriture, effacements multi-plans ;
5. copy-back multi-plans, lecture et écriture en mode cache-multi-plan ;
6. opération de type LUN-entrelacées ;
7. opération multi-canaux.

Pour ce faire on utilise les variables et constantes définies dans la Table A.1.

On fait les hypothèses suivantes :

— Le temps d'envoi des commandes (*opcodes*) et adresses sont considérés comme négligeables et ne sont pas modélisés ;

— Le transfert d'une page sur le bus d'E/S (IO) prend un temps fixe, indépendamment du sens (hôte vers puce ou puce vers hôte) ;

— TIN et TON peuvent être variables, notamment en fonction de l'indice de page accédée sur puce flash MLC. Ainsi, on utilisera des termes tels que TIN_{addr} et TON_{addr}.

Sauf mention contraire, l'unité de temps est la microseconde. L'unité de puissance est le Watt, et l'énergie est en microjoules.

2 Opérations *legacy*

2.1 Lecture *legacy*

La commande de lecture *legacy* permet de lire une page flash. C'est une commande flash de base. Il n'y a pas de contraintes particulière quant à son utilisation, mis à part le fait que la page adressée doit être dans l'espace d'adressage de l'architecture flash concernée. Cette contrainte relative à l'espace d'adressage est bien entendu commune à toutes les opération flash, et ne sera pas répétée. L'opération de lecture de page lit l'intégralité de la page, c'est à dire les données utilisateur et OOB. La décomposition de l'exécution d'une lecture en mode *legacy* en sous-opérations est présentée sur la figure A.1.

nom	description	unité	variable ?
TIN	*Transfer In Nand*, temps d'exécution du transfert d'une page depuis le page register (ou cache register en mode cache) vers la matrice NAND.	μs	oui
TON	*Transfer Out of Nand*, temps d'exécution du transfert d'une page depuis la matrice NAND vers le page (cache) register.	μs	oui
IO	Temps d'exécution du transfert d'une page sur le bus d'E/S entre l'hôte et la ou les puce(s) flash.	μs	non
$BERS$	*Block ERaSe*, temps d'exécution d'un effacement de bloc flash.	μs	non
$PTIN$	Puissance moyenne constatée pendant TIN.	W	non
$PTON$	Puissance moyenne constatée pendant TON.	W	non
PIO	Puissance moyenne constatée pendant IO.	W	non
$PBERS$	Puissance moyenne constatée pendant $BERS$.	W	non

TABLE A.1 – Variables et constantes briques de base des modèles

a) Performance

Le temps d'exécution de cette opération est modélisé comme suit :

$$T_{LegacyRead}(addr) = TON_{addr} + IO \tag{A.1}$$

$addr$ est l'adresse de la page lue, donc TON_{addr} représente le temps de transfert de cette page depuis la matrice de transistor NAND vers le page register.

b) Consommation

L'énergie consommée lors d'une opération de lecture flash est modélisé comme suit :

$$E_{LegacyRead}(addr) = PTON * TON_{addr} + PIO * IO \tag{A.2}$$

2.2 Ecriture *legacy*

La commande d'écriture *legacy* permet d'écrire dans une page flash complète (données utilisateurs + OOB). Les contraintes suivantes s'appliquent :

1. On ne peut écrire dans une page contenant des données (précédemment écrite), il faut au préalable effacer le bloc la contenant dans son intégralité ;
2. Les écritures de pages dans un bloc doivent être séquentielles (de la page 0 à la dernière page dans le bloc) pour éviter des perturbations dues à la mauvaise fiabilité de la mémoire flash NAND.

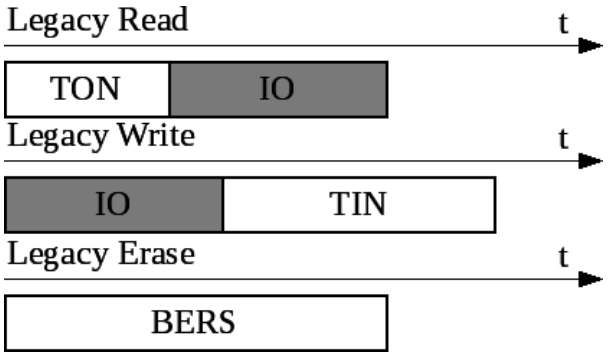


Figure A.1 – Exécution des opérations *legacy* : lecture

Les contraintes ci-dessus sont les contraintes principales des mémoires flash NAND. Dans le reste de ce document, elles sont valable pour toute opération avancée mettant en œuvre des écritures de pages flash. L'exécution d'une d'écriture en mode *legacy* décomposée en plusieurs sous-opérations est présentée sur la figure A.1.

a) Performance

$$T_{LegacyWrite}(addr) = IO + TIN_{addr} \tag{A.3}$$

b) Consommation

$$E_{LegacyWrite}(addr) = PIO * IO + PTIN * TIN_{addr} \tag{A.4}$$

2.3 Effacement *legacy*

L'opération d'effacement est réalisée sur un bloc entier. Aucune contrainte particulière ne s'applique. Toutes les pages contenues dans le bloc cibles sont effacées et passent dans l'état libre (prêtes à être écrites) La décomposition d'une opération d'effacement en mode *legacy* en sous-opérations est présentée sur la figure A.1.

a) Performance

$$T_{LegacyErase} = BERS \tag{A.5}$$

b) Consommation

$$E_{LegacyErase} = PBERS * BERS \tag{A.6}$$

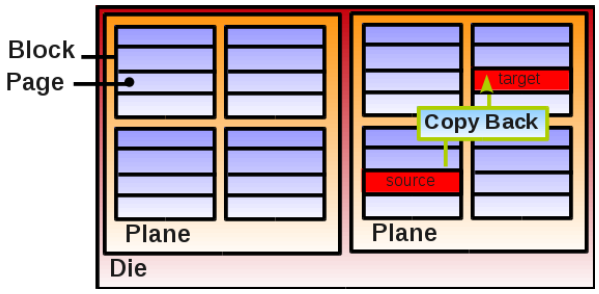


FIGURE A.2 – Un exemple d'opération copy-back

3 Copy-back et mode cache

3.1 Copy-Back

L'opération avancée *copy-back* peut également être nommée *internal data move*. Cette opération consite à copier le contenu d'une page dans une autre au sein d'un même plan, en utilisant le page buffer, sans occuper le bus d'E/S. Les contraintes suivantes s'appliquent :

1. Les indices des pages source (lue) et cible (écrite) dans leur bloc contenants doivent être tous deux pairs, ou tous deux impairs ;
2. La source et la cible doivent être dans le même plan ;
3. Les contraintes relatives à l'opération d'écriture s'appliquent à la page cible.

La décomposition de l'exécution d'une opération copy-back en sous-opérations est présentée sur la figure A.3.

a) Performance

$$T_{CopyBack}(src, dest) = TON_{src} + TIN_{dest} \tag{A.7}$$

src et $dest$ sont les adresses des pages sources et cibles de l'opération.

b) Consommation

$$E_{CopyBack}(src, dest) = PTON * TON_{src} + PTIN * TIN_{dest} \tag{A.8}$$

3.2 Lecture en mode cache

La commande cache read est une commande avancée rendue possible via l'introduction d'un *cache buffer* de la taille d'une page flash en plus du *page buffer* au sein d'un plan d'une puce flash. Ces deux buffers sont reliés. Il est alors possible, lorsque l'on lit plusieurs pages dont les indices sont physiquement séquentiels, de pipeliner les transferts (A) depuis la matrice NAND dans le cache buffer et (B) depuis le page buffer sur le bus d'E/S vers l'hôte. Cela permet une amélioration des performances en lecture séquentielle. La contrainte suivante s'applique : Les pages doivent être lues de manière séquentielle, du point de vue de leur indice ou adresses physiques. Un exemple de lecture en mode cache est représenté sur la gauche de la figure A.4.

La décomposition de l'exécution d'une opération de lecture en mode cache en plusieurs sous-opérations est présentée sur la figure A.3.

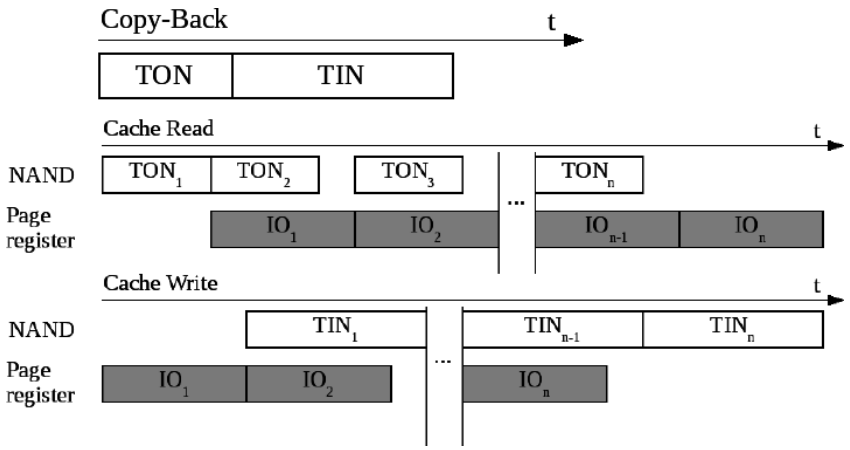


Figure A.3 – Exécution des opérations copy-back, lecture et écriture en mode cache

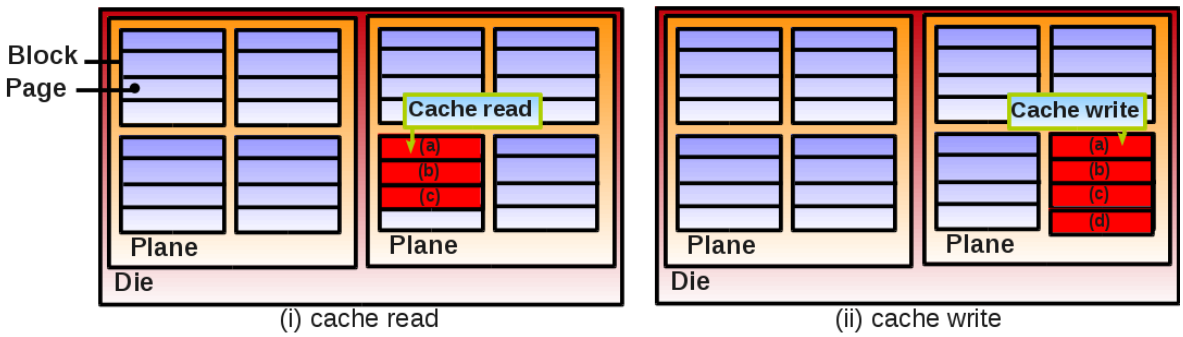


Figure A.4 – Exemple de lecture (gauche) et écriture (droite) en mode cache.

a) Performance

$$T_{CacheRead}(addr_1, addr_2, ..., addr_n) = TON_{page_1} + \sum_{i=2}^{n} max(TON_{addr_i}, IO) + IO \quad \text{avec } n \geq 2 \tag{A.9}$$

b) Consommation

$$E_{CacheRead}(addr_1, addr_2, ..., addr_n) = \sum_{i=1}^{n} (PTON * TON_{addr_i} + PIO * TIO) \quad \text{avec } n \geq 2 \tag{A.10}$$

$addr_1, ..., addr_n$ sont les adresses des différentes pages lues séquentiellement en mode cache. A noter que n doit être supérieur ou égal à 2.

3.3 Écriture en mode cache

L'opération d'écriture en mode cache fonctionne de manière similaire à la lecture. Outre les contraintes relatives à l'opération de lecture, l'écriture en mode cache est sujette aux contraintes relatives à l'écriture de page flash, et ce concernant l'intégralité des pages touchées par l'opération.

Une exemple d'opération d'écriture en mode cache est présenté sur la droite de la figure A.4. Sa décomposition en sous-opérations est illustrée sur la figure A.3.

a) Performance

$$T_{CacheWrite}(addr_1, addr_2, ..., addr_n) = IO + \sum_{i=2}^{n} max(TIN_{addr_i}, IO) + TIN_{addr_n} \quad \text{avec } n \geq 2 \tag{A.11}$$

b) Consommation

$$E_{CacheWrite}(addr_1, addr_2, ..., addr_n) = \sum_{i=1}^{n} (PTIN * TIN_{addr_i} + PIO * TIO) \quad \text{avec } n \geq 2 \tag{A.12}$$

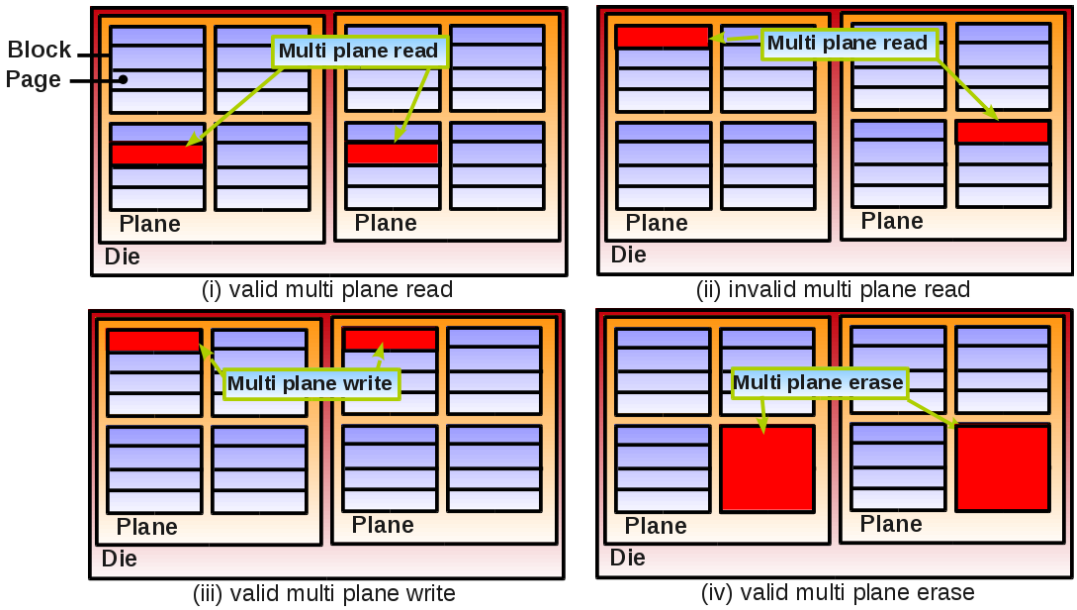


Figure A.5 – Exemples d'opération multi-plans : une lecture multi-plan valide (i), invalide (ii), une écriture multi-plans valide (iii) et un effacement multi-plans valide (iv).

4 Opérations multi-plans

4.1 Lecture multi-plans

La lecture multi-plans permet de lire différentes pages flash dans plusieurs plan d'une même LUN. Les transferts depuis la matrice NAND vers les page buffers sont réalisés en parallèle dans chaque plan ciblé. De multiples contraintes s'appliquent à cette opération (et aux opérations multi-plans en général) :

1. Les pages ciblées doivent chacune être dans un plan différent d'une même LUN ;
2. L'adresse de la page à lire dans chaque plan est représentée par un couple (*indice de bloc dans le plan, indice de page dans le bloc*). Toutes les pages ciblées par une opération multi-plan doivent avoir le même couple d'adresses (*bloc*, *page*) dans le plan ciblé ;
3. L'opération *multi-plans* est effectuée sur *tous* les plans d'une même LUN.

Des exemples d'opérations multi-plans valide et invalide sont présentées dans la partie supérieure de la figure A.5. En haut à gauche est représentée l'opération valide, car le couple (*bloc*, *page*) dans chaque plan ciblé est le même. En haut à droite on peut voir une opération invalide, non réalisable, car l'adresse dans chaque plan est différente.

La décomposition d'une d'opération de lecture multi-plans est présenté sur la figure A.6.

a) Performance

On définit T_{P_i} comme étant le temps d'une opération de lecture d'une page dans le plan i.

$$\begin{aligned} T_{P_1}(addr_1) &= TON_{addr_1} + IO \\ T_{P_i}(addr_i) &= max(T_{P_{i-1}}, TON_{addr_i}) + IO \end{aligned} \tag{A.13}$$

$addr_i$ est l'adresse de la page à lire dans le plan i. Ainsi :

$$\begin{aligned} T_{MultiPlaneRead}(addr_1, addr_2, ..., addr_n) &= max_i(T_{P_i}) \\ &\text{avec } n \geq 2 \end{aligned} \tag{A.14}$$

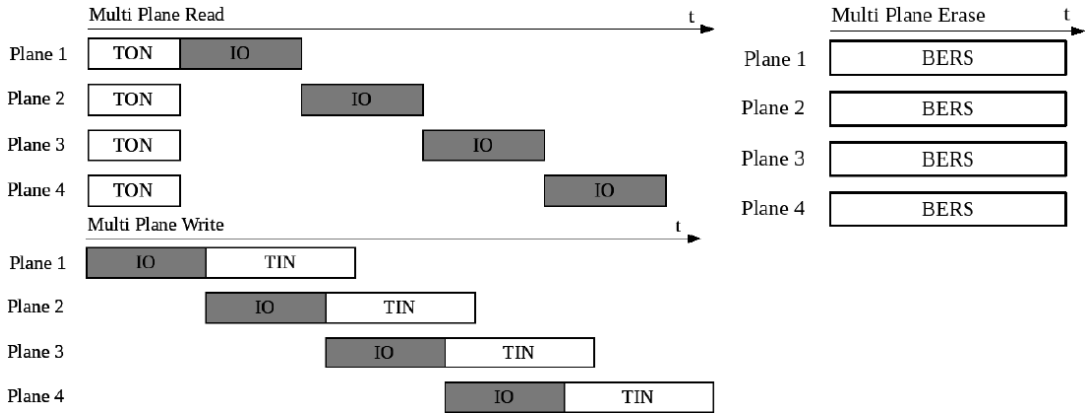


FIGURE A.6 – Exécution des opérations de lecture, écriture et effacement multi-plans

n est le nombre de plans participants à l'opération multi-plan.

b) Consommation

$$E_{MultiPlaneRead}(addr_1, addr_2, ..., addr_n) = \sum_{i=1}^{n}(PTON * TON_{addr_i}) + n * PIO * IO \quad \text{avec } n \geq 2 \tag{A.15}$$

4.2 Écriture multi-plans

L'opération d'écriture multi-plans fonctionne de manière similaire à l'opération de lecture. Outre les contraintes propres aux opérations multi-plans, les contraintes relatives à l'opération d'écriture flash s'appliquent sur chaque page écrite. Un exemple d'opération d'écriture multi-plans est présenté sur la figure A.5. Sa décomposition en sous-opérations est présentée sur la figure A.6

a) Performance

T_{P_i} est le temps d'une opération d'écriture de page dans le plan i. Il est définit comme suit :

$$T_{P_i}(addr_i) = i * IO + TIN_{addr_i} \tag{A.16}$$

$addr_i$ est l'adresse de la page à écrire dans le plan i. Ainsi :

$$\begin{aligned} T_{MultiPlaneWrite}(addr_1, addr_2, ..., addr_n) &= max_i(T_{P_i}) \\ &= max_i(i * IO + TIN_{addr_i}) \\ &\text{avec } n \geq 2 \end{aligned} \tag{A.17}$$

n est le nombre de plans participants à l'opération mutli-plan.

b) Consommation

$$E_{MultiPlaneWrite}(addr_1, addr_2, ..., addr_n) = n * PIO * IO + \sum_{i=1}^{n}(PTIN * TIN_{addr_i}) \quad \text{avec } n \geq 2 \tag{A.18}$$

4.3 Effacement multi-plans

L'effacement multi-plans permet d'effacer en parallèle plusieurs blocs au sein de plans différents d'une même LUN. Les contraintes multi-plans s'appliquent, en particulier le fait que les blocs ciblés doivent avoir le même indice dans chaque plan participant à l'opération. Un exemple d'effacement multi-plans est présenté sur la figure A.6, la décomposition de cette opération en sous opération est illustrée sur la figure A.6.

a) Performance

$$T_{MultiPlaneErase}(addr_1, addr_2, ..., addr_n) = BERS \quad \text{avec } n \geq 2 \tag{A.19}$$

n est le nombre de plans participants à l'opération mutli-plan.

b) Consommation

$$E_{MultiPlaneErase}(addr_1, addr_2, ..., addr_n) = n * PBERS * BERS \quad \text{avec } n \geq 2 \tag{A.20}$$

4.4 Copy-back multi-plan

Cette opération permet d'effectuer en parallèle plusieurs opérations copy-back dans chaque plan d'une même LUN. Les contraintes suivantes s'appliquent :

1. Chaque opération copy-back doit être réalisée dans un plan différent d'une même LUN ;
2. Les pages sources de chaque opération copy-back doivent toutes avoir le même couple d'adresses (*bloc*, *page*) au sein de chaque plan concerné ;
3. Idem que le point ci-dessus pour les pages cibles ;
4. Les opérations de copy-back sont exécutées dans chaque plan d'une même LUN ;
5. Les indices au sein des blocs contenants pour les pages cible et sources de chaque copy-back doivent être tous deux pairs, ou tous deux impairs ;
6. Les contraintes relatives à l'écriture flash s'appliquent pour chaque page cible des opérations copy-back.

Un exemple d'opération copy-back multi-plans est illustrée sur la Figure A.7.

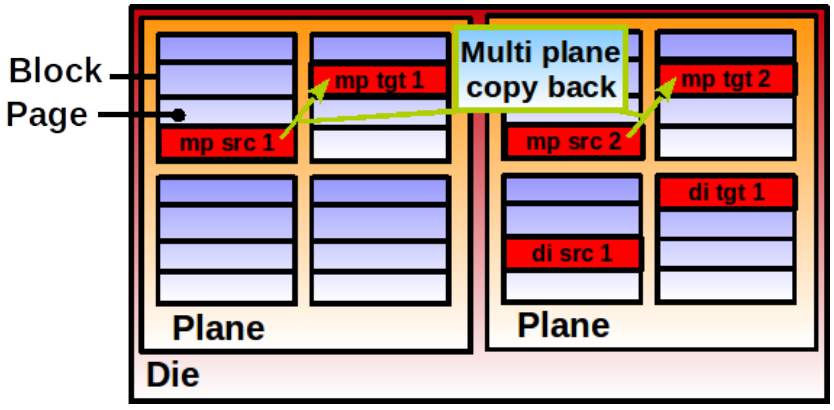


FIGURE A.7 – Exemple d'opération copy-back multi-plans

a) Performance

$$T_{MultiPlaneCopyBack}(src_1, src_2, ..., src_n, dest_1, dest_2, ..., dest_n) = max_i(TON_{src_i} + TIN_{dest_i}) \quad \text{avec } n \geq 2 \tag{A.21}$$

n est le nombre de plans participants à l'opération mutli-plan.

b) Consommation

$$E_{MultiPlaneCopyBack}(src_1, src_2, ..., src_n, dest_1, dest_2, ..., dest_n) = \sum_{i=1}^{n}(PTON * TON_{src_i} + PTIN * TIN_{dest_i}) \quad \text{avec } n \geq 2 \tag{A.22}$$

4.5 Lecture et écriture multi-plans en mode cache

Les opérations de lecture et écriture multi-plans en mode cache permettent de lire et écrire des pages de manière séquentielle, dans tous les plans d'une même LUN. Les contraintes relatives aux opérations (A) multi-plans et (B) en mode cache s'appliquent. De plus, pour l'opération d'écriture multi-plan en mode cache, les contraintes relatives à l'écriture flash s'appliquent pour chacune des pages à écrire. Des exemples de ces opérations sont présentés sur la figure A.8.

Les opérations de lecture et écriture multi-plans peuvent être définies comme des ensembles de listes d'adresses. Chaque liste représente une opération de lecture / écriture en mode cache. Il y a une liste par plan participant à l'opération multi-plans en mode cache. Chaque liste contient les adresses des pages à lire / écrire séquentiellement via une opération en mode cache. Prenons par exemple une opération multi-plans en mode cache ciblant 4 plans, chacun subissant une lecture ou écriture séquentielle de 3 pages. On peut décrire cette opération ainsi :

$$[addr_{1,1}, addr_{1,2}, addr_{1,3}, addr_{2,1}, addr_{2,2}, addr_{2,3}, addr_{3,1}, addr_{3,2}, addr_{3,3}, addr_{4,1}, addr_{4,2}, addr_{4,3}] \tag{A.23}$$

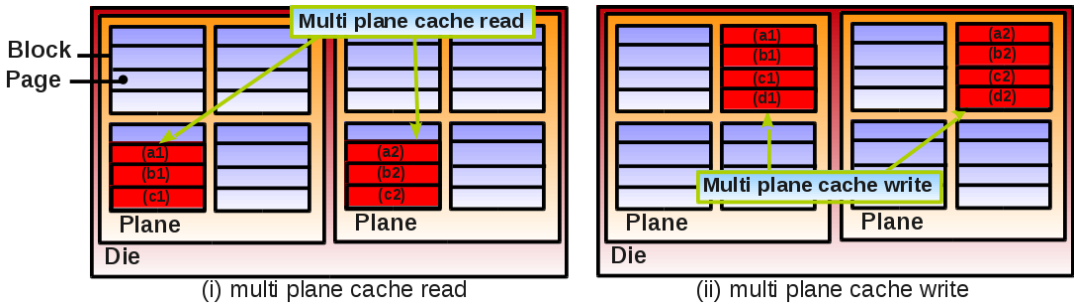


Figure A.8 – Exemples d'opérations multi-plans en mode cache

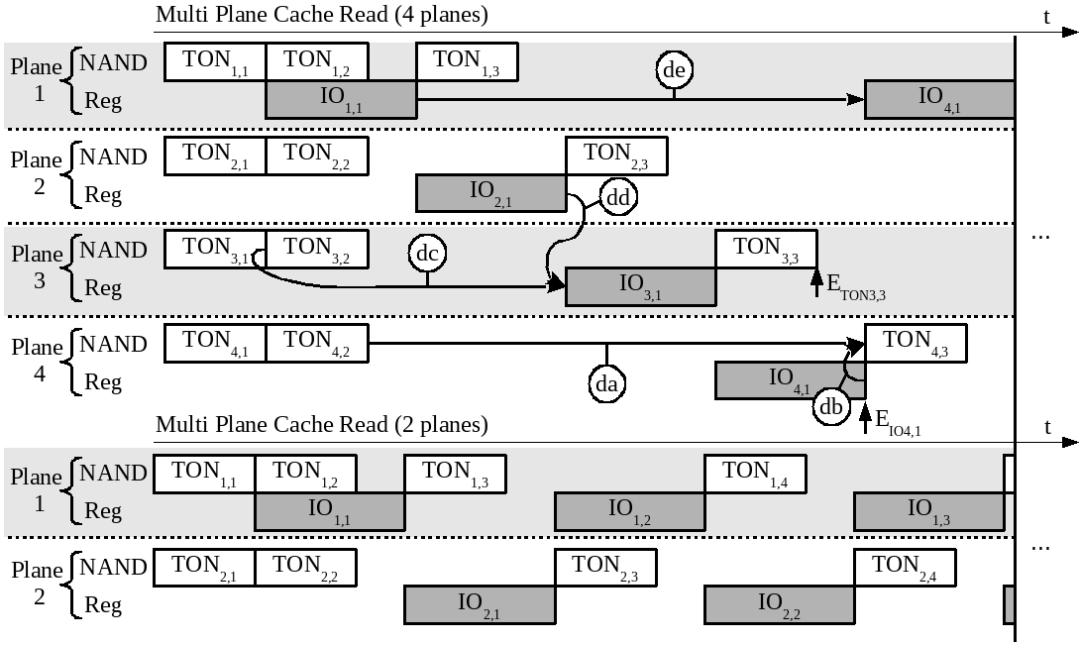


Figure A.9 – Exécution d'une opération de lecture multi-plans en mode cache

$addr_{i,j}$ représente l'adresse de la j-ème page à lire / écrire dans le plan i. Il y a un $TON_{i,j}$ et un $IO_{i,j}$ relatif à chaque opération de lecture de page. De la même façon, il y a un $IO_{i,j}$ et un $TIN_{i,j}$ pour chaque page à écrire. Ces valeurs sont présentées dans l'exemple d'éxécution des opérations de lecture et écriture multi-plans en mode cache, sur les figures A.9 et A.10. A noter que IO est une constante, ainsi chaque $IO_{i,j}$ est égal à IO (sauf lorsque $addr_{i,j}$ est $none$, comme expliqué plus bas). $TON_{i,j}$ et $TINi, j$ sont des variables.

Dans le cas où le nombre de pages à lire / écrire séquentiellement est différent d'un plan à l'autre, la taille de chaque liste est égale à la taille de la plus grande liste. On introduit le terme $none$, utilisé pour remplir les "trous" dans les listes de tailles inférieures. Prenons l'exemple d'une opération multi-plans en mode cache comprenant 4 pages accédées séquentiellement dans le premier plan, et seulement deux pages accédées séquentiellement dans le second plan. Cette opération est représentée comme suit :

$$[addr_{1,1}, addr_{1,2}, addr_{1,3}, addr_{1,4}, addr_{2,1}, addr_{2,2}, addr_{2,3} = none, \\ addr_{2,4} = none] \tag{A.24}$$

Les termes $none$ sont utiles dans le calcul des performances et de la consommation : on définit

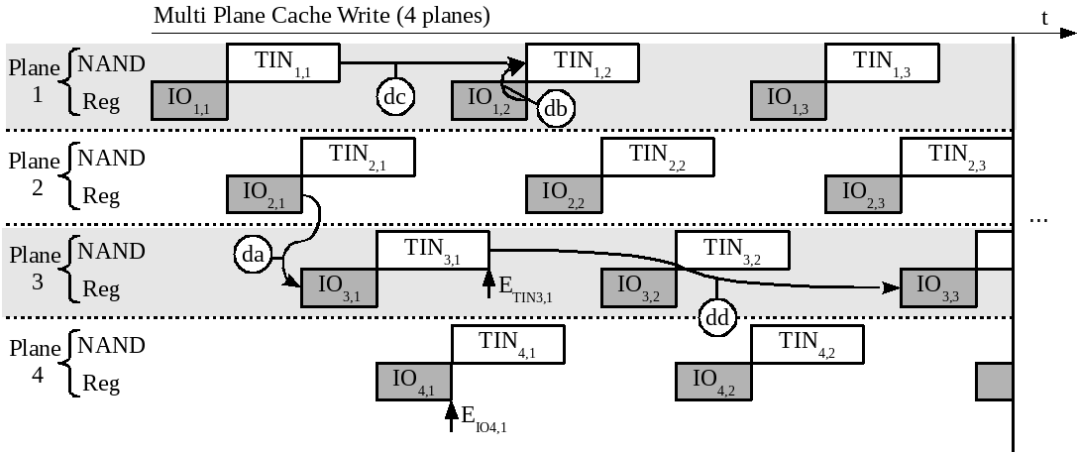


FIGURE A.10 – Exécution d'une opération d'écriture multi-plans en mode cache

les valeurs TON, IO et TIN pour une adresse égale à $none$ à 0 :

$$TON_{i,j} = \begin{cases} TON_{addr_{i,j}} & \text{si } addr_{i,j} \neq none \\ 0 & \text{sinon} \end{cases} \tag{A.25}$$

$$TIN_{i,j} = \begin{cases} TIN_{addr_{i,j}} & \text{si } addr_{i,j} \neq none \\ 0 & \text{sinon} \end{cases} \tag{A.26}$$

$$IO_{i,j} = \begin{cases} IO & \text{si } addr_{i,j} \neq none \\ 0 & \text{sinon} \end{cases} \tag{A.27}$$

On définit également, concernant les opérations de lecture / écriture multi-plans en mode cache, les termes $E_{TON_{i,j}}$, $E_{IO_{i,j}}$ et $E_{TIN_{i,j}}$. Ce sont respectivement les moments où les opérations TON, IO et TIN se terminent, pour une page donnée. $E_{TON_{3,3}}$ et $E_{IO_{4,1}}$ sont représentés sur la figure A.9. $E_{TIN_{3,1}}$ est représenté sur la figure A.10.

Dans les équations suivantes, n correspond au nombre de plans mis en œuvre dans l'opération de lecture / écriture multi-plans en mode cache, et m_i est le nombre de pages lues / écrites dans le plan i. m_{max} est le nombre de pages lues / écrites dans le plan contenant le nombre le plus important de pages accédées. Pour notre exemple représenté dans l'équation A.24, $n = 2$, $m_1 = 4$, $m_2 = 2$, et $m_{max} = max_i(m_i) = m_1 = 4$.

a) Lecture multi-plans en mode cache - Performance

$E_{TON_{i,j}}$ dépend des valeurs suivantes :

1. $E_{IO_{i,j-2}}$ (db sur la figure A.9) : le cache buffer contenant la page $(i, j-2)$ doit en premier lieu être vidé. Il est par la suite remplit avec la page $(i, j-1)$, venant du page buffer. Pendant que le page buffer est vidé, $TON_{i,j}$ peut avoir lieu ;
2. $E_{TON_{i,j-1}}$ (da sur la figure A.9) ;
3. $TON_{i,j}$.

$E_{IO_{i,j}}$ dépend de :

1. $E_{IO_{i-1,j}}$ (dd sur la figure A.9) : le bus d'E/S doit être libre ;
2. $E_{TON_{i,j}}$ (dc sur la figure A.9) : la page à envoyer sur le bus d'E/S doit être disponible dans le page buffer ;

3. $E_{IO_{i,j-1}}$ (de sur la figure A.9) ;
4. $IO_{i,j}$.

Ainsi, on peut modéliser le temps d'exécution d'une opération de lecture multi-plans en mode cache comme suit :

$$\begin{aligned} E_{IO_{1,1}} &= TON_{1,1} + IO \\ E_{IO_{1,j}} &= max(E_{IO_{n,j-1}}, E_{TON_{1,j}}, E_{IO_{i,j-1}}) + IO && \text{avec } j \neq 1 \\ E_{IO_{i,j}} &= max(E_{IO_{i-1,j}}, E_{TON_{i,j}}, E_{IO_{i,j-1}}) + IO && \text{avec } j \neq 1 \text{ et } i \neq 1 \end{aligned} \tag{A.28}$$

$$\begin{aligned} E_{TON_{i,1}} &= TON_{i,1} \\ E_{TON_{i,2}} &= E_{TON_{i,1}} + TON_{i,2} \\ E_{TON_{i,j}} &= max(E_{IO_{i,j-2}}, E_{TON_{i,j-1}}) + TON_{i,j} && \text{avec } j > 2 \end{aligned} \tag{A.29}$$

Finalement :

$$\begin{aligned} T_{MultiPlaneCacheRead}(addr_{1,1}, addr_{1,2}, ..., addr_{1,m_{max}}, \\ addr_{2,1}, ..., addr_{n,m_{max}}) = max_i(E_{IO_{i,m_{max}}}) \\ \text{avec } n \geq 2 \text{ et } m_{max} \geq 2 \end{aligned} \tag{A.30}$$

b) Lecture multi-plans en mode cache - Consommation

$$\begin{aligned} E_{MultiPlaneCacheRead}(addr_{1,1}, addr_{1,2}, ..., addr_{1,m_{max}}, \\ addr_{2,1}, ..., addr_{n,m_{max}}) = \sum_{i=1}^{n}[\sum_{j=1}^{m_{max}}(TON_{addr_{i,j}} * PTON + IO_{i,j} * PIO)] \\ \text{avec } n \geq 2 \text{ et } m_{max} \geq 2 \end{aligned} \tag{A.31}$$

c) Écriture multi-plans en mode cache - Performance

$E_{TIN_{i,j}}$ dépend de :

1. $E_{TIN_{i,j-1}}$ (dc sur la figure A.10) ;
2. $E_{IO_{i,j}}$ (db sur la figure A.10) : le transfert sur le bus d'E/S depuis l'hôte doit être terminé pour pouvoir charger la page dans la matrice NAND ;
3. $TIN_{i,j}$.

$E_{IO_{i,j}}$ dépend de :

1. $E_{IO_{i-1,j}}$ (da sur la figure A.10) : le bus d'E/S doit être libre pour le transfert de la page (i, j) ;
2. $E_{TIN_{i,j-2}}$ (dd sur la figure A.10) : lorsque l'écriture de la page $(i, j-2)$ dans la matrice NAND est complète, le page buffer est vide. Ainsi, la page $(i, j-1)$ peut y être placée venant du cache buffer. Ensuite, le cache buffer étant libre, on peut effectuer $TIN_{i,j}$;
3. $IO_{i,j}$.

On peut modéliser le temps d'exécution d'une opération d'écriture multi-plans en mode cache comme suit :

$$\begin{aligned}
E_{IO_{1,1}} &= IO_{i,j} \\
E_{IO_{1,j}} &= E_{IO_{n,j-1}} + IO_{i,j} && \text{avec } 2 \leq j \leq 3 \\
E_{IO_{1,j}} &= max(E_{IO_{n,j-1}}, E_{TIN_{1,j-2}}) + IO_{i,j} && \text{avec } j > 3 \\
E_{IO_{i,j}} &= E_{IO_{i-1,j}} + IO_{i,j} && \text{avec } i \neq 1 \text{ et } 2 \leq j \leq 3 \\
E_{IO_{i,j}} &= max(E_{IO_{i-1,j}}, E_{TIN_{i,j-2}}) + IO_{i,j} && \text{avec } i \neq 1 \text{ et } j > 3
\end{aligned} \tag{A.32}$$

$$\begin{aligned}
E_{TIN_{1,1}} &= E_{IO_{1,1}} + TIN_{1,1} \\
E_{TIN_{i,j}} &= max(E_{IO_{i,j}}, E_{TIN_{i,j-1}}) + TIN_{i,j} && \text{avec } i \neq 1 \text{ et } j \neq 1
\end{aligned} \tag{A.33}$$

Une fois de plus ici n est le nombre de plans participants à l'opération. Finalement nous avons :

$$\begin{aligned}
T_{MultiPlaneCacheWrite}(addr_{1,1}, addr_{1,2}, ..., addr_{1,m_{max}}, \\
addr_{2,1}, ..., addr_{n,m_{max}}) = max_i(E_{TIN_{i,m_{max}}}) \\
\text{avec } n \geq 2 \text{ et } m_{max} \geq 2
\end{aligned} \tag{A.34}$$

d) Écriture multi-plans en mode cache - Consommation

$$\begin{aligned}
E_{MultiPlaneCacheWrite}(addr_{1,1}, addr_{1,2}, ..., addr_{1,m_{max}}, \\
addr_{2,1}, ..., addr_{n,m_{max}}) = \sum_{i=1}^{n}[\sum_{j=1}^{m_{max}}(TIN_{addr_{i,j}} * PTIN + IO_{addr_{i,j}} * PIO)] \\
\text{avec } n \leq 2 \text{ et } m_{max} \leq 2
\end{aligned} \tag{A.35}$$

5 Opérations multi-canaux

Les opérations multi-canaux sont purement parallèles : il n'y a pas de dépendance entre les différentes opérations composant une opération multi-canaux.

e) Performance

$$\begin{aligned}
T_{MultiChannel}(cmd_1, cmd_2, ..., cmd_n) = max_i(T_{cmd_i}) \\
\text{avec } n \geq 2
\end{aligned} \tag{A.36}$$

n est le nombre de canaux participants à l'opération.
$cmd_1, ..., cmd_n$ sont les différentes commandes envoyées en parallèle aux différents canaux. T_{cmd_i} est le temps d'exécution de cmd_i, calculé en utilisant l'une des formules présentée précédemment.

f) Consommation

$$\begin{aligned}
E_{MultiChannel}(cmd_1, cmd_2, ..., cmd_n) = \sum_{i=1}^{n} E_{cmd_i} \\
\text{avec } n \geq 2
\end{aligned} \tag{A.37}$$

E_{cmd_i} représente l'énergie consommée pendant l'exécution de cmd_i, calculée via l'une des formules présentées ci-dessus.

ANNEXE B

Validation fonctionnelle de read-ahead, read-ahead et Linux version pré-3.13

1 Validation fonctionnelle de read-ahead

1.1 Introduction

La méthodologie de validation est présentée au chapitre 5 page 190. Les scénarios de validation y sont également présentés, page 195. Chacune des métriques de validations (voir page 196) se voit assignée un nom :

1. Le nombre de page cache hits : *cache_hits* ;
2. La liste contenant les index des pages demandées par le programme de test qui débouchent sur un appel à la fonction noyau page_cache_async_readahead() : *async_indexes* ;
3. Idem que ci-dessus pour les appels à read-ahead en mode synchrone : *sync_indexes* ;
4. Le nombre total d'appels à read-ahead : *nb_pass* ;
5. La liste contenant, pour chaque passe de read-ahead, (synchrone ou asychrone), les tailles de fenêtre calculée par chacune des passes : *size* ;
6. La liste des index de pages Linux sur lesquelles read-ahead pose le flag read-ahead : *async_size* ;
7. Le nombre de pages Linux du fichier cible lues via le FFS : *ffs_readpages* ;
8. Le nombre d'appels à vfs_read : *vfs_read*.

2 Résultats

Pour les scénarios générant des logs (1. de mesures et 2. de simulation) relativement courts, ces logs sont inclus dans ce document. Pour les scénarios 4-9 il est impossible d'inclure les logs, car ils sont trop importants. Cela est vrai également pour certaines métriques de validation. Par exemple les listes des tailles de fenêtres read-ahead. Comme il y a un élément par appel à read-ahead ces listes sont importantes pour les scénarios 4-9 qui sont relativement intensifs. On inclut donc dans ce document non pas les listes entières, mais plutôt le résultat de la comparaison de la liste tirée des mesures réelles avec la liste tirée d'une simulation. Cette comparaison est réalisée avec des outils automatisés.

Scénario 1 La table B.1 présente les résultats relatif au scénario 1. On peut voir que le comportement fonctionnel est identique en environnement réel (à gauche dans la table) et en simulation (à droite). Cette remarque est valable pour l'intégralité des scénarios.

Scénario 1 - Logs d'exécution	
Mesures en environnement réel	**Log de simulation** (certaines informations ont été omises pour des raisons de clarté)
0.171112000;VFS_READ; 1 pages starting from page 0 0.171193000; SYNC_RA; offset: 0, req_size: 1 0.171233000; ONDEMAND_RA; offset: 0, req_size: 1 0.171271000; RA_SUBMIT; start: 0, size:4, async_size:3 0.171310000; RA_FLAGSET; offset: 0, nr_to_read: 4, lookahead_size: 3 0.171369000; FFS_READPAGE; page 0 0.171660000; FFS_READPAGE; page 1 0.171864000; FFS_READPAGE; page 2 0.172060000; FFS_READPAGE; page 3	[VFS] VfsRead ino: 2, offset:0, count:4096 [RA] Sync RA ino#2 @0 for 1 pages [RA] Ondemand Read-Ahead [RA] Start of file, reset window [RA] ino#2, Reading @0 for 4 pages [RA] async_size : 3 [JFFS2] readpage ino:2, page index:0 [JFFS2] readpage ino:2, page index:1 [RA] setting ra flag on #ino2 page 1 [JFFS2] readpage ino:2, page index:2 [JFFS2] readpage ino:2, page index:3

Scénario 1 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	0	0
async_indexes	{} (0 élément, pas d'appel à RA en asynchrone)	{} (0 élément)
sync_indexes	{0} (1 élément)	{0} (1 élément)
nb_pass	1	1
size	{ 4 } (1 élément)	{ 4 } (1 élément)
async_size	{ 3 }	{ 3 }
ffs_readpages	4	4
vfs_read	1	1

TABLE B.1 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 1.

Scénario 2 Le scénario 2 (lecture de 512 octets @0) donne exactement les mêmes résultats que le scénario 1 (lecture de 4096 octets @0). C'est tout à fait normal, car la lecture de 512 octets déclenche la lecture le la page complète contenant les données demandées (donc lecture de 4KO @0). La table B.2 présente les résultats relatif au scénario 2.

Scénario 2 - Logs d'exécution	
Mesures en environnement réel	**Log de simulation**
0.994190000;VFS_READ; 1 pages starting from page 0 0.994231000; SYNC_RA; offset: 0, req_size: 1 0.994252000; ONDEMAND_RA; offset: 0, req_size: 1 0.994278000; RA_SUBMIT; start: 0, size:4, async_size:3 0.994295000; RA_FLAGSET; offset: 0, nr_to_read: 4, lookahead_size: 3 0.994345000; FFS_READPAGE; page 0 0.994837000; FFS_READPAGE; page 1 0.995100000; FFS_READPAGE; page 2 0.995301000; FFS_READPAGE; page 3	[VFS] VfsRead ino: 2, offset:0, count:512 [RA] Sync RA ino#2 @0 for 1 pages [RA] Ondemand Read-Ahead [RA] Start of file, reset window [RA] ino#2, Reading @0 for 4 pages [RA] async_size : 3 [JFFS2] readpage ino:2, page index:0 [JFFS2] readpage ino:2, page index:1 [RA] setting ra flag on #ino2 page 1 [JFFS2] readpage ino:2, page index:2 [JFFS2] readpage ino:2, page index:3

Scénario 2 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	0	0
async_indexes	{} (0 élément, pas d'appel à RA en asynchrone)	{} (0 élément)
sync_indexes	{0} (1 élément)	{0} (1 élément)
nb_pass	1	1
size	{ 4 } (1 élément)	{ 4 } (1 élément)
async_size	{ 3 }	{ 3 }
ffs_readpages	4	4
vfs_read	1	1

Table B.2 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 2.

Scénario 3 La table B.3 présente les résultats relatif au scénario 3.

Scénario 3 - Logs d'exécution	
Mesures en environnement réel	**Log de simulation**
1.800666000;VFS_READ; 5 pages starting from page 0 1.800724000; SYNC_RA; offset: 0, req_size: 5 1.800758000; ONDEMAND_RA; offset: 0, req_size: 5 1.800790000; RA_SUBMIT; start: 0, size:16, async_size:11 1.800816000; RA_FLAGSET; offset: 0, nr_to_read: 16, lookahead_size: 11 1.800880000; FFS_READPAGE; page 0 1.801202000; FFS_READPAGE; page 1 1.801418000; FFS_READPAGE; page 2 1.801610000; FFS_READPAGE; page 3 /* … */ 1.804499000; FFS_READPAGE; page 15	[VFS] VfsRead ino: 2, offset:0, count:20480 [RA] Sync RA ino#2 @0 for 5 pages [RA] Ondemand Read-Ahead [RA] Start of file, reset window [RA] ino#2, Reading @0 for 16 pages [RA] async_size : 11 [JFFS2] readpage ino:2, page index:0 [JFFS2] readpage ino:2, page index:1 [JFFS2] readpage ino:2, page index:2 [JFFS2] readpage ino:2, page index:3 /* … */ [JFFS2] readpage ino:2, page index:15 [VFS] PC Hit (2, 1) [VFS] PC Hit (2, 2) [VFS] PC Hit (2, 3) [VFS] PC Hit (2, 4)

Scénario 3 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	4	4
async_indexes	{} (0 élément, pas d'appel à RA en asynchrone)	{} (0 élément)
sync_indexes	{0} (1 élément)	{0} (1 élément)
nb_pass	1	1
size	{16} (1 élément)	{16} (1 élément)
async_size	{11} (1 élément)	{11} (1 élément)
ffs_readpages	16	16
vfs_read	1	1

TABLE B.3 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 3.

Scénario 4 Pour les scénarios 4 à 9, les tailles des listes *size* et *async_size* sont trop importantes pour que ces listes soient représentées dans ce rapport. On indique donc ici seulement si ces valeurs sont égales en environnement réel et en simulation. Pour ce faire ces listes ont été extraites des logs de mesures / de simulation dans des fichiers qui ont étés comparé via l'outil Linux *diff*. De plus, on ne présente pas de log de d'exécution pour le scénarios 5 à 9, car ces logs sont trop longs, et même un sous-ensemble représentatif serait lui-même trop long pour apparaître ici. La table B.4 présente les résultats relatif au scénario 4.

Scénario 4 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	5119	5119
async_indexes	{1 , 4 , 12 , 28 , …, 5052 , 5084 , 5116 } (163 éléments)	{1 , 4 , 12 , 28 , …, 5052 , 5084 , 5116 } (163 éléments)
sync_indexes	{0} (1 élément)	{0} (1 élément)
nb_pass	164	164
size	{ 4, 8, 16, 32, 32, 32, 32, …, 32} (164 éléments)	{ 4, 8, 16, 32, 32, 32, 32, …, 32} (164 éléments)
async_size	{ 3, 8, 16, 32, 32, 32, 32, …, 32} (164 éléments)	{ 3, 8, 16, 32, 32, 32, 32, …, 32} (164 éléments)
ffs_readpages	5120	5120
vfs_read	5120	5120

Table B.4 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 4.

Scénario 5 La table B.5 présente les résultats relatif au scénario 5. On peut voir par rapport à ce scénario 5 que la taille de la liste *size*, correspondant au nombre de fenêtres read-ahead calculées, est différente du nombre total de passes (*nb_pass*). En effet il y a 3 appels à *page_cache_async_readahead()*, situées en fin de scénario de test, qui sont appelées alors que le fichier est intégralement dans le page cache. Il n'y a donc rien à lire et pas de fenêtre à calculer. Ce comportement se retrouve également dans certains des scénarios suivants.

Scénario 5 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	5113	5113
async_indexes	{1 , 4 , 1285 , 2565 , 3845 , 12 , 1293, …, 5119} (169 éléments)	{1 , 4 , 1285 , 2565 , 3845 , 12 , 1293, …, 5119} (169 éléments)
sync_indexes	{0 , 1280 , 2560 , 3840 , 1281 , 2561 , 3841} (7 éléments)	{0 , 1280 , 2560 , 3840 , 1281 , 2561 , 3841} (7 éléments)
nb_pass	176	176
size	{ 4, 1, 1, 1, 8, 12, …, 32} (173 éléments)	{ 4, 1, 1, 1, 8, 12, …, 32} (173 éléments)
async_size	{ 3, 8, 8, 8, 8, 18, …, 32} (170 éléments)	{ 3, 8, 8, 8, 8, 18, …, 32} (170 éléments)
ffs_readpages	5120	5120
vfs_read	5120	5120

Table B.5 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 5.

Scénario 6 La table B.6 présente les résultats relatif au scénario 6.

Scénario 6 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	3914	3914
async_indexes	{2559, 159 , 2262 , 1734 , 628 , 2567, …, 565} (354 éléments)	{2559, 159 , 2262 , 1734 , 628 , 2567, …, 565} (354 éléments)
sync_indexes	{572 , 2574 , 4016 , 2270 , 1600 , 1210, …, 3503} (1206 éléments)	{572 , 2574 , 4016 , 2270 , 1600 , 1210, …, 3503} (1206 éléments)
nb_pass	1560	1560
size	{1 , 1 , 1 , 4 , 4 , 4, …, 32, 4} (1365 éléments)	{1 , 1 , 1 , 4 , 4 , 4, …, 32, 4} (1365 éléments)
async_size	{3, 3, 3, 3, 3, …, 32, 3} (824 éléments)	{3, 3, 3, 3, 3, …, 32, 3} (824 éléments)
ffs_readpages	5103	5103
vfs_read	5120	5120

TABLE B.6 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 6.

Scénario 7 La table B.7 présente les résultats relatif au scénario 7.

Scénario 7 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	9281	9281
async_indexes	{1900 , 1051 , 850 , 1684 , 1195, …, 66} (440 éléments)	{1900 , 1051 , 850 , 1684 , 1195, …, 66} (440 éléments)
sync_indexes	{266 , 2010 , 1049 , 3973 , 1898, …, 1630} (958 éléments)	{266 , 2010 , 1049 , 3973 , 1898, …, 1630} (958 éléments)
nb_pass	1398	1398
size	{2 , 2 , 4 , 2 , 4, …, 32, 4} (1112 éléments)	{2 , 2 , 4 , 2 , 4, …, 32, 4} (1112 éléments)
async_size	{2, 2, 2, 2, …, 32, 3} (710 éléments)	{2, 2, 2, 2, …, 32, 3} (710 éléments)
ffs_readpages	5110	5110
vfs_read	5120	5120

TABLE B.7 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 7.

Scénario 8 La table B.8 présente les résultats relatif au scénario 8.

Scénario 8 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	124	124
async_indexes	{1470 , 2742 , 2926 , 1395 , 2177 , 1336 , 1194 , 2491 , 1298 , 2362 , 1317} (11 éléments)	{1470 , 2742 , 2926 , 1395 , 2177 , 1336 , 1194 , 2491 , 1298 , 2362 , 1317} (11 éléments)
sync_indexes	{191 , 1975 , 979 , 3902 , 1817 , 1176, …, 2844} (440 éléments)	{191 , 1975 , 979 , 3902 , 1817 , 1176, …, 2844} (440 éléments)
nb_pass	451	451
size	{1 , 1 , 4 , 1 , 4, ...1, 2} (451 éléments)	{1 , 1 , 4 , 1 , 4, ...1, 2} (451 éléments)
async_size	{3, 3, 3, 3, …, 2, 3, 3} (230 éléments)	{3, 3, 3, 3, …, 2, 3, 3} (230 éléments)
ffs_readpages	1422	1422
vfs_read	500	500

TABLE B.8 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 8.

Scénario 9 La table B.9 présente les résultats relatif au scénario 9.

Scénario 9 - Comparaison des mesures de métriques de validation		
Métrique	**Env. réel**	**Simulation**
cache_hits	2686	2686
async_indexes	{2636 , 4204 , 1608 , 1968 , 2355, …, 3093} (48 éléments)	{2636 , 4204 , 1608 , 1968 , 2355, …, 3093} (48 éléments)
sync_indexes	{610 , 2170 , 1369 , 4297 , 2270, …, 471} (314 éléments)	{610 , 2170 , 1369 , 4297 , 2270, …, 471} (314 éléments)
nb_pass	362	362
size	{6 , 6 , 16 , 6 , 16 , 16 , 6 , …, 22, 16} (348 éléments)	{6 , 6 , 16 , 6 , 16 , 16 , 6 , …, 22, 16} (348 éléments)
async_size	{10, 10, 10, 10, …, 22, 10} (212 éléments)	{10, 10, 10, 10, …, 22, 10} (212 éléments)
ffs_readpages	4201	4201
vfs_read	500	500

Table B.9 – Validation fonctionnelle de read-ahead - résultats relatifs au scénario 9.

3 Read-ahead et Linux version pré-3.13

3.1 Introduction

Il existe une erreur dans le code du noyau qui conduit à un comportement incorrect de read-ahead. Cette erreur n'a été corrigée que très récemment, dans la version 3.13 du noyau, dont la révision la plus ancienne (3.13.1) date du 29 janvier 2014. Cela signifie que les versions de Linux pour les cartes que nous utilisons (Armadeus : 2.6.29/2.6.38 ; Omap : 2.6.37) sont sujettes à cette erreur. Plus globalement, on peut raisonnablement penser qu'une grande partie des systèmes embarqués actuels tournant sous Linux incluent cette erreur.

Il ne s'agit pas d'un bug sérieux à proprement parler : son impact fait que read-ahead va lire, sous des charges aléatoires, plus de pages que nécessaire (plus de page que read-ahead devrait lire selon ses spécifications).

3.2 Explications

La fonction noyau *ondemand_readahead()* est appelée à chaque passe de read-ahead. Elle implémente le coeur de l'algorithme de pré-chargement. Son travail est premièrement de déterminer si l'accès actuel est un accès séquentiel. Deuxièmement, en fonction des conclusions tirées sur le type de l'accès courant et de l'historique des accès au fichier concerné, *ondemand_readahead()* va calculer la taille de la fenêtre courante, et la position du prochain flag read-ahead à poser. Plus l'accès est considéré comme séquentiel, plus la taille de la fenêtre sera grande. L'un des tests réalisé par cette fonction pour déterminer la taille de la fenêtre courante est le suivant (extrait des lignes 445 à 449 du fichier `mm/readahead.c` présent dans les sources du noyau version 2.6.38) :

```
/*                                                                    445
 * sequential cache miss                                              446
 */                                                                   447
if (offset - (ra->prev_pos >> PAGE_CACHE_SHIFT) <= 1UL)               448
  goto initial_readahead;                                             449
```

La variable *offset* correspond à l'index de la page dans le fichier qui a déclenché l'appel à la passe de read-ahead (synchrone ou asynchrone). *ra* est un descripteur d'opération read-ahead, il en existe un par couple (*processus*, *fichier accédé*), c'est à dire un par processus ouvrant un fichier. Ici, *ra→prev_pos* correspond à l'adresse de la requête de lecture précédente en octets. En décalant cette valeur à droite de *PAGE_CACHE_SHIFT* (qui vaut 12 sur architecture ARM), on réalise une division par 4096. C'est la

taille d'une page Linux, donc *ra→prev_pos » PAGE_CACHE_SHIFT* correspond à l'index de la précédente page du fichier demandée par le processus. Le résultat de la soustraction des deux termes est comparé avec 1 pour savoir si l'on est dans la continuité de l'accès précédent. Si c'est le cas, on saute au label *initial_readahead* qui initialise une nouvelle fenêtre de lecture puis lance la lecture effective et la pose du flag.

offset est de type *pgoff_t*, qui est en fait un *unsigned int*. Or, *prev_pos* est de type *loff_t* qui est lui-même sous ARM un *long long*, c'est à dire un type signé, contrairement au type de *offset*. C'est à dire qu'en fonction de la charge d'E/S, et en particulier lors de charges aléatoires, le résultat de la soustraction ligne 448 peut être inférieur à 0. Cela rend la clause *if* vraie, et a pour effet final de faire croire à read-ahead que l'on est dans la continuité de la requête précédente, alors que l'on est peut être dans le cas d'une charge totalement aléatoire.

ANNEXE C

Validation des modèles de performances JFFS2 : résultats complémentaires

Ici on présente des résultats complémentaires pour la validation en performances des modèles JFFS2, présentée au chapitre 5 (voir page 197).

1 Lecture - calcul de l'erreur d'estimation

Dans la table C.1, on présente en détails les résultats de validation des performances de JFFS2 en lecture, ainsi que l'erreur d'estimation associée, le tout pour chaque scénario de validation et chaque mode de création du fichier cible.

Id. de scénario	Mode de création du fichier cible	Moyenne des temps d'exécution de *jffs2_readpage*		
		Env. réel (μs)	Simulation (μs)	Erreur (%)
1	Séquentiel	601.64	463.20	**23.01**
	Aléatoire	3143.38	2293.20	**27.05**
2	Séquentiel	625.44	463.20	**25.94**
	Aléatoire	3009.98	2293.20	**23.81**
3	Séquentiel	494.77	448.06	**9.44**
	Aléatoire	2373.99	2471.82	**4.12**
4	Séquentiel	418.63	402.64	**3.82**
	Aléatoire	2504.43	2632.66	**5.12**
5	Séquentiel	424.37	408.64	**3.70**
	Aléatoire	2586.74	2635.98	**1.90**
6	Séquentiel	483.34	469.14	**2.94**
	Aléatoire	2611.33	2668.36	**2.18**
7	Séquentiel	469.18	454.43	**3.14**
	Aléatoire	2535.35	2659.16	**4.88**
8	Séquentiel	476.11	462.34	**2.89**
	Aléatoire	2556.28	2675.57	**4.67**
9	Séquentiel	434.84	419.99	**3.42**
	Aléatoire	2569.34	2633.76	**2.51**

Table C.1 – Calcul de l'erreur de l'estimation des performances de JFFS2 en lecture pour les scénarios de tests présentés au chapitre 5.

On rappelle que les scénarios 1 et 2 n'effectuent qu'une seule requête, on en s'inquiète donc pas de l'erreur d'estimation à ce niveau.

2 Lectures - distributions des temps d'exécutions des appels à *jffs2_readpage()*

Les figures C.1 à C.5 présentent une comparaison des temps d'exécution de *jffs2_readpage()* entre les mesures réelles (à gauche sur les figures) et les valeurs estimées (à droite). Les figures présentent

les résultat pour chaque scénario, et pour chaque mode de création du fichier cible (séquentiel et aléatoire).

Dans ces figures, la largeur plus ou moins importante d'un histogramme correspond à l'étendue plus ou moins grande des données représentées.

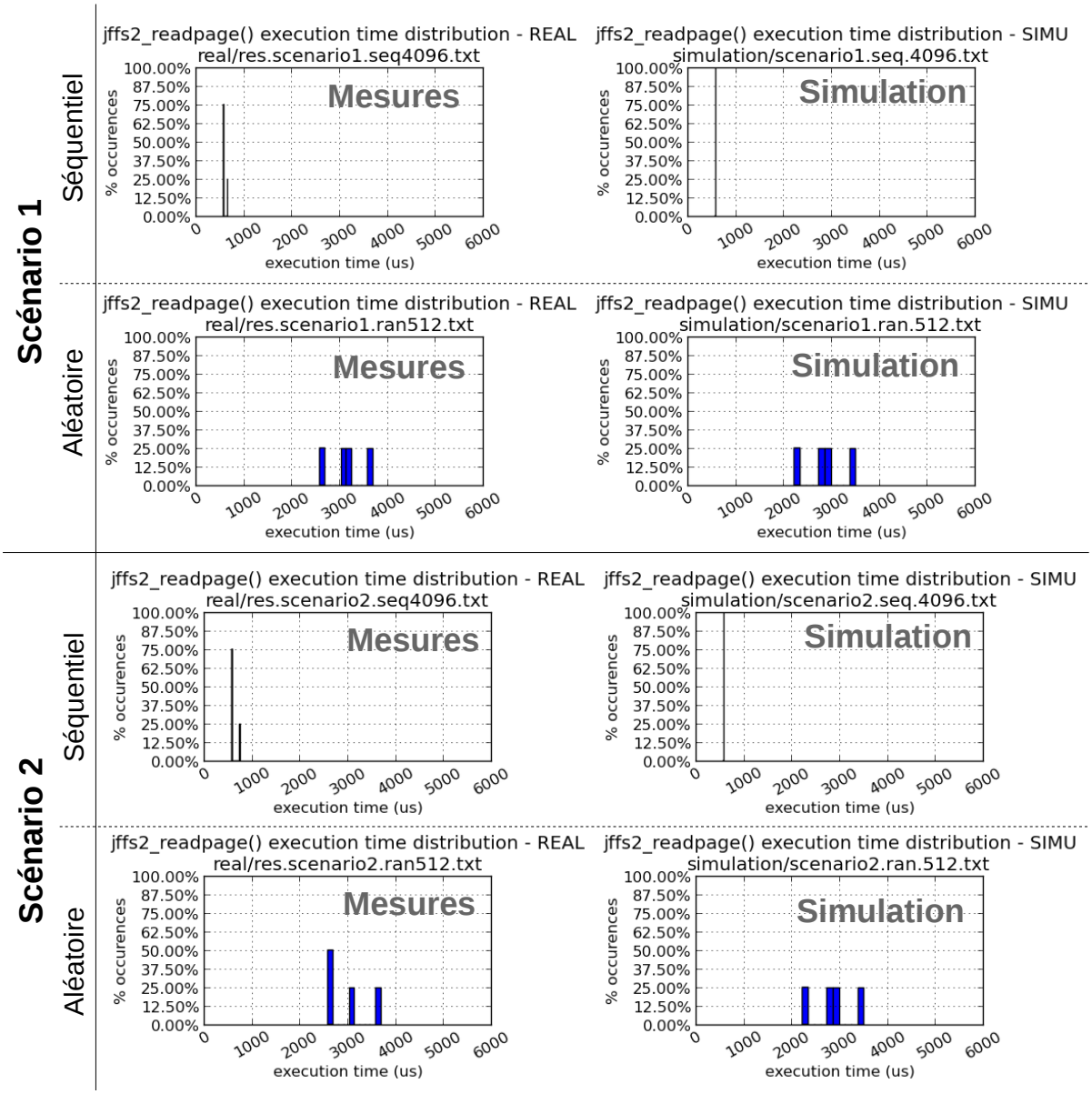


FIGURE C.1 – Distributions des temps d'exécutions des appels à *jffs2_readpage()* - scénarios 1 et 2

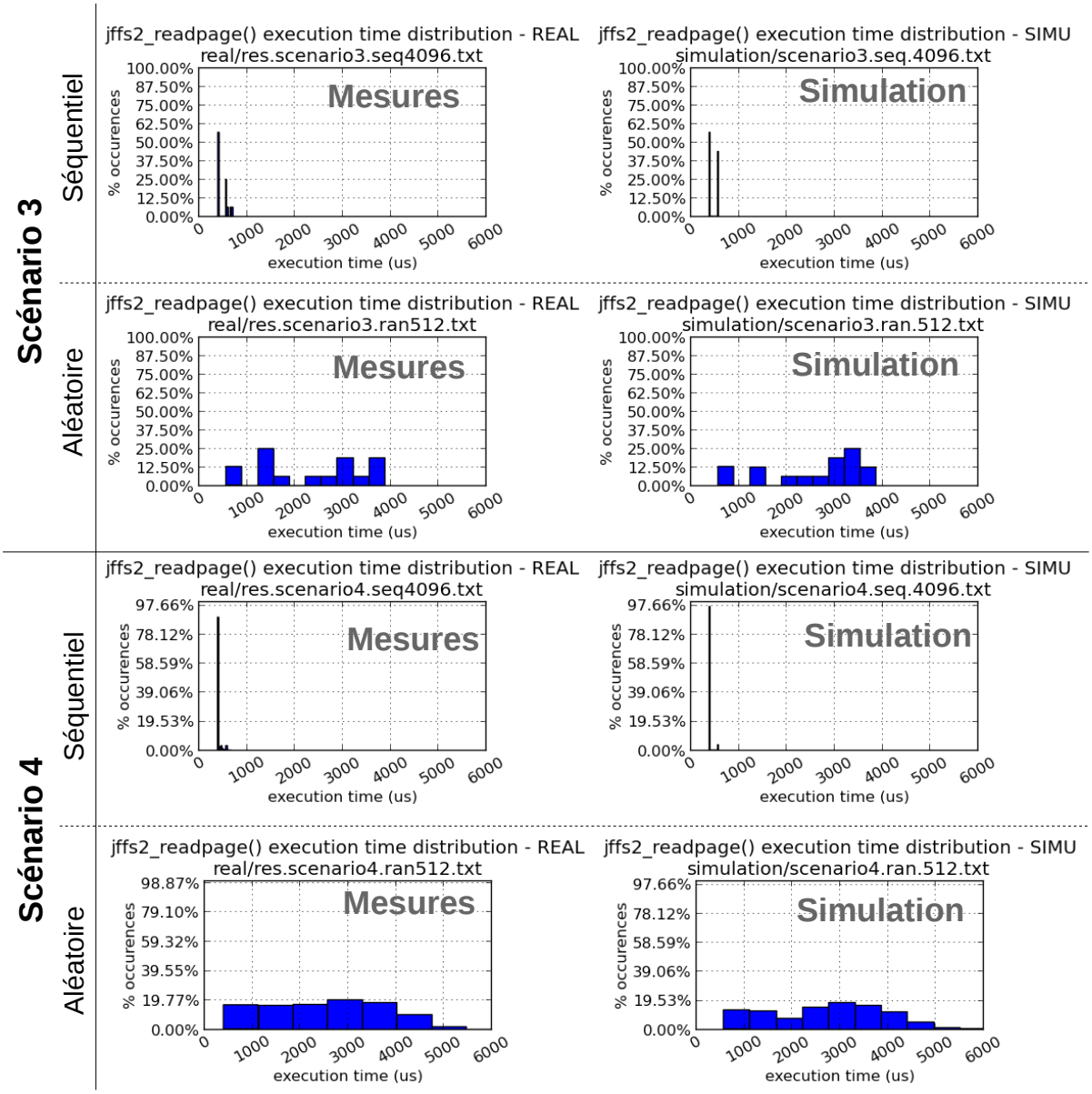


Figure C.2 – Distributions des temps d'exécutions des appels à *jffs2_readpage()* - scénarios 3 et 4

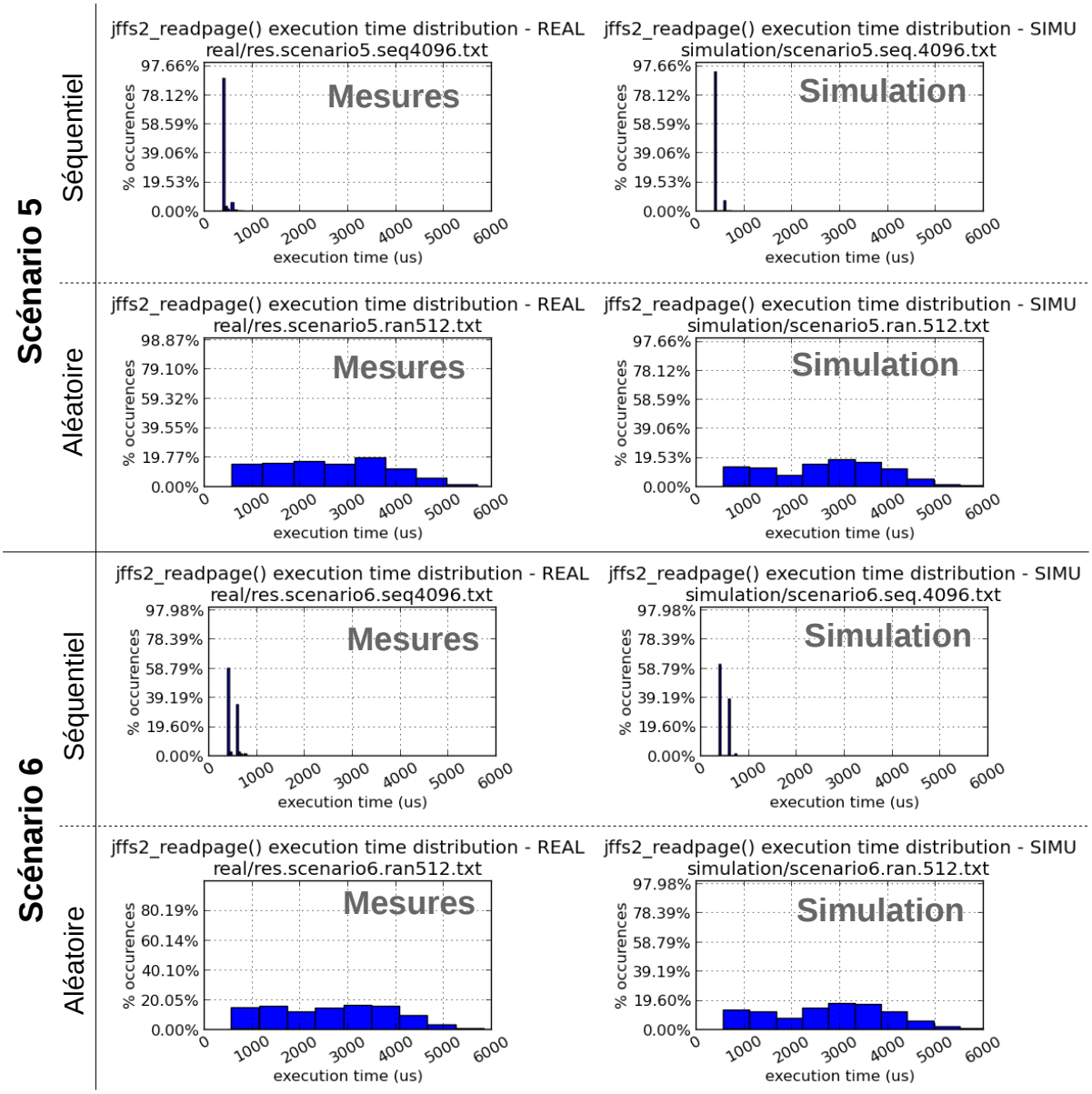


FIGURE C.3 – Distributions des temps d'exécutions des appels à *jffs2_readpage()* - scénarios 5 et 6

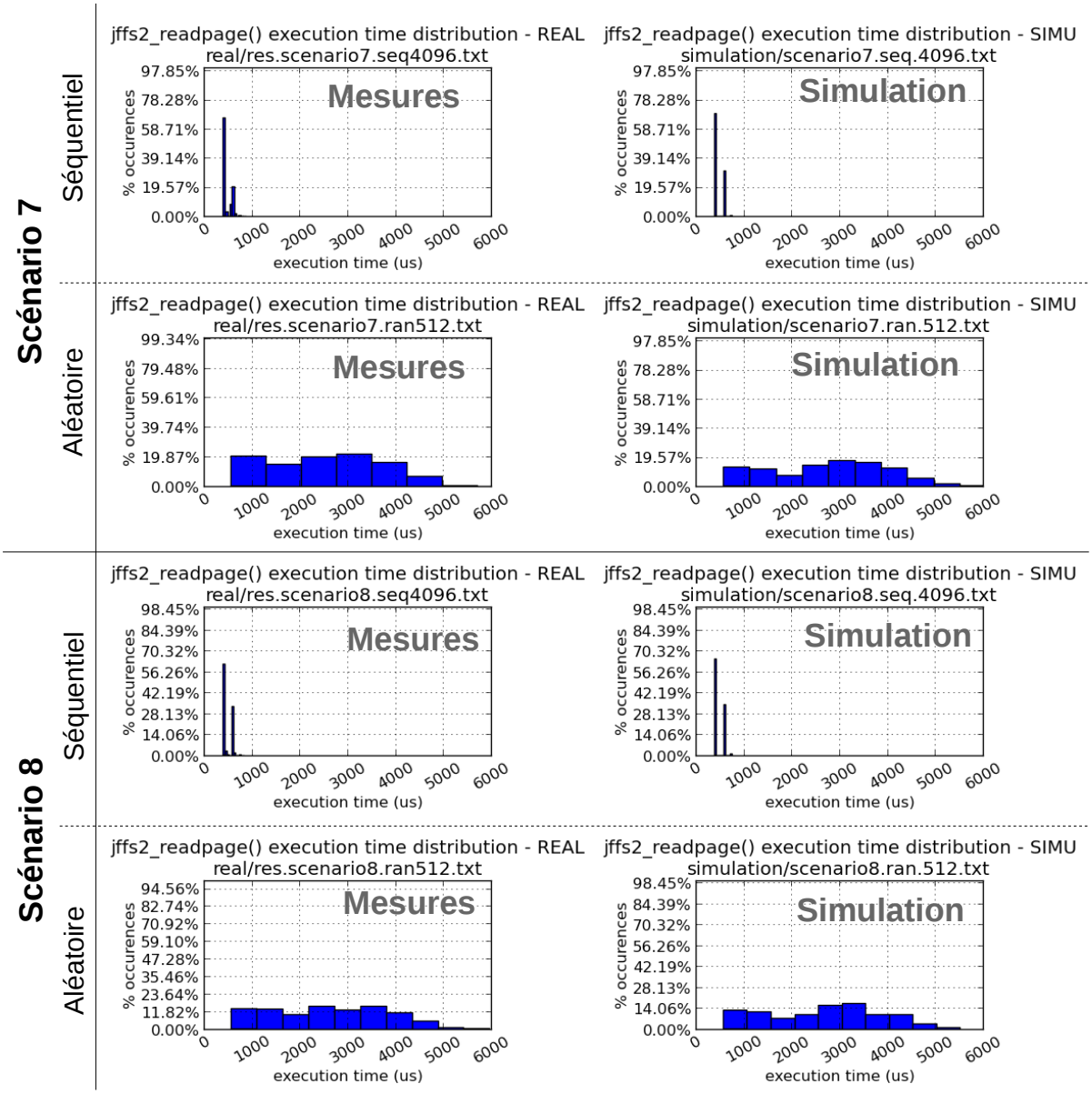


Figure C.4 – Distributions des temps d'exécutions des appels à *jffs2_readpage()* - scénarios 7 et 8

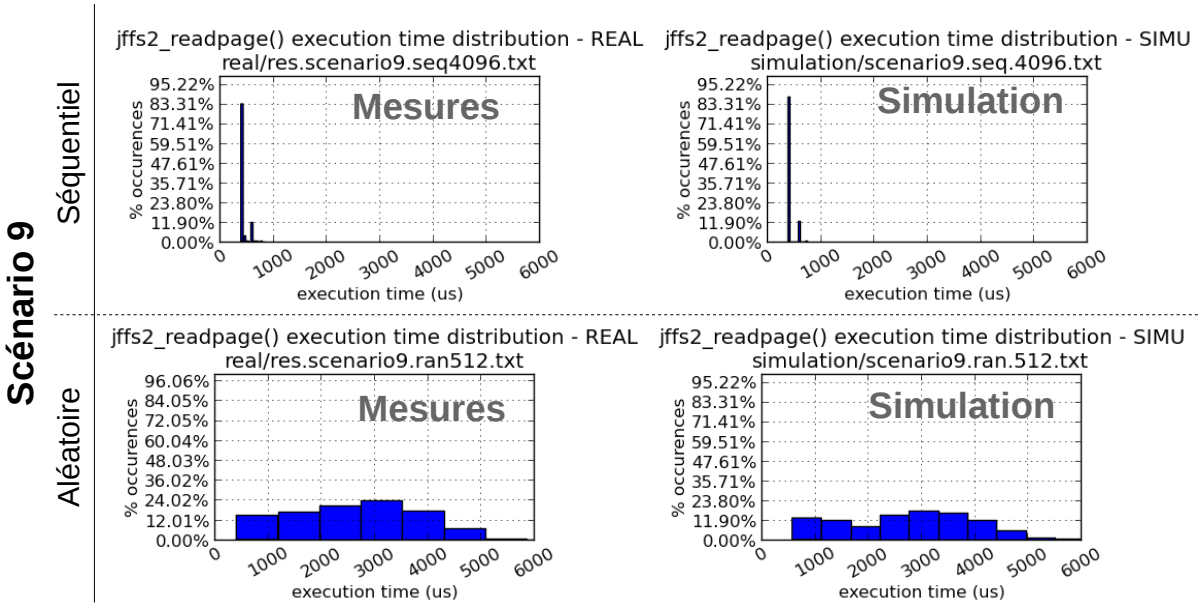


FIGURE C.5 – Distributions des temps d'exécutions des appels à *jffs2_readpage()* - scénario 9

3 Écriture - calcul de l'erreur d'estimation

La table C.2 présente les résultats détaillés de la phase de validation des performances de JFFS2 en écriture, ainsi que l'erreur d'estimation concernant (A) le temps d'exécution moyen de *jffs2_write_begin()*, (B) le temps d'exécution moyen de *jffs2_write_end()* et (C) le nombre total d'écritures de pages flash, dans les trois dernière colonnes.

Id. de scénario	Temps d'exécution moyen (μs)				(C) Nombre d'écritures de pages flash		Erreur (%)		
	(A) jffs2_write_begin()		(B) jffs2_write_end()						
	Env. réel	Simu.	Env. réel	Simu.	Env. réel	Simu.	(A)	(B)	(C)
1	22.5	30	911	887	10423	10432	**33.33**	**2.63**	**0.09**
2	6.4	8.7	240	257	20582	20604	**35.94**	**7.08**	**0.11**
3	5.7	5.7	922	886	10989	11348	**0.00**	**3.90**	**3.27**
4	5	5.7	701	682	16568	16887	**14.00**	**2.71**	**1.93**
5	4.89	5.7	233	240	21927	23118	**16.56**	**3.00**	**5.43**
6	391	374	910	887	11123	11387	**4.35**	**2.53**	**2.37**

TABLE C.2 – Calcul de l'erreur de l'estimation des performances de JFFS2 en écriture pour les scénarios de tests présentés au chapitre 5.

4 Écritures - distributions des temps d'exécutions des appels aux fonctions *jffs2_write_begin()* et *jffs2_write_end()*

Les figures C.6 à C.8 présentent une comparaison des temps d'exécution de *jffs2_write_begin()* et *jffs2_write_end()* entre les mesures réelles (à gauche sur les figures) et les valeurs estimées (à droite). Les figures présentent les résultat pour chaque scénario.

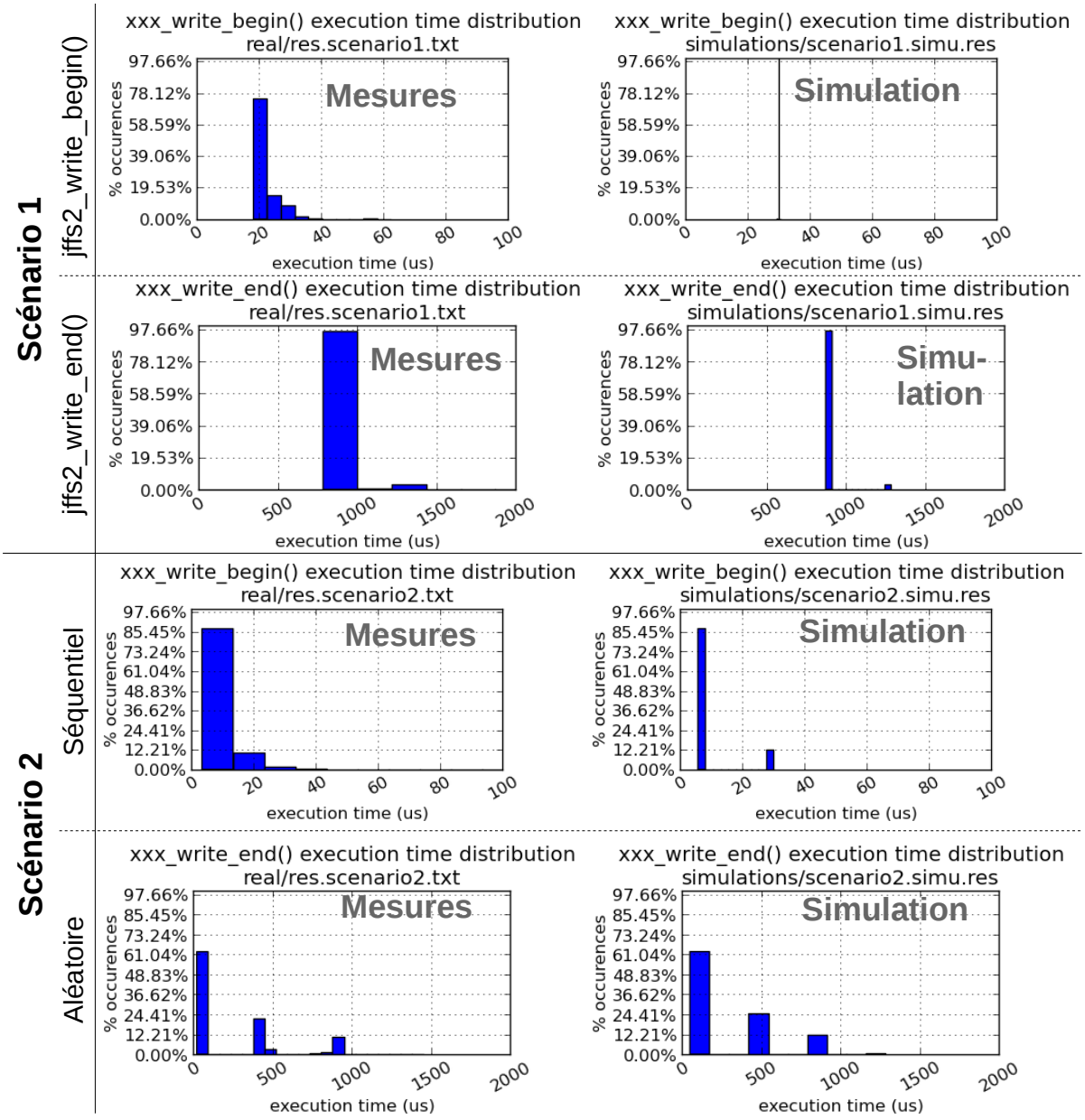


Figure C.6 – Distributions des temps d'exécutions des appels à *jffs2_write_begin()* et *jffs2_write_end()* - scénarios 1 et 2

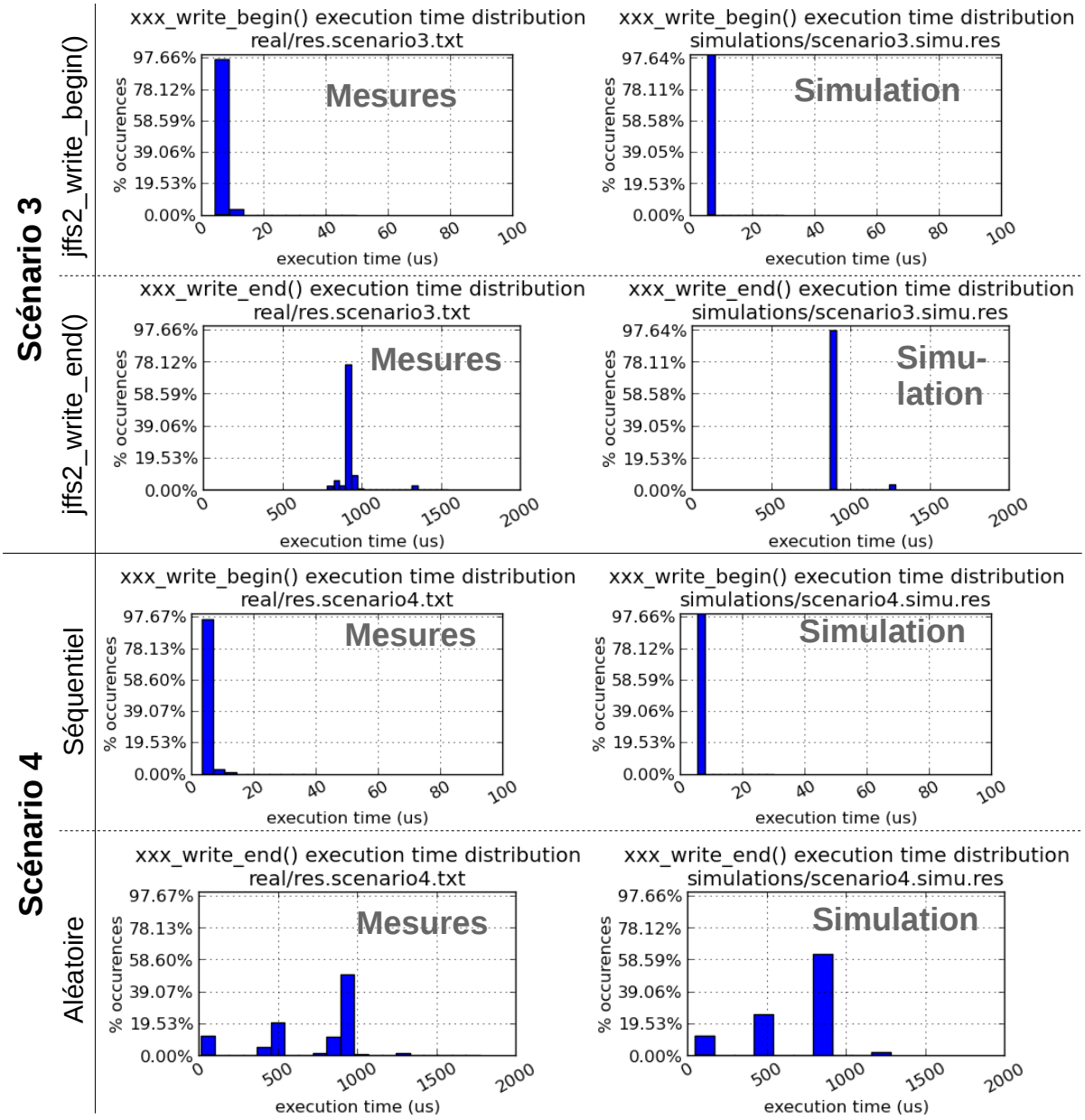


FIGURE C.7 – Distributions des temps d'exécutions des appels à *jffs2_write_begin()* et *jffs2_write_end()* - scénarios 3 et 4

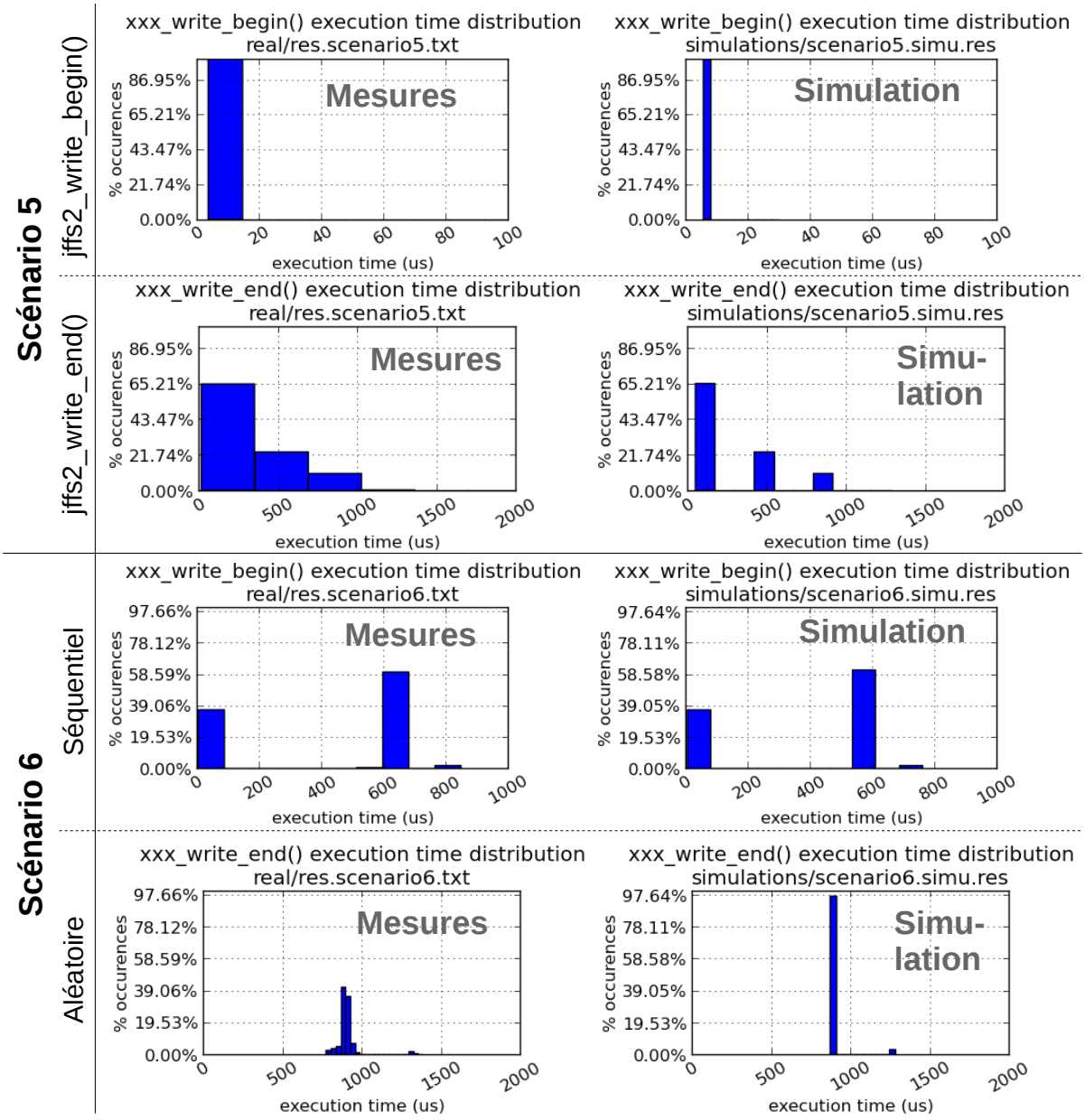


Figure C.8 – Distributions des temps d'exécutions des appels à *jffs2_write_begin()* et *jffs2_write_end()* - scénarios 5 et 6

En conclusion, on peut voir qu'en termes de distribution des temps de réponse des scénarios de test, les modèles concernant JFFS2 sont très proche des mesures en environnement réel.

ANNEXE D

Évaluation de l'impact de Flashmon sur le système tracé

Il est important qu'un outil de trace comme Flashmon ne soit pas intrusif, c'est à dire qu'il n'affecte pas de manière trop importante les résultats qu'il est censé tracer. De plus, dans un contexte embarqué, l'empreinte RAM de Flashmon doit rester faible.

1 Mesure de l'impact de Flashmon sur les performances du système

Cet impact a été mesuré via deux expériences, réalisées sur la carte *Armadeus APF27*. Sur la carte s'exécute la version 2.6.38 de Linux. Les expériences consistent en la mesure du temps d'exécution de programmes effectuant des accès à la mémoire flash, dans un système (1) avec et (2) sans Flashmon. Les temps d'exécution sont alors comparés pour estimer la baisse de performance introduite par l'insertion de Flashmon. Les programmes lancés sont les suivants :

- Expérience A : les programmes MTD *flash_erase*, *nanddump* et *nandwrite* sont des programmes permettant respectivement d'effacer, de lire et d'écrire directement aux adresses physiques de la mémoire flash. Ils sont lancés depuis l'espace utilisateur pour :
  - Effacer une plage correspondante à 100 Mo d'espace flash ;
  - Écrire en flash 5 Mo de données aléatoire contiguës ;
  - Lire depuis la flash les 5 Mo de données précédemment écrites.
- Expérience B : Une configuration du benchmark Postmark est lancée (présentée dans la table D.1) sur une partition flash de 50 Mo, totalement effacée avant l'expérience. Pour cette configuration, Postmark reporte un total de 9.25 Mo lus et 14.45 Mo écrits. L'expérience est répétée sur une partition JFFS2, YAFFS2 et UBIFS.

Les résultats sont présentés dans la table D.2. Le coût de Flashmon sur les performances du système tracé reste inférieur à 6 %.

Paramètre	Valeur
Nombre de fichiers créés à l'initialisation	800
Taille des fichiers créés	entre 512 octets et 10 Ko
Nombre de transactions	3000
Taille des requêtes de lecture et écriture	4 Ko
Ratio transactions lectures / ajout	50%
Ratio transactions création / suppression	50%
Nombre de sous-répertoires	10

Table D.1 – Configuration de Postmark utilisée pour évaluer l'impact de Flashmon sur les performances du système tracé

Expérience A			
Opération	**Temps d'exécution moyen (s)**		**Coût Flashmon (%)**
	Avec Flashmon	**Sans Flashmon**	
Effacer 100 Mo	0.470	0.489	**3.85**
Écrire 5 Mo	1.498	1.570	**4.79**
Lire 5 Mo	2.386	2.515	**5.40**
Expérience B			
FFS	**Temps d'exécution moyen de Postmark (s)**		**Coût Flashmon (%)**
	Avec Flashmon	**Sans Flashmon**	
JFFS2	10.523589	10.628663	**3.08**
YAFFS2	19.793678	20.626676	**4.21**
UBIFS	4.690151	4.754635	**1.37**

TABLE D.2 – Résultats des mesures de l'impact de Flashmon sur les performances du système tracé

2 Estimation de l'empreinte RAM de Flashmon

La quantité de RAM utilisée par Flashmon est égale à la somme des parties dynamique (mémoire allouée via l'équivalent niveau noyau de *malloc()*) et statique (le reste) de la mémoire utilisée par le module. La taille de la partie statique peut être obtenue en observant l'entrée correspondant à Flashmon dans le fichier */proc/modules*, centralisant des informations sur les modules en cours d'exécution. Sur la carte Armadeus, la version actuelle de Flashmon utilise 8861 octets de mémoire statique. Pour ce qui est de la partie dynamique, elle est composée elle-même de deux sous-parties :

— Les compteurs stockant les informations relatives à la vue spatiale : il s'agit de trois compteurs (lectures / écritures /effacements) de type entier non signé sur 32 bits (4 octets). Il y a un triplet de compteur pour chaque bloc de la mémoire flash tracée ;

— Le log temporel : sa taille en nombre d'évènements est spécifiée au chargement de Flashmon et l'intégralité du log est allouée dès le chargement. L'empreinte RAM relative au log est donc égale à la taille d'une entrée (un évènement tracé) multipliée par la taille du log. Sur la carte Armadeus une entrée à une taille de 36 octets.

On peut alors modéliser la consommation RAM de Flashmon via la formule suivante :

$$C_{Flashmon} = C_{statique} + C_{dynamique} \tag{D.1}$$

$$= C_{statique} + 4 * 3 * N_{blocs} + T_{log} * N_{log} \text{ octets} \tag{D.2}$$

Dans cette équation $C_{Flashmon}$ est la quantité de RAM utilisé par Flashmon, dont la $C_{statique}$ est la partie statique et $C_{dynamique}$ est la partie dynamique. N_{blocs} est le nombre de blocs flash contenus dans la partition tracée. T_{log} est la taille du log temporel en nombre d'entrées, et N_{log} est la taille d'une entrée. Prenons l'exemple d'une partition tracée de 50 Mo sur la carte Armadeus, dont la puce flash contient des blocs de 128 Ko ce qui fait 400 blocs pour cette partition :

$$C_{Flashmon} = 8861 + 4 * 3 * 400 + 36 * N_{log} \tag{D.3}$$

La taille du log pouvant potentiellement être très importante, c'est elle qui va déterminer la majeure partie de l'empreinte mémoire de Flashmon. Sur la carte Armadeus, dans les 36 octets composant la taille d'une entrée du log, 16 sont utilisés pour stocker le nom de la tâche courante. Flashmon possède une option pour désactiver la trace du nom du processus en cours d'exécution, ce qui permet d'économiser un espace RAM non négligeable en réduisant la taille d'une entrée à 20 octets. Il faut

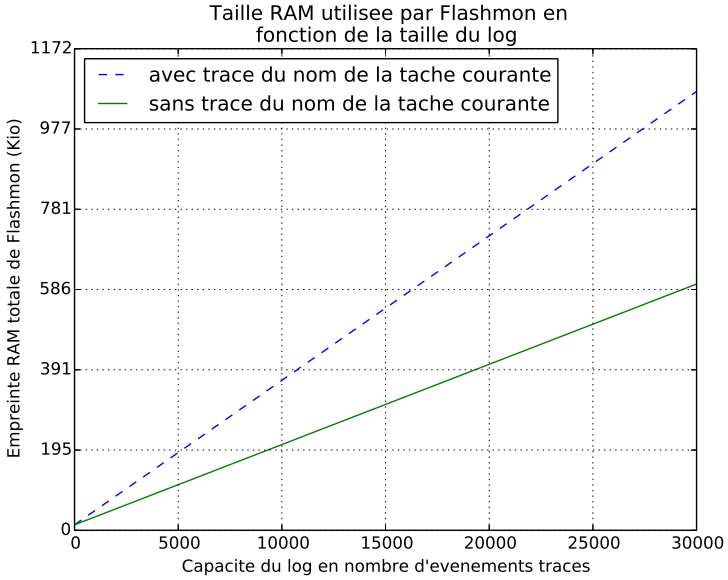


Figure D.1 – Évolution de la taille de l'empreinte RAM de Flashmon en fonction de la taille du log, dans le cas de la trace sur la carte Armadeus (blocs flash de l28 Ko) des accès à une partition de 50 Mo.

noter qu'il serait bien entendu possible d'appliquer plusieurs optimisations au niveau du stockage du nom de la tâche courante, autres que la suppression complète de cette fonctionnalité. Au cours du développement de Flashmon cela n'a pas été considéré comme une priorité.

On peut tracer l'évolution de la taille de l'empreinte RAM de Flashmon en fonction de la taille maximale du log dans les conditions telles que présentées ci-dessus. Cela est illustré sur la figure D.1. A partir d'une taille maximale de log égale à 30 000 évènements, la consommation RAM de Flashmon est d'environ 1 Mo si l'on trace le nom de la tâche courante, mais cette consommation est réduite à moins de 600 Ko lorsque ce nom n'est pas tracé.

Références

Anderson D. et Chase J., « Fstress : A flexible network file service benchmark », Rapport technique, Technical Report CS-2002-01, Duke University, Department of Computer Science, adresse : http://www.cs.duke.edu/ari/fstress/fstress.pdf, 2002.

Aranya A., Wright C.P. et Zadok E., « Tracefs : A file system to trace them all. », Dans *FAST*, p. 129–145, 2004, adresse : https://www.usenix.org/legacy/publications/library/proceedings/fast04/tech/full_papers/aranya/aranya_html/.

Axboe J., Fio documentation, adresse : https://github.com/axboe/fio/blob/master/README, 2014.

Axis Communications, JFFS home page, adresse : http://developer.axis.com/old/software/jffs/, 1999.

Ban A., United states patent : 5404485 - flash file system, adresse : http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5404485, avr. 1995.

Barrett P.L., Quinn S.D. et Lipe R.A., United states patent : 5392427 - system for updating data stored on a flash-erasable, programmable, read-only memory (FEPROM) based upon predetermined bit value of indicating pointers, adresse : http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5392427, fév. 1995.

Bellard F., « QEMU, a fast and portable dynamic translator. », Dans *USENIX Annual Technical Conference, FREENIX Track*, p. 41–46, 2005, adresse : https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard_html/.

Benmoussa Y., Senn E. et Boukhobza J., « Open-PEOPLE, a collaborative platform for remote and accurate measurement and evaluation of embedded systems power consumption. », Dans *22nd IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2014.

Bez R., Camerlenghi E., Modelli A. et Visconti A., « Introduction to flash memory », Dans *Proceedings of the IEEE*, p. 489–502, 2003, adresse : http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.313.5309.

Bityutskiy A., *JFFS3 design issues*, adresse : http://idke.ruc.edu.cn/people/dazhou/Papers/JFFS3design.pdf, 2005.

Bjørling M., Bonnet P., Bouganim L., Jónsson B. et coll., « uFLIP : understanding the energy consumption of flash devices », *IEEE Data Engineering Bulletin*, vol. 33, n° 4, p. 48–54, 2010a, adresse : http://hal.inria.fr/hal-00552170/.

Bjørling M., Le Folgoc L., Mseddi A., Bonnet P., Bouganim L. et Jónsson B., « Performing sound flash device measurements : some lessons from uFLIP », Dans *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, p. 1219–1222, ACM, 2010b, adresse : http://dl.acm.org/citation.cfm?id=1807324.

Boschetti L., Software profile : Journaling flash file system, version 2 (jffs2), 2011.

Bouganim L. et Bonnet P., « Flash device support for database management », Dans *Proceedings of the 5th Biennal Conference on Innovative Data System*, p. 1–8, Asilomar, California, États-Unis, 2011, adresse : http://hal.archives-ouvertes.fr/hal-00666180.

Bouganim L., Jónsson B. et Bonnet P., « uFLIP : understanding flash IO patterns », *arXiv preprint arXiv :0909.1780*, 2009, adresse : http://arxiv.org/abs/0909.1780.

BOUKHOBZA J., OLIVIER P. et RUBINI S., « A cache management strategy to replace wear leveling techniques for embedded flash memory », Dans *Performance Evaluation of Computer & Telecommunication Systems (SPECTS), 2011 International Symposium on*, p. 1–8, IEEE, 2011, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5984840.

BOVET D.P. et CESATI M., *Understanding the Linux kernel*, " O'Reilly Media, Inc.", adresse : http://books.google.fr/books?hl=en&lr=&id=h0lltXyJ8aIC&oi=fnd&pg=PT11&dq=understanding+the+linux+kernel&ots=g0-pITb9GR&sig=UwyNONC-jNaJnmVLzIIHcgsHQ8c, 2005.

BRAY T., The bonnie file system benchmark, adresse : http://www.tbray.org/ongoing/misc/Software#p-1, 1996.

BROWN N., « An F2FS teardown », *Linux Weekly News*, 2012, adresse : http://lwn.net/Articles/518988/.

BRUNELLE A.D., Blktrace user guide, 2007.

BRYANT R., FORESTER R. et HAWKES J., « Filesystem performance and scalability in linux 2.4. 17 », Dans *Proceedings of the Freenix Track : 2002 USENIX Annual Technical Conference*, p. 259–274, USENIX Association, 2002, adresse : http://dl.acm.org/citation.cfm?id=715936.

BUCY J.S., SCHINDLER J., SCHLOSSER S.W. et GANGER G.R., « The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101) », Rapport technique, Parallel Data Laboratory, adresse : http://repository.cmu.edu/cgi/viewcontent.cgi?article=1025&context=pdl, 2008.

CARROLL A. et HEISER G., « An analysis of power consumption in a smartphone. », Dans *USENIX annual technical conference*, p. 271–285, 2010, adresse : https://www.usenix.org/event/usenix10/tech/full_papers/Carroll.pdf.

CAULFIELD A.M., GRUPP L.M. et SWANSON S., « Gordon : using flash memory to build fast, power-efficient clusters for data-intensive applications », *ACM Sigplan Notices*, vol. 44, nº 3, p. 217–228, 2009, adresse : http://dl.acm.org/citation.cfm?id=1508270.

CHANG L. et KUO T., « An adaptive striping architecture for flash memory storage systems of embedded systems », Dans *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, p. 187–196, IEEE, 2002, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1137393.

CHEN F., JIANG S. et ZHANG X., « SmartSaver : turning flash drive into a disk energy saver for mobile computers », Dans *Low Power Electronics and Design, 2006. ISLPED'06. Proceedings of the 2006 International Symposium on*, p. 412–417, IEEE, 2006, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4271878.

CHEN J.J., MIELKE N.R. et CALVIN HU C., Flash memory reliability, Dans *Nonvolatile Memory Technologies with Emphasis on Flash*, BREWER J. et GILL M. (coordinateurs), p. 445–590, John Wiley & Sons, Inc., 2007, ISBN 9780470181355, adresse : http://onlinelibrary.wiley.com/doi/10.1002/9780470181355.ch11/summary.

CHIANG M.L. et CHANG R.C., « Cleaning policies in mobile computers using flash memory », *Journal of Systems and Software*, vol. 48, nº 3, p. 213–231, nov. 1999, ISSN 0164-1212, doi : 10.1016/S0164-1212(99)00059-X, adresse : http://www.sciencedirect.com/science/article/pii/S016412129900059X.

CHIAO M. et CHANG D., « ROSE : a novel flash translation layer for NAND flash memory based on hybrid address translation », *Computers, IEEE Transactions on*, vol. 60, nº 6, p. 753–766, 2011, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5740855.

CHOUDHURI S., *Macromodeling and characterization of filesystem energy consumption for diskless embedded systems*, Thèse de doctorat, Texas A&M University, adresse : http://repository.tamu.edu/handle/1969.1/295, 2003.

CHUNG T., PARK D., PARK S., LEE D., LEE S. et SONG H., « A survey of flash translation layer », *Journal of Systems Architecture*, vol. 55, nº 5–6, p. 332–343, mai 2009, ISSN 1383-7621, doi : 10.1016/j.sysarc.2009.03.005, adresse : http://www.sciencedirect.com/science/article/pii/S1383762109000356.

Coker R., « The bonnie++ benchmark », *URL http ://www. coker. com. au/bonnie+*, 2009.

Davis J.D. et Rivoire S., « Building energy-efficient systems for sequential workloads », *Energy*, vol. 1, p. 0BINTRODUCTION, 2010, adresse : http://research.microsoft.com/pubs/121686/joulesort_trv1.0.pdf.

Dayan N., Svendsen M.K.e., Björling M., Bonnet P. et Bouganim L., « EagleTree : exploring the design space of SSD-based algorithms », *Proceedings of the VLDB Endowment*, vol. 6, n° 12, p. 1290–1293, 2013, adresse : http://dl.acm.org/citation.cfm?id=2536298.

Doh I.H., Lee H.J., Moon Y.J., Kim E., Choi J., Lee D. et Noh S.H., « Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems », Dans *Proceedings of the 2009 ACM symposium on Applied Computing*, p. 1658–1663, ACM, 2009, adresse : http://dl.acm.org/citation.cfm?id=1529654.

Dong X., Xu C., Xie Y. et Jouppi N.P., « Nvsim : A circuit-level performance, energy, and area model for emerging nonvolatile memory », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, n° 7, p. 994–1007, 2012, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6218223.

Douglis F., Caceres R., Kaashoek M.F., Krishnan P., Li K., Marsh B. et Tauber J., Storage alternatives for mobile computers, Dans *Mobile Computing*, p. 473–505, Springer, 1996, adresse : http://link.springer.com/chapter/10.1007/978-0-585-29603-6_18.

Douhib S., *Operating systems power and energy consumption characterization for critical embedded systems described in AADL*, Thèse de doctorat, Université de Bretagne Sud, 2009.

Du Y., Cai M. et Dong J., « Adaptive energy-aware design of a multi-bank flash-memory storage system », Dans *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings*, p. 311–316, août 2005, doi : 10.1109/RTCSA.2005.18.

Eigler F.C., Prasad V., Cohen W., Nguyen H., Hunt M., Keniston J. et Chen B., *Architecture of systemtap : a Linux trace/probe tool*, 2005.

El Maghraoui K., Kandiraju G., Jann J. et Pattnaik P., « Modeling and simulating flash based solid-state disks for operating systems », Dans *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, p. 15–26, ACM, 2010, adresse : http://dl.acm.org/citation.cfm?id=1712611.

Engel J. et Mertens R., « LogFS-finally a scalable flash file system », Dans *12th International Linux System Technology Conference*, 2005, adresse : http://www.cs.xu.edu/csci390/08s/logfs1.pdf.

Fazio A. et Bauer M., Multilevel cell digital memories, Dans *Nonvolatile Memory Technologies with Emphasis on Flash*, Brewer J. et Gill M. (coordinateurs), p. 591–616, John Wiley & Sons, Inc., 2007, ISBN 9780470181355, adresse : http://onlinelibrary.wiley.com/doi/10.1002/9780470181355.ch12/summary.

Forni G., Ong C., Rice C., McKee K. et Bauer R.J., Flash memory applications, Dans *Nonvolatile Memory Technologies with Emphasis on Flash*, Brewer J. et Gill M. (coordinateurs), p. 19–62, John Wiley & Sons, Inc., 2007, ISBN 9780470181355, adresse : http://onlinelibrary.wiley.com/doi/10.1002/9780470181355.ch2/summary.

Gleditsch A.G. et Gjermshus P.K., *Linux Cross Reference*, 2006.

Gleixner T., Haverkamp F. et Bityutskiy A., « UBI-Unsorted block images », Rapport technique, adresse : http://linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf, 2006.

Gough B., *GNU scientific library reference manual*, Network Theory Ltd., 2009.

Goyal N., *Macro-modeling and energy efficiency studies of file management in embedded systems with flash memory*, Thèse de doctorat, Texas A&M University, adresse : http://repository.tamu.edu/handle/1969.1/3766, 2005.

GOYAL N. et MAHAPATRA R., « Energy characterization of cramfs for embedded systems », Dans *IWSSPS. Proc. International Workshop on Software Support for Portable Storage (March 2005)*, 2005, adresse : http://research.cs.tamu.edu/codesign/papers/iwssps_final.pdf.

GRUPP L.M., CAULFIELD A.M., COBURN J., SWANSON S., YAAKOBI E., SIEGEL P.H. et WOLF J.K., « Characterizing flash memory : anomalies, observations, and applications », Dans *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, p. 24–33, IEEE, 2009, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5375312.

GUPTA A., KIM Y. et URGAONKAR B., « DFTL : a flash translation layer employing demand-based selective caching of page-level address mappings », Dans *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. ASPLOS XIV, p. 229–240, New York, NY, USA, ACM, 2009, ISBN 978-1-60558-406-5, doi : 10.1145/1508244.1508271, adresse : http://doi.acm.org/10.1145/1508244.1508271.

HAMMING R.W., *Coding and information theory*, Prentice-Hall, Inc., adresse : http://dl.acm.org/citation.cfm?id=5484, 1986.

HAVASI F., An improved b+ tree for flash file systems, Dans *SOFSEM 2011 : Theory and Practice of Computer Science*, p. 297–307, Springer, 2011, adresse : http://link.springer.com/chapter/10.1007/978-3-642-18381-2_25.

HEGER D., JACOBS J., RAO S. et SANTOS J., The flexible file system benchmark webpage, adresse : http://sourceforge.net/projects/ffsb/, 2008.

HIDAKA H., « Evolution of embedded flash memory technology for MCU », p. 1–4, IEEE, mai 2011, ISBN 978-1-4244-9019-6, doi : 10.1109/ICICDT.2011.5783209, adresse : http://scdproxy.univ-brest.fr:2182/xpl/articleDetails.jsp?tp=&arnumber=5783209&queryText%3DEvolution+of+embedded+flash+memory+technology+for+MCU.

HOMMA T., Evaluation of flash file systems for large NAND flash memory, adresse : http://elinux.org/images/7/7e/ELC2009-FlashFS-Toshiba.pdf, 2009.

HOWARD J.H., KAZAR M.L., MENEES S.G., NICHOLS D.A., SATYANARAYANAN M., SIDEBOTHAM R.N. et WEST M.J., « Scale and performance in a distributed file system », *ACM Transactions on Computer Systems (TOCS)*, vol. 6, nº 1, p. 51–81, 1988, adresse : http://dl.acm.org/citation.cfm?id=35059.

HP LABS, Cello 99 traces - SNIA, adresse : http://iotta.snia.org/traces/21, 1999.

HU X., ELEFTHERIOU E., HAAS R., ILIADIS I. et PLETKA R., « Write amplification analysis in flash-based solid state drives », Dans *Proceedings of SYSTOR 2009 : The Israeli Experimental Systems Conference*, p. 10, ACM, 2009, adresse : http://dl.acm.org/citation.cfm?id=1534544.

HU Y., JIANG H., FENG D., TIAN L., LUO H. et ZHANG S., « Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity », Dans *Proceedings of the international conference on Supercomputing*, p. 96–107, ACM, 2011, adresse : http://dl.acm.org/citation.cfm?id=1995912.

HUI S., RUI Z., JIN C., LEI L., FEI W. et SHENG X.C., « Analysis of the file system and block IO scheduler for SSD in performance and energy consumption », p. 48–55, IEEE, déc. 2011, ISBN 978-1-4673-0206-7, 978-0-7695-4624-7, doi : 10.1109/APSCC.2011.29, adresse : http://scdproxy.univ-brest.fr:2182/search/searchresult.jsp?newsearch=true&queryText=Analysis+of+the+File+System+and+Block+IO+Scheduler+for+SSD+in+Performance+and+Energy+Consumption.

HUNTER A., « A brief introduction to the design of UBIFS », Rapport technique, March, 2008.

HWANG J., F2FS : a new file system designed for flash storage in mobile, adresse : http://elinux.org/images/8/81/A_New_File_System_Designed_for_Flash_Storage_in_Mobile.pdf, 2012.

IC INSIGHTS, Research bulletin - total flash memory market will surpass DRAM for first time in 2012, adresse : http://www.icinsights.com/news/bulletins/Total-Flash-Memory-Market-Will-Surpass-DRAM-For-First-Time-In-2012/, 2012.

Intel Corp., Intel X25-M solid state drive datasheet, adresse : http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf, 2009.

Intel Corp. and Micron Inc., Intel and micron develop the world's fastest NAND flash memory with 5X faster performance, http ://www.intc.com/releasedetail.cfm ?ReleaseID=311576, adresse : http://www.intc.com/releasedetail.cfm?ReleaseID=311576, 2008.

Jain R., *The art of computer systems performance analysis*, John Wiley & Sons, 2008.

Ji J., Wang C. et Zhou X., « System-level early power estimation for memory subsystem in embedded systems », *Proc. of SEC*, 2008, adresse : http://home.ustc.edu.cn/~saintwc/Conf/SEC-Power.pdf.

Johnson N., « Jetstress field guide », Rapport technique, Microsoft, adresse : http://gallery.technet.microsoft.com/office/Jetstress-2013-Field-Guide-2438bc12, 2013.

Joukov N., Wong T. et Zadok E., « Accurate and efficient replaying of file system traces. », Dans *FAST*, vol. 5, p. 25–25, 2005, adresse : http://static.usenix.org/events/fast05/tech/full_papers/joukov/joukov_html/.

Jung D., Kim J.s., Park S.y., Kang J. et Lee J., « Fass : A flash-aware swap system », Dans *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS)*, 2005, adresse : http://csl.skku.edu/papers/iwssps05.pdf.

Jung D., Kim J., Kim J. et Lee J., « ScaleFFS : a scalable log-structured flash file system for mobile multimedia systems », *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, vol. 5, n° 1, p. 9, 2008a, adresse : http://dl.acm.org/citation.cfm?id=1404889.

Jung M., Wilson E.H., Donofrio D., Shalf J. et Kandemir M.T., « NANDFlashSim : intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level », Dans *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, p. 1–12, IEEE, 2012, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6232389.

Jung S., Lee K., Kim K., Shin S., Lee S., Om J., Bae G. et Lee J., « Modeling of shift in nand Flash-Memory cell device considering crosstalk and Short-Channel effects », *Electron Devices, IEEE Transactions on*, vol. 55, n° 4, p. 1020–1026, 2008b, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4475401.

Kang Y. et Miller E.L., « Adding aggressive error correction to a high-performance compressing flash file system », Dans *Proceedings of the Seventh ACM International Conference on Embedded Software*, vol. EMSOFT '09, p. 305–314, New York, NY, USA, ACM, 2009, ISBN 978-1-60558-627-4, doi : 10.1145/1629335.1629376, adresse : http://doi.acm.org/10.1145/1629335.1629376.

Katcher J., « Postmark : A new file system benchmark », Rapport technique, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, adresse : https://koala.cs.pub.ro/redmine/attachments/download/605/Katcher97-postmark-netapp-tr3022.pdf, 1997.

Kawaguchi A., Nishioka S. et Motoda H., « A flash-memory based file system », Dans *Proceedings of the USENIX 1995 Technical Conference Proceedings*, vol. TCON'95, p. 13–13, Berkeley, CA, USA, USENIX Association, 1995, adresse : http://dl.acm.org/citation.cfm?id=1267411.1267424.

Keniston J., Panchamukhi P.S. et Hiramatsu M., Linux kernel probes documentation, adresse : https://www.kernel.org/doc/Documentation/kprobes.txt, 2014.

Kgil T. et Mudge T., « FlashCache : a NAND flash memory file cache for low power web servers », Dans *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, p. 103–112, ACM, 2006, adresse : http://dl.acm.org/citation.cfm?id=1176774.

Kgil T., Roberts D. et Mudge T., « Improving NAND flash based disk caches », Dans *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, p. 327–338, IEEE, 2008, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4556737.

KIM H. et AHN S., « BPLRU : a buffer management scheme for improving random writes in flash storage. », Dans *FAST*, vol. 8, p. 1–14, 2008, adresse : http://static.usenix.org/legacy/events/fast08/tech/full_papers/kim/kim.pdf.

KIM H., WON Y. et KANG S., « Embedded NAND flash file system for mobile multimedia devices », *Consumer Electronics, IEEE Transactions on*, vol. 55, n° 2, p. 545–552, 2009a, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5174420.

KIM H., AGRAWAL N. et UNGUREANU C., « Revisiting storage for smartphones », *ACM Transactions on Storage (TOS)*, vol. 8, n° 4, p. 14, 2012a, adresse : http://dl.acm.org/citation.cfm?id=2385607.

KIM J., SHIM H., PARK S., MAENG S. et KIM J., « FlashLight : a lightweight flash file system for embedded systems », *ACM Trans. Embed. Comput. Syst.*, vol. 11S, n° 1, p. 18 :1–18 :23, juin 2012b, ISSN 1539-9087, doi : 10.1145/2180887.2180895, adresse : http://doi.acm.org/10.1145/2180887.2180895.

KIM J. et KIM J., Androbench : Benchmarking the storage performance of android-based mobile devices, Dans *Frontiers in Computer Education*, p. 667–674, Springer, 2012, adresse : http://link.springer.com/chapter/10.1007/978-3-642-27552-4_89.

KIM Y., TAURAS B., GUPTA A. et URGAONKAR B., « Flashsim : A simulator for nand flash-based solid-state drives », Dans *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, p. 125–131, IEEE, 2009b, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5283998.

KOPYTOV A., « SysBench manual », *MySQL AB*, 2012, adresse : http://blog.chinaunix.net/attachment/attach/26/89/68/62268968627f7a99c8f760d9c37db4d4c308e2d4f9.pdf.

KUMAR RETHINAGIRI S., *Une approche système pour l'estimation de la consommation de puissance des plateformes MPSoC*, Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambresis, 2014.

KUOPPALA M., Tiobench-Threaded I/O bench for linux, adresse : http://sourceforge.net/projects/tiobench/, 2002.

KWON O. et KOH K., « Swap-Aware garbage collection for NAND flash memory based embedded systems », Dans *7th IEEE International Conference on Computer and Information Technology, 2007. CIT 2007*, p. 787–792, oct. 2007, doi : 10.1109/CIT.2007.76.

LAHIRI K., RAGHUNATHAN A., DEY S. et PANIGRAHI D., « Battery-driven system design : a new frontier in low power design », Dans *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, p. 261–267, 2002, doi : 10.1109/ASPDAC.2002.994932.

LEE C. et LIM S., « Caching and deferred write of metadata for yaffs2 flash file system », Dans *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, p. 41–46, IEEE, 2011, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6104497.

LEE K. et WON Y., « Smart layers and dumb result : IO characterization of an android-based smartphone », Dans *Proceedings of the tenth ACM international conference on Embedded software*, p. 23–32, ACM, 2012, adresse : http://dl.acm.org/citation.cfm?id=2380367.

LEE S. et KIM J., « Improving performance and capacity of flash storage devices by exploiting heterogeneity of MLC flash memory », *IEEE Transactions on Computers*, vol. Early Access Online, 2013, ISSN 0018-9340, doi : 10.1109/TC.2013.120.

LEE S.W., PARK D.J., CHUNG T.S., LEE D.H., PARK S. et SONG H.J., « A log buffer-based flash translation layer using fully-associative sector translation », *ACM Trans. Embed. Comput. Syst.*, vol. 6, n° 3, juil. 2007, ISSN 1539-9087, doi : 10.1145/1275986.1275990, adresse : http://doi.acm.org/10.1145/1275986.1275990.

LEE S., MOON B. et PARK C., « Advances in flash memory SSD technology for enterprise database applications », Dans *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, p. 863–870, ACM, 2009a, adresse : http://dl.acm.org/citation.cfm?id=1559937.

Lee S. et Kim J., « Using dynamic voltage scaling for energy-efficient flash-based storage devices », Dans *SoC Design Conference (ISOCC), 2010 International*, p. 63–66, IEEE, 2010, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5682971.

Lee S., Ha K., Zhang K., Kim J. et Kim J., « FlexFS : a flexible flash file system for MLC NAND flash memory », Dans *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, vol. USENIX'09, p. 9–9, Berkeley, CA, USA, USENIX Association, 2009b, adresse : http://dl.acm.org/citation.cfm?id=1855807.1855816.

Lensing P.H., Cortes T. et Brinkmann A., « Direct lookup and hash-based metadata placement for local file systems », Dans *Proceedings of the 6th International Systems and Storage Conference*, p. 5, ACM, 2013, adresse : http://dl.acm.org/citation.cfm?id=2485741.

Levon J., « OProfile manual », *Victoria University of Manchester*, 2004, adresse : https://pixhawk.ethz.ch/_media/software/optimization/oprofile/oprofilemanual.pdf.

Li H., Yang C. et Tseng H., « Energy-aware flash memory management in virtual memory system », *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, nº 8, p. 952–964, 2008, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4570467.

Lim S. et Park K., « An efficient NAND flash file system for flash memory storage », *Computers, IEEE Transactions on*, vol. 55, nº 7, p. 906–912, 2006, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1637405.

Lim S., Baek S., Hwang J. et Park K., Write back routine for JFFS2 efficient I/O, Dans *Embedded and Ubiquitous Computing*, Sha E., Han S., Xu C., Kim M., Yang L.T. et Xiao B. (coordinateurs), nº 4096 de « Lecture Notes in Computer Science », p. 795–804, Springer Berlin Heidelberg, jan. 2006, ISBN 978-3-540-36679-9, 978-3-540-36681-2, adresse : http://link.springer.com/chapter/10.1007/11802167_80.

Lindner M., libconfig–c/c++ configuration file library, adresse : http://www.hyperrealm.com/libconfig/, 2012.

Linux Contributors, Lseek man page, adresse : http://man7.org/linux/man-pages/man2/lseek.2.html, 2014.

Liu S., Guan X., Tong D. et Cheng X., « Analysis and comparison of NAND flash specific file systems », *Chinese Journal of Electronics*, vol. 19, nº 3, 2010, adresse : http://mprc.pku.cn/~tongdong/papers/2010sliu.CJE10.pdf.

Macronix, Nand error correction codes introduction, 2014.

Manning C., How YAFFS works, adresse : http://users.actrix.co.nz/manningc/yaffs-docs/HowYaffsWorks.pdf, 2010.

Mariano M., Ecc options for improving nand flash memory reliability, 2012.

Mathur G., Desnoyers P., Ganesan D. et Shenoy P., « Capsule : an energy-optimized object storage system for memory-constrained sensor devices », Dans *Proceedings of the 4th international conference on Embedded networked sensor systems*, p. 195–208, ACM, 2006, adresse : http://dl.acm.org/citation.cfm?id=1182827.

Mathur G., Desnoyers P., Chukiu P., Ganesan D. et Shenoy P., « Ultra-low power data storage for sensor networks », *ACM Transactions on Sensor Networks (TOSN)*, vol. 5, nº 4, p. 33, 2009, adresse : http://dl.acm.org/citation.cfm?id=1614385.

Memik G., Mangione-Smith W.H. et Hu W., « Netbench : A benchmarking suite for network processors », Dans *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, p. 39–42, IEEE Press, 2001, adresse : http://dl.acm.org/citation.cfm?id=603103.

Mesnier M., Intel open storage toolkit, adresse : http://sourceforge.net/projects/intel-iscsi/, 2011.

MICRON INC., NAND flash performance increase using the micron PAGE READ CACHE MODE command, adresse : http://www.micron.com/-/media/documents/products/technical%20note/nand%20flash/tn2901.pdf, 2004.

MICRON INC., « Small-Block vs. Large-Block NAND flash devices », Rapport technique TN-29-07, adresse : https://www.micron.com/-/media/documents/products/technical%20note/nand%20flash/tn2907.pdf, 2005.

MICRON INC., NAND flash performance increase with PROGRAM PAGE CACHE MODE command, adresse : http://www.micron.com/-/media/documents/products/technical%20note/nand%20flash/tn2914.pdf, 2006.

MICRON INC., MT29F2G16ABDHC-ET :D NAND flash memory datasheet, adresse : http://media.digikey.com/pdf/Data%20Sheets/Micron%20Technology%20Inc%20PDFs/MT29F2G(08,16)AAD,ABD.pdf, 2007.

MICRON INC., NAND flash and mobile LPDDR 168-Ball Package-on-Package (PoP) MCP combination memory (TI OMAP) datasheet, 2009.

MICRON INC., Micron NAND flash memory 8Gb,16Gb : x8, x16 NAND flash memory features, adresse : http://e2e.ti.com/cfs-file.ashx/__key/telligent-evolution-components-attachments/00-716-00-00-00-23-31-42/Micron-8Gb-RevC-NAND-Flash.pdf, 2010.

MICROSOFT RESEARCH, SSD extension for DiskSim simulation environment, adresse : http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/, 2009.

MISTRAL SOLUTIONS, OMAP35x evaluation module, adresse : http://www.mistralsolutions.com/pes-products/development-platforms/omap35x-evm-.html, 2013.

MOHAMMAD M. et SALUJA K., « Flash memory disturbances : modeling and test », Dans *VLSI Test Symposium, 19th IEEE Proceedings on. VTS 2001*, p. 218–224, 2001, doi : 10.1109/VTS.2001.923442.

MOHAN V., *Modeling the Physical Characteristics of NAND FLASH Memory*, Thèse de doctorat, University of Virginia, adresse : http://www.cs.virginia.edu/~vm9u/files/Master's%20Thesis.pdf, 2010.

MOHAN V., GURUMURTHI S. et STAN M.R., « FlashPower : a detailed power model for NAND flash memory », Dans *Proceedings of the Conference on Design, Automation and Test in Europe*, p. 502–507, European Design and Automation Association, 2010, adresse : http://dl.acm.org/citation.cfm?id=1871046.

MOHAN V., BUNKER T., GRUPP L., GURUMURTHI S., STAN M.R. et SWANSON S., « Modeling power consumption of NAND flash memories using FlashPower », *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, nº 7, p. 1031–1044, 2013, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6532423.

MTD CONTRIBUTORS, JFFS2 documentation - MTD website, adresse : http://www.linux-mtd.infradead.org/doc/jffs2.html, 2005.

MTD CONTRIBUTORS, NandSim linux flash simulator, adresse : http://www.linux-mtd.infradead.org/faq/nand.html, 2008.

MTD CONTRIBUTORS, MTD general documentation - can i mount ext2 over an MTD device - MTD website, adresse : http://www.linux-mtd.infradead.org/faq/general.html#L_ext2_mtd, 2009.

NATIONAL INSTRUMENTS, Carte d'acquisition NI PXI-4472B, adresse : http://sine.ni.com/nips/cds/print/p/lang/fr/nid/12184, 2014.

NGUYEN D.T., « Evaluating impact of storage on smartphone energy efficiency », Dans *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, p. 319–324, ACM, 2013, adresse : http://dl.acm.org/citation.cfm?id=2501083.

NORCOTT W.D. et CAPPS D., Iozone filesystem benchmark, adresse : http://www.iozone.org/, 2003.

Olivier P., Boukhobza J. et Senn E., « Modeling driver level NAND flash memory I/O performance and power consumption for embedded linux », Dans *2013 11th International Symposium on Programming and Systems (ISPS)*, p. 143–152, 2013a, doi : 10.1109/ISPS.2013.6581480.

Olivier P., Boukhobza J. et Senn E., « Toward a Unified Performance and Power Consumption NAND Flash Memory Model of Embedded and Solid State Secondary Storage Systems », *ArXiv e-prints*, 2013b.

Olivier P., Boukhobza J. et Senn E., « Micro-benchmarking flash memory File-System wear leveling and garbage collection : A focus on initial state impact », Dans *Proceedings of the 2012 IEEE 15th International Conference on Computational Science and Engineering*, vol. CSE '12, p. 437–444, Washington, DC, USA, IEEE Computer Society, 2012a, ISBN 978-0-7695-4914-9, doi : 10.1109/ICCSE.2012.67, adresse : http://dx.doi.org/10.1109/ICCSE.2012.67.

Olivier P., Boukhobza J. et Senn E., « Performance evaluation of flash file systems », *arXiv :1208.6390 [cs]*, 2012b, adresse : http://arxiv.org/abs/1208.6390, arXiv : 1208.6390.

Olivier P., Boukhobza J. et Senn E., « On benchmarking embedded linux flash file systems », *ACM SIGBED Review*, vol. 9, n° 2, p. 43–47, 2012c, adresse : http://dl.acm.org/citation.cfm?id=2318844.

Olivier P., Boukhobza J. et Senn E., Flash-Based storage in embedded systems, Dans *Embedded Computing Systems : Applications, Optimization, and Advanced Design*, Khalgui M., Mosbahi O. et Valentini G. (coordinateurs), p. 439–455, IGI Global, 2013c, ISBN 9781466639225, 9781466639232, adresse : http://www.igi-global.com/chapter/flash-based-storage-embedded-systems/76969.

Olivier P., Boukhobza J. et Senn E., « Flashmon v2 : Monitoring raw NAND flash memory I/O requests on embedded linux », *SIGBED Rev.*, vol. 11, n° 1, p. 38–43, 2014a, ISSN 1551-3688, doi : 10.1145/2597457.2597462, adresse : http://doi.acm.org/10.1145/2597457.2597462.

Olivier P., Boukhobza J. et Senn E., « Revisiting read-ahead efficiency for raw nand flash storage in embedded linux », *Proceedings of the 4th Embed With Linux (EWiLi) Workshop*, 2014b.

Olivier P., Boukhobza J., Soula M., Le Grand M., Chaib Draa I. et Senn E., « A tracing toolset for embedded linux flash file systems », Dans *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, Bratislava, Slovaquie, 2014c, (A paraître).

ONFI Workgroup, « Open NAND flash interface specification - revision 4.0 », Rapport technique, adresse : http://www.onfi.org/-/media/onfi/specs/onfi_4_0%20gold.pdf, 2014.

Opdenacker M., Flash filesystem benchmarks, adresse : http://elinux.org/images/d/d7/Elce2010-flash-filesystems.pdf, 2010.

Open People contributors, Wiki TRAC d'Open-PEOPLE, adresse : https://dev.open-people.fr/, 2012a.

Open People contributors, Site web général d'Open-PEOPLE, adresse : https://www.open-people.fr, 2012b.

OpenBenchmarking.com contributors, Aio-Stress test profile - OpenBenchmarking.com, adresse : http://openbenchmarking.org/test/pts/aio-stress, 2013.

OpenSSD contributors, OpenSSD project, adresse : http://www.openssd-project.org/wiki/The_OpenSSD_Project, 2014.

Ou Y. et Härder T., Trading memory for performance and energy, Dans *Database Systems for Adanced Applications*, p. 241–253, Springer, 2011, adresse : http://link.springer.com/chapter/10.1007/978-3-642-20244-5_24.

Ouni B., *High-level energy characterization, modeling and estimation for OS-based platforms*, Thèse de doctorat, Université de Nice - Sophia Antipolis, 2013.

Pallister J., Eder K., Hollis S.J. et Bennett J., « A high-level model of embedded flash energy consumption », *arXiv preprint arXiv :1404.1602*, 2014, adresse : http://arxiv.org/abs/1404.1602.

PARK A., BECKER J.C. et LIPTON R.J., « IOStone : a synthetic file system benchmark », *ACM SIGARCH Computer Architecture News*, vol. 18, nº 2, p. 45–52, 1990, adresse : http://dl.acm.org/citation.cfm?id=88242.

PARK C., KANG J., PARK S. et KIM J., « Energy-aware demand paging on NAND flash-based embedded storages », Dans *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, vol. ISLPED '04, p. 338–343, New York, NY, USA, ACM, 2004, ISBN 1-58113-929-2, doi : 10.1145/1013235.1013317, adresse : http://doi.acm.org/10.1145/1013235.1013317.

PARK J., YOO S., LEE S. et PARK C., Power modeling of solid state disk for dynamic power management policy design in embedded systems, Dans *Software Technologies for Embedded and Ubiquitous Systems*, LEE S. et NARASIMHAN P. (coordinateurs), nº 5860 de « Lecture Notes in Computer Science », p. 24–35, Springer Berlin Heidelberg, jan. 2009, ISBN 978-3-642-10264-6, 978-3-642-10265-3, adresse : http://link.springer.com/chapter/10.1007/978-3-642-10265-3_3.

PARK S.O. et KIM S.J., « ENFFiS : an enhanced NAND flash memory file system for mobile embedded multimedia system », *ACM Trans. Embed. Comput. Syst.*, vol. 12, nº 2, p. 23 :1–23 :13, fév. 2013, ISSN 1539-9087, doi : 10.1145/2423636.2423641, adresse : http://doi.acm.org/10.1145/2423636.2423641.

PARK S.Y., JUNG D., KANG J.U., KIM J.S. et LEE J., « CFLRU : a replacement algorithm for flash memory », Dans *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, p. 234–241, ACM, 2006a, adresse : http://dl.acm.org/citation.cfm?id=1176789.

PARK S., LEE T. et CHUNG K., A flash file system to support fast mounting for NAND flash memory based embedded systems, Dans *Embedded Computer Systems : Architectures, Modeling, and Simulation*, p. 415–424, Springer, 2006b, adresse : http://link.springer.com/chapter/10.1007/11796435_42.

PARK Y., LIM S., LEE C. et PARK K.H., « PFFS : a scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash », Dans *Proceedings of the 2008 ACM symposium on Applied computing*, p. 1498–1503, ACM, 2008, adresse : http://dl.acm.org/citation.cfm?id=1364038.

R. CORE TEAM, « R : A language and environment for statistical computing », 2012, adresse : http://cran.case.edu/web/packages/dplR/vignettes/timeseries-dplR.pdf.

RAJOVIC N., CARPENTER P.M., GELADO I., PUZOVIC N., RAMIREZ A. et VALERO M., « Supercomputing with commodity cpus : Are mobile socs ready for hpc ? », Dans *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, vol. SC '13, p. 40 :1–40 :12, New York, NY, USA, ACM, 2013, ISBN 978-1-4503-2378-9, doi : 10.1145/2503210.2503281, adresse : http://doi.acm.org/10.1145/2503210.2503281.

RAO S., HEGER D. et PRATT S., « Examining linux 2.6 Page-Cache performance », Dans *Proceedings of the 2005 Linux Symposium*, Ottawa, Canada, 2005, adresse : https://www.kernel.org/doc/ols/2005/ols2005v2-pages-87-98.pdf.

REARDON J., CAPKUN S. et BASIN D.A., « Data node encrypted file system : Efficient secure deletion for flash memory », Dans *USENIX Security Symposium*, p. 333–348, 2012, adresse : https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final74.pdf.

RICHTER D., Fundamentals of Non-Volatile memories, Dans *Flash Memories*, nº 40 de « Springer Series in Advanced Microelectronics », p. 5–110, Springer Netherlands, jan. 2014a, ISBN 978-94-007-6081-3, 978-94-007-6082-0, adresse : http://link.springer.com/chapter/10.1007/978-94-007-6082-0_2.

RICHTER D., Fundamentals of reliability for flash memories, Dans *Flash Memories*, nº 40 de « Springer Series in Advanced Microelectronics », p. 149–166, Springer Netherlands, jan. 2014b, ISBN 978-94-007-6081-3, 978-94-007-6082-0, adresse : http://link.springer.com/chapter/10.1007/978-94-007-6082-0_4.

RIEL R.V., « Page replacement in linux 2.4 memory management », Dans *Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference*, p. 165–172, Berkeley, CA, USA, USENIX Association, 2001, ISBN 1-880446-10-3, adresse : http://dl.acm.org/citation.cfm?id=647054.715629.

Roohparvar F.F., United states patent : 7366013 - single level cell programming in a multiple level cell non-volatile memory device, adresse : http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/7366013, avr. 2008.

Rosenblum M. et Ousterhout J.K., « The design and implementation of a log-structured file system », *ACM Transactions on Computer Systems (TOCS)*, vol. 10, nº 1, p. 26–52, 1992, adresse : http://dl.acm.org/citation.cfm?id=146943.

Ross K.A., « Modeling the performance of algorithms on flash memory devices », Dans *Proceedings of the 4th international workshop on Data management on new hardware*, p. 11–16, ACM, 2008, adresse : http://dl.acm.org/citation.cfm?id=1457153.

Rostedt S., Ftrace documentation, adresse : https://www.kernel.org/doc/Documentation/trace/ftrace.txt, 2008.

Russinovich M. et Cogswell B., FileMon for windows v7. 04, adresse : http://technet.microsoft.com/fr-fr/sysinternals/bb896642.aspx, 2006.

Sakui K. et Suh K.d., NAND flash memory technology, Dans *Nonvolatile Memory Technologies with Emphasis on Flash*, Brewer J. et Gill M. (coordinateurs), p. 223–311, John Wiley & Sons, Inc., 2007, ISBN 9780470181355, adresse : http://onlinelibrary.wiley.com/doi/10.1002/9780470181355.ch6/summary.

Samsung Electronics Co., Samsung K9XXG08XXM NAND flash chip datasheet, 2007.

Samsung Electronics Co., Samsung PM800 solid state drive datasheet, 2009.

Saxena M. et Swift M.M., « FlashVM : virtual memory management on flash. », Dans *USENIX Annual Technical Conference*, 2010, adresse : https://www.usenix.org/legacy/event/usenix10/tech/full_papers/Saxena.pdf.

Schall D., Hudlet V. et Härder T., « Enhancing energy efficiency of database applications using SSDs », Dans *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, p. 1–9, ACM, 2010, adresse : http://dl.acm.org/citation.cfm?id=1822328.

Schierl A., Schellhorn G., Haneberg D. et Reif W., Abstract specification of the UBIFS file system for flash memory, Dans *FM 2009 : Formal Methods*, p. 190–206, Springer, 2009, adresse : http://link.springer.com/chapter/10.1007/978-3-642-05089-3_13.

Senn E., Chillet D., Zendra O., Belleudy C., Bilavarn S.B., Atitallah R.B., Samoyeau C. et Fritsch A., « Open-people : Open power and energy optimization PLatform and estimator », Dans *Digital System Design (DSD), 2012 15th Euromicro Conference on*, p. 668–675, IEEE, 2012, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6386956.

Seo E., Park S. et Urgaonkar B., « Empirical analysis on energy efficiency of flash-based SSDs. », Dans *HotPower*, 2008, adresse : https://www.usenix.org/event/hotpower08/tech/full_papers/seo/seo_html/.

Serial ATA International Organization, SATA-ST 3.1 - serial ATA re- vision 3.1, adresse : http://www.knowledgetek.com/graphicsNew/SerialATA_Revision_3_1_Gold-KT.pdf, 2011.

Shin S. et Shin D., « Power analysis for flash memory SSD », *Work-shop for Operating System Support for Non-Volatile RAM (NVRAMOS 2010 Spring)(Jeju, Korea, April 2010)*, 2010, adresse : http://dcslab.hanyang.ac.kr/nvramos10spring/documents/20_doc/NVRAMOS-shin.pdf.

Shinohara T., United states patent : 5905993 - flash memory card with block memory address arrangement, adresse : http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5905993, mai 1999.

Skadron K., Stan M.R., Huang W., Velusamy S., Sankaranarayanan K. et Tarjan D., « Temperature-aware computer systems : Opportunities and challenges », *IEEE Micro*, vol. 23, nº 6, p. 52–61, 2003.

SMITH K., Benchmarking ssds : The devil is in the preconditioning details, Flash Memory Summit, 2009, http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2009/20090811_F2A_Smith.pdf.

SNIA, Storage network industry association iotta trace repository, 2011, http://iotta.snia.org/.

SONG H., CHOI S., CHA H. et HA R., « Improving energy efficiency for flash memory based embedded applications », *Journal of Systems Architecture*, vol. 55, nº 1, p. 15–24, jan. 2009, ISSN 1383-7621, doi : 10.1016/j.sysarc.2008.07.004, adresse : http://www.sciencedirect.com/science/article/pii/S1383762108001100.

SOWA K., Choosing a linux file system for flash memory devices, 2011.

SPC CONTRIBUTORS, « SPC benchmark 1 specifications », Rapport technique, adresse : http://www.storageperformance.org/specs/SPC-1_SPC-1E_v1.14.pdf, 2013.

SPEC CONTRIBUTORS, « SPECSFS2008 user guide », Rapport technique, adresse : http://www.spec.org/sfs2008/docs/usersguide.pdf, 2008.

ST MICROELECTRONICS, « Bad block management in NAND flash memories », Application Note AN1819, adresse : http://www.eetasia.com/ARTICLES/2004NOV/A/2004NOV29_MEM_AN06.PDF?SOURCES=DOWNLOAD, 2004.

STALLMAN R.M. et PESCH R.H., *Using GDB : A guide to the GNU source-level debugger*, Free software foundation, 1991.

STEFFEN J.L. ET COLL., « Interactive examination of a c program with cscope », Dans *Proceedings of the USENIX Winter Conference*, p. 170–175, 1985.

SU X., JIN P., XIANG X., CUI K. et YUE L., « Flash-DBSim : a simulation tool for evaluating flash-based database algorithms », Dans *2nd IEEE International Conference on Computer Science and Information Technology, 2009. ICCSIT 2009*, p. 185–189, août 2009, doi : 10.1109/ICCSIT.2009.5234967.

TEHRANI S., Advancement in charge trap flash memory technology, adresse : http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130813_Plenary_Tehrani.pdf, 2013.

TOSHIBA INC., NAND vs. NOR flash memory technology overview, adresse : http://umcs.maine.edu/~cmeadow/courses/cos335/Toshiba%20NAND_vs_NOR_Flash_Memory_Technology_Overviewt.pdf, 2006.

TOSHIBA INC., Evaluation of UBI and UBIFS, adresse : http://elinux.org/images/f/f8/CELFJamboree30-UBIFS_update.pdf, 2009.

TPC CONTRIBUTORS, Transaction performance council benchmarks webpage, adresse : http://www.tpc.org/information/benchmarks.asp, 2014.

TRAEGER A., ZADOK E., JOUKOV N. et WRIGHT C.P., « A nine year study of file system and storage benchmarking », *ACM Transactions on Storage (TOS)*, vol. 4, nº 2, p. 5, 2008, adresse : http://dl.acm.org/citation.cfm?id=1367831.

TRIDGELL A., DBench benchmark documentation, adresse : ftp://samba.org/pub/tridge/dbench/README, 2008.

TSENG H., LI H. et YANG C., « An energy-efficient virtual memory system with flash memory as the secondary storage », Dans *Low Power Electronics and Design, 2006. ISLPED'06. Proceedings of the 2006 International Symposium on*, p. 418–423, IEEE, 2006, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4271879.

TSENG H., GRUPP L.M. et SWANSON S., « Underpowering NAND flash : Profits and perils », Dans *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, p. 1–6, IEEE, 2013, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6560755.

UBI Developers, UBI FAQ - MTD website, adresse : http://www.linux-mtd.infradead.org/faq/ubi.html, 2009a.

UBI Developers, MTD website - UBI documentation - NAND flash sub-pages, http ://www.linux-mtd.infradead.org/doc/ubi.html#L_subpage, adresse : http://www.linux-mtd.infradead.org/doc/ubi.html#L_subpage, 2009b.

UBIFS Contributors, Mtd website - raw flash versus ftl devices, 2009.

UBM Tech, Embedded market study, 2013, http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D_2013EmbeddedMarketStudyb.pdf.

UCAR, UCAR metarates benchmark, adresse : http://www.cisl.ucar.edu/css/software/metarates, 2004.

UMASS, Umass (university of massachusetts) storage trace repository, adresse : http://traces.cs.umass.edu/index.php/Storage/Storage, 2009.

Vandenbergh H., Vdbench users guide, adresse : http://www.oracleimg.com/technetwork/server-storage/vdbench-1901683.pdf, 2012.

Wang Y., Shu J., Xue W. et Xue M., « VFS interceptor : Dynamically tracing file system operations in real environments », Dans *First International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008). Held in conjunction with HPCA*, 2008.

Wei M.Y.C., Grupp L.M., Spada F.E. et Swanson S., « Reliably erasing data from Flash-Based solid state drives. », Dans *FAST*, vol. 11, p. 8–8, 2011, adresse : http://static.usenix.org/legacy/events/fast11/tech/full_papers/Wei.pdf.

Wilson A., « The new and improved FileBench », Dans *Proceedings of 6th USENIX Conference on File and Storage Technologies*, 2008, adresse : https://www.usenix.org/conference/fast-08/new-and-improved-filebench.

Wilton S.J. et Jouppi N.P., « CACTI : an enhanced cache access and cycle time model », *Solid-State Circuits, IEEE Journal of*, vol. 31, nº 5, p. 677–688, 1996, adresse : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=509850.

Wolman B. et Olson T.M., « IOBENCH : a system independent IO benchmark », *SIGARCH Comput. Archit. News*, vol. 17, nº 5, p. 55–70, sept. 1989, ISSN 0163-5964, doi : 10.1145/71302.71309, adresse : http://doi.acm.org/10.1145/71302.71309.

Wong G., *Inside Solid State Drives (SSDs)*, chap. SSD Market Overview, adresse : http://www.springer.com/materials/book/978-94-007-5145-3, 2014.

Woodhouse D., « JFFS2 : the journalling flash file system version 2 », Dans *Ottawa Linux Symposium*, Ottawa, Canada, 2001, adresse : https://sourceware.org/jffs2/jffs2.pdf.

Wookey, YAFFS - a NAND flash file system, adresse : http://wookware.org/talks/yaffscelf2007.pdf, 2007.

Wu M. et Zwaenepoel W., « eNVy : a non-volatile, main memory storage system », Dans *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. ASPLOS VI, p. 86–97, New York, NY, USA, ACM, 1994, ISBN 0-89791-660-3, doi : 10.1145/195473.195506, adresse : http://doi.acm.org/10.1145/195473.195506.

YAFFS2 Contributors, Google android - YAFFS2 website, adresse : http://www.yaffs.net/google-android, 2012.

Yoo B., WON Y., CHOI J., YOON S., CHO S. et KANG S., « SSD characterization : From energy consumption's perspective », Dans *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, vol. HotStorage'11, p. 3–3, Berkeley, CA, USA, USENIX Association, 2011, adresse : http://dl.acm.org/citation.cfm?id=2002218.2002221.

Yoo S. et PARK C., Low power mobile storage : SSD case study, Dans *Energy-Aware System Design*, p. 223–246, Springer, 2011, adresse : http://link.springer.com/10.1007/978-94-007-1679-7_9.