# Simulation générique et contribution à l'optimisation de la robustesse des systèmes de données à large échelle

Sebastien Gougeaud

NNT : 2017SACLV011

# Thèse de doctorat de l'Université Paris-Saclay préparée Université de Versailles Saint-Quentin en Yvelines

Ecole doctorale n°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat : Informatique

par

**M. Sébastien Gougeaud**

## Simulation générique et contribution à l'optimisation de la robustesse des systèmes de stockage de données à large échelle

Thèse présentée et soutenue à Versailles,
le 11 mai 2017.

Composition du Jury :

| | | |
|---|---|---|
| Mme | Joanna Tomasik<br>Professeur | Président du Jury<br>CentraleSupélec |
| M. | Jacques Jorda<br>Maître de conférences – HDR | Rapporteur<br>IRIT, Université de Toulouse |
| M. | Raymond Namyst<br>Professeur | Rapporteur<br>LaBRI, Université de Bordeaux |
| M. | Jacques-Charles Lafoucrière<br>Responsable SI Scientifique et Réseaux | Examinateur<br>CEA-DAM |
| M. | Philippe Couvée<br>Responsable R&D Data Management | Examinateur<br>Atos |
| M. | William Jalby<br>Professeur | Directeur de thèse<br>UVSQ |
| Mme. | Soraya Zertal<br>Maître de conférences | Co-encadrante de thèse<br>UVSQ |

*Petite tribune à ceux qui ont permis l'existence de ce manuscrit, et des travaux qu'il contient :*

- *à Soraya Zertal, qui est à l'origine de cette aventure en me proposant le sujet de cette thèse un jour d'été 2013, et qui m'a encadré à chaque instant, me prodiguant conseils et méthodes de travail et m'offrant sa confiance dans le travail de recherche et d'enseignement ;*
- *à William Jalby, pour avoir dirigé ma thèse et pour l'accueil qu'il m'a fait au sein du laboratoire de recherche ;*
- *à Jacques-Charles Lafoucrière et Philippe Deniel, pour leur confiance et leur contribution à ma recherche ;*
- *aux membres du jury de thèse, pour l'intérêt porté à mes travaux et leur divers retours ;*

- *à Antoine, Florent, Colin et Yohan, mes collègues de bureau à travers ces trois années ;*
- *à Mihail, Thomas et Christopher, camarades de Master s'étant également engagés dans cette expérience qu'est le doctorat ;*
- *à Benoît et Nicolas, mes premiers contacts avec le monde de la recherche, qui m'ont donné envie d'embrasser cette voie ;*

- *à Pablo, Thomas et de manière générale aux membres du laboratoire, pour les diverses discussions qui ont rythmé mes journées ;*

- *et enfin, à ma femme, ma mère, ma famille et mes amis, qui ont fait de moi ce que je suis aujourd'hui.*

*Je vous dis merci.*

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

**Mots clés :** Simulation, Stockage de données à large échelle, Robustesse, Disques magnétiques, Disques à mémoire Flash

**Résumé :** La capacité des systèmes de stockage de données ne cesse de croître pour atteindre actuellement l'échelle de l'exaoctet en ce qui concerne les centres de données. Cette croissance s'explique par le nombre grandissant de disques utilisés au sein d'une baie de stockage, mais également par leur capacité, pouvant désormais dépasser la dizaine de téraoctets. Ce constat a un réel impact sur la robustesse des systèmes de stockage. Effectivement, plus le nombre de disques contenus dans un système est grand, plus la probabilité que l'un de ses disques devienne défaillant est importante. De même, le temps utilisé pour la reconstruction d'un disque suite à une défaillance est proportionnel à sa capacité. Le développement de nouveaux mécanismes de robustesse et l'importance de quantifier cette robustesse au même titre que les métriques de performance deviennent donc décisifs.

La simulation est un outil qui permet le test de nouveaux mécanismes dans des conditions quasi réelles. Or, la littérature n'offre pas d'outils supportant des systèmes de stockage représentatifs des systèmes actuels, à savoir hétérogènes et à large échelle. L'hétérogénéité d'un système est définie par l'utilisation d'une association de disques issues de technologies différentes, comme par exemple pour les systèmes actuels, une association de disques durs magnétiques (HDD) et de disques basés sur la mémoire Flash (SSD).

*Open and Generic data Storage system Simulation tool* (OGSSim), l'outil de simulation que nous proposons, supporte l'hétérogénéité et la taille importante de ces systèmes. OGSSim est un logiciel parallèle calculant, pour une configuration matérielle et un jeu de requêtes donnés, différentes métriques de performances. Sa décomposition modulaire permet d'entreprendre chaque technologie de stockage, schéma de placement ou modèle de calcul comme des briques qui peuvent être ajoutées et combinées entre elles pour paramétrer au mieux la simulation. L'idée d'ouverture incorporée à OGSSim donne également la possibilité à n'importe quel utilisateur d'y ajouter ses propres algorithmes pour pouvoir conduire des tests. La première étape de validation d'OGSSim a donné des résultats satisfaisants, d'où le dépôt de la première version du logiciel sous licence LGPL.

La robustesse étant un paramètre critique dans ces systèmes, nous utilisons le *declustered RAID* pour l'implémenter et assurer ainsi la distribution de la reconstruction des données d'un disque en cas de défaillance. Il consiste en la subdivision en blocs de données d'un système de stockage logique, et de leur placement selon un schéma défini, sur un système physique. L'efficacité de ce mécanisme dépend directement du schéma de placement utilisé, et des contraintes que le schéma cherche à respecter. On peut séparer les schémas actuels en deux catégories : ceux utilisant des réplicas (copie exacte de la donnée) de blocs de données comme la méthode Crush, et ceux n'en utilisant pas. Nous proposons le *Symmetric Difference of Source Sets* (SD2S), un algorithme de création de schéma de placement basé sur le respect de quatre contraintes que nous

fixons axées sur le parallélisme d'accès et la fiabilité du système après une ou plusieurs défaillances. La création du schéma repose sur le décalage des blocs de données, *stripe* après *stripe*, selon un pas $\lambda$. Ce $\lambda$ est déterminé par le calcul de la matrice des degrés, utilisée pour évaluer la proximité des ensembles de provenance logique des blocs d'un disque physique.

Pour quantifier l'efficacité de SD2S, nous l'avons implémenté au sein d'OGSSim, et testé dans différentes configurations. En comparant SD2S à la méthode Crush, exempté des réplicas, nous montrons d'une part que la création du schéma de placement, aussi bien en mode normal qu'en mode défaillant, est plus rapide en faveur de SD2S, et d'autre part que le coût en espace mémoire est également réduit, voire nul si le système ne subit aucune défaillance. De plus, en cas de double défaillance, SD2S assure la sauvegarde d'une partie, voire de la totalité, des données.

Cette thèse a permis de développer OGSSim, notre outil de simulation pour les systèmes de stockage, et SD2S, un algorithme optimisant la robustesse de larges systèmes de stockage. Des extensions à ces travaux sont prévus, avec une ouverture sur le parallélisme multi-noeud pour OGSSim, et l'optimisation de la fraction de données sauvegardées en cas de multiples défaillances pour SD2S.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

**Title : Generic Simulation and Contribution to the Robustness Optimization of Large-Scale Data Storage Systems**

**Keywords :** Simulation, Large-Scale Data Storage, Robustness, Hard Disk Drives, Solid-State Drives

**Summary :** Capacity of data storage systems does not cease to increase to currently reach the exabyte scale for data centers. This growth is explained by the expanding number of disks used in a storage array, but also by the drives capacity, whose now exceed ten terabytes. This observation gets a real impact over storage system robustness. Actually, the more the number of disks in a system is, the greater the probability of a failure happening is. Also, the time used for a disk reconstruction following its failure is proportional to its capacity. Thus, developing new robustness mechanisms and quantifying this robustness as performance metrics become decisive.

Simulation is an appropriate technique to test new mechanisms in near real conditions. Yet, literature does not offer tools which can handle representative and modern storage systems, namely heterogeneous and large-scales ones. The system heterogeneity is defined by the use of disks from different technologies. For instance, in current systems, the combination of hard disk drives (HDD) and solid-state drives (SSD) is unavoidable.

We propose a new software we call Open and Generic data Storage system Simulation tool (OGSSim). It handles the heterogeneity and the large size of these modern systems. OGSSim is a parallel software which computes, for a given hardware configuration and requests set, different performance metrics. Its modularity permits the undertaking of each storage technology, placement scheme or computation model as bricks which can be added and combined to optimally configure the simulation. OGSSim is an open software which means that it allows any user to add its own algorithms to conduct experiments. The first step of OGSSim validation against real systems returns satisfying results, which leads us to deposite the first version of OGSSim under LGPL license.

Robustness is a critical issue for these systems. So we use the declustered RAID to implement and ensure the data distribution and reconstruction of a disk in case of a failure. It is composed of two steps. First, the logical data storage system is divided into blocks. Then, the blocks are distributed across a physical system. The efficiency of this mechanism depends directly on the used placement scheme, and on the constraints the scheme has to respect. Current schemes can be separated into two categories: those using replicas like Crush method, and those without replicas. We propose the Symmetric Difference of Source Sets (SD2S) algorithm to achieve the placement scheme of data blocks. This placement scheme is based upon the respect of four constraints dealing with the disks and stripes parallelism, the system reliability after one or multiple failures and the load balancing. The placement scheme creation lies on shifting data

blocks, stripe by stripe, according to an offset $\lambda$. This $\lambda$ is determined by computing the degree matrix, used to evaluate the distance between logical source sets of physical disk blocks.

To evaluate the SD2S efficiency, we implemented it into OGSSim, and conducted experiments under different configurations. The comparison of SD2S to Crush method without replicas results into two outcomes. First, the placement scheme creation in normal and failure modes is faster with SD2S. Second, the memory space cost of SD2S is greatly reduced (null if there is no failure). Furthermore, SD2S ensures the partial, if not the total, reconstruction of data in case of multiple failures.

The main contributions of this thesis is the development of OGSSim, our simulation tool for large and heterogeneous data storage systems, and our SD2S algorithm which enhances the robustness of large-scale storage systems. Extensions of these works are planned, as a multi-node parallel version of OGSSim, and the SD2S optimization in case of multiple failures.

# Table des matières

# Table des figures

# Table des tableaux

# Introduction

## Contexte

De nos jours, le stockage des données est devenu sans aucun conteste une problématique qui concerne chaque utilisateur. Que ce soit pour une utilisation nomade, avec les *smartphones*, les cartes SD, les clés USB ou de manière plus conventionnelle avec les disques magnétiques, ceux basés sur la mémoire Flash. De plus, l'avènement des centres de données donne une autre dimension aux besoins, en terme de capacité et de robustesse.

Ces deux dernières années, deux leaders dans le marché du stockage, à savoir HGST et Samsung ont présenté respectivement un modèle de HDD et de SSD possédant une capacité de 10 (HGST, 2015) et de 16 teraoctets (To) (Samsung, 2016a). Seagate a quant à lui annoncé la commercialisation prochaine d'un SSD de 60 To de données (Seagate, 2016). L'arrivée sur le marché et la popularisation du *cloud* et des serveurs de stockage en réseau NAS (*Network Attached Storage*) affirment également cette tendance.

Les systèmes de stockage à large échelle, autrement dit les centres de données, possèdent deux moyens d'accroître leur offre de capacité :

— l'augmentation de la capacité des disques qu'ils utilisent ;

— l'augmentation du nombre de disques composant le système.

Les centres de données actuels possèdent de grandes surfaces physiques, permettant l'installation de milliers de disques. Par exemple, deux des plus grands centres de données mondiaux, le Switch SUPERNAP campus (Etats-Unis) et le Range International Group data center (Chine), dont la construction s'est terminée en 2016, s'étendent sur des espaces respectifs de 600'000 $m^2$ et 585'000 $m^2$ (Reno Luxury Homes, 2015; Ligato, 2016).

La croissance en taille des centres de données apporte plusieurs problématiques, dont le coût financier, le coût énergétique dû à l'utilisation des disques et au système de refroidissement ou encore la robustesse du système. En effet, pour cette dernière, chaque disque a une certaine probabilité de subir une défaillance, et l'augmentation de ce nombre de disques fait croître la probabilité que des données soient perdues. De plus, le temps utilisé pour la reconstruction d'un disque étant directement proportionnel à sa taille, l'augmentation de la capacité des disques a donc un impact direct sur la durée du mécanisme de reconstruction. Ces deux constats mènent à la conclusion que sans renforcement de la robustesse par un mécanisme matériel et/ou logiciel, les systèmes de stockage de données sont de plus en plus vulnérables aux défaillances.

La simulation est un moyen permettant l'étude du comportement d'un système défini face à un scénario donné. Elle est à la fois assez représentative du comportement réel du système de stockage des données et permet de tester diverses situations existantes ou non. Simuler plusieurs scénarios sur différents systèmes permet de les comparer, et de choisir la meilleure solution pour une utilisation précise.

Dans le cas de l'étude de la robustesse, la simulation peut être utilisée pour déterminer l'impact d'un événement sur un système défini. En paramétrant le système de stockage simulé de telle sorte à ce qu'il se rapproche d'un système réel, de par les disques utilisés, la configuration adoptée pour ces disques, les bus reliant les différents composants du système, nous pouvons observer son comportement face à une ou plusieurs défaillances et apporter une réponse à cette liste non exhaustive de questions :

— La défaillance d'un élément du système impacte-t-elle l'expérience de l'utilisateur ?
— Quelle est la durée du mécanisme de reconstruction des données ?
— Le temps de réponse des requêtes utilisateur augmente-t-il durant le mécanisme de reconstruction ?
— Y a-t-il perte de données en cas de multiples défaillances successives ?
— Quel est le gain (respectivement perte) par rapport à d'autres expériences ?

Parce que l'utilisateur ne doit pas être lésé des événements survenant sur le système, il faut donc respecter deux contraintes majeures : l'utilisateur ne doit pas (ou peu) subir de latence supplémentaire à la suite d'une défaillance **et** ne doit pas subir de perte de ses données.

# Contributions

Pour répondre à la problématique posée, nous articulons le travail de thèse autour de deux propositions :

1. OGSSim, un outil de simulation générique et ouvert supportant des systèmes de stockage à large échelle ;
2. SD2S, un algorithme de placement de données pour configuration de stockage orientée robustesse, sans utilisation de réplicas.

## Open and Generic data Storage system Simulation tool (OGSSim)

La taille et le coût des systèmes de stockage de données ne cessant de croître, la simulation permet la prédiction de la performance, la robustesse et la rentabilité de ces systèmes. Il est alors nécessaire de disposer d'un outil capable de déduire les meilleures solutions à ces problèmes en analysant leur comportement. *Open and Generic data Storage system Simulation tool* (OGSSim), l'outil de simulation que nous proposons, supporte ces systèmes hétérogènes et à large échelle.

Les systèmes de stockage simulés avec OGSSim respectent une topologie hiérarchique *tier/volume/device*. Chaque niveau de la hiérarchie est relié à l'aide de canaux de communication. Cette topologie peut tout aussi bien représenter des systèmes de stockage composés d'un ou de quelques disques ou bien des systèmes de plus larges échelles de type centre de données.

OGSSim est un logiciel *multi-threadé* et modulaire organisé en trois groupes, ou chaînes, de modules. La première, appelée chaîne d'extraction, récupère les informations contenues dans les différents fichiers d'entrée et renseignant sur le paramétrage de l'outil, la configuration architecturale, les spécifications des disques et le jeu de requêtes à simuler. Chaque type d'information est récupéré par un module spécifique, et les informations sont envoyées, par le biais de ZeroMQ (iMatix Corporation, 2016), à la chaîne de décomposition. Cette chaîne de décomposition assure la traduction logique/physique des requêtes : chaque requête est analysée afin de déterminer quelle configuration est ciblée par cette dernière, puis sera décomposée de telle sorte que chaque sous-requête ne concerne qu'une suite de données contigües d'un disque. La dernière chaîne calcule les métriques de performance liées à l'exécution des requêtes, comme par exemple les temps d'attente et de réponse, le taux d'utilisation des disques, etc. Les résultats de la simulation sont ensuite stockés dans les fichiers de sortie, compilant les valeurs calculées et des graphes de visualisation.

OGSSim a fait l'objet de deux publications dans des conférences internationales. La première (Gougeaud et al., 2015a), pour *International Conference on Simulation Tools and Techniques (SIMUtools)*, présente une première version du logiciel, offrant le support des systèmes de stockage hétérogène à large échelle pour une simulation en mode normal sans événements externes. La seconde (Gougeaud et al., 2016b), pour *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, présente la version actuelle du simulateur, avec l'intégration du mode défaillant. La première publication a également été sélectionnée pour faire partie du journal *EAI Endorsed Transactions on Scalable Information Systems* (Gougeaud et al., 2015b). Un dépot de logiciel pour OGSSim est actuellement en cours, sous licence LGPL.

## Symmetric Difference of Source Sets (SD2S)

La robustesse des systèmes de stockage à large échelle est un point critique, notamment avec l'augmentation en nombre et en capacité des disques. Le recours à un moyen logiciel ou matériel pour renforcer la robustesse devient capital. *Symmetric Difference of Source Sets* (SD2S) est un algorithme de placement pour *declustered RAID* que nous proposons à cet effet. Il s'applique sur une configuration de disques où chacun possède une partie non utilisée (*spare*), servant à stocker les données reconstruites des suites d'une défaillance.

Notre algorithme SD2S, orienté robustesse, minimise la part de données perdues en cas de défaillances multiples successives, grâce à un ensemble défini de quatre contraintes. Trois de ces contraintes se focalisent sur le parallélisme d'accès aux disques, l'accès aux *stripes* (blocs de données faisant partie du même groupe de parité/code d'erreur) et l'équilibrage de charge entre les disques. La dernière contrainte, point névralgique de l'algorithme, demande aux ensembles de provenance des blocs de données de chaque disque physique, d'être au moins partiellement disjoints.

L'algorithme consiste en un schéma de placement à décalage sur les *stripes* d'un pas $\lambda$. Ce pas est déterminé à l'aide d'une structure de données, appelée matrice des degrés, représentant pour chaque couple de disques physiques, la comparaison de leurs ensembles de provenance. L'utilisation du placement à décalage et de la matrice des

degrés assure le respect des contraintes de parallélisme d'accès et de la disjonction partielle des ensembles de provenances, contraintes que nous estimons plus critiques que l'équilibrage de charge.

SD2S a fait l'objet d'une publication (Gougeaud et al., 2016a) pour la conférence internationale *International Conference on High Performance Computing and Simulation* (HPCS), et présentait l'algorithme de construction du schéma de placement en mode normal, en comparant les résultats obtenus avec une méthode basée sur Crush (Li and Goel, 2012).

## Plan

Le manuscrit est donc découpé en deux grandes parties, chacune composée de plusieurs chapitres.

La première partie du document concerne la simulation des systèmes de stockage. Elle démarre par un état de l'art des différentes technologies utilisées dans le stockage, présentant les disques existants et les configurations de stockage rencontrées. Ce chapitre dresse également une liste des simulateurs de systèmes de stockage présents dans la littérature et fait le bilan de leurs points forts et faibles.

Dans le deuxième chapitre, nous présentons OGSSim, notre outil de simulation pour systèmes de stockage de données hétérogènes et à large échelle. Nous y détaillons le schéma conceptuel, le rôle que tient chaque module constituant l'outil ainsi que les communications ayant lieu entre chacun d'entre eux. Sont également présentées les bibliothèques externes utilisées.

Le troisième chapitre porte sur l'implémentation d'OGSSim, avec une description détaillée de chacun des modules le composant. OGSSim est découpée en trois chaînes de modules : la chaîne d'extraction, la chaîne de décomposition et la chaîne de calcul. La première étape de validation est également présentée, confrontant OGSSim à des modèles de SSD et HDD actuels.

Le quatrième chapitre clos la partie simulation, avec une présentation du modèle de communication actuellement implémenté dans OGSSim, ainsi que les problèmes engendrés par l'utilisation du parallélisme dans notre logiciel et les solutions apportées. On peut alors distinguer deux types de problèmes : ceux de synchronisation des données, pour la cohérence des résultats et la bonne prédiction du comportement du système simulé et ceux de mise à l'échelle du logiciel. Le chapitre se termine par une analyse du temps d'exécution d'OGSSim en fonction de la taille du système simulé.

La seconde partie du manuscrit est dédiée au traitement de la robustesse des systèmes de stockage. Elle débute par le cinquième chapitre, qui présente l'intégration du mode défaillant à OGSSim, une description du *declustered RAID*, une organisation de systèmes de stockage axée sur la robustesse des données et enfin un premier algorithme de placement des données axé sur la robustesse du système.

Le sixième et dernier chapitre présente SD2S, notre algorithme de placement de données pour *declustered RAID*, optimisant la robustesse du système de stockage face à de multiples défaillances, ainsi qu'une série de résultats issue de la comparaison entre SD2S et la méthode Crush (Weil et al., 2006a) sans réplicas, en utilisant notre outil de simulation.

# Chapitre 1

# Le stockage de données : état de l'art



*Le stockage est un domaine de recherche qui a été négligé au profit du calcul. Mais comme dans beaucoup de domaines, les évolutions répondent aux besoins sans cesse grandissants des utilisateurs. De nombreuses technologies de supports de stockage se sont succédées pour désormais permettre des capacités de l'ordre de plusieurs téraoctets de données pour un seul disque. Peuplant les centres de données, ils permettent l'exploitation d'exaoctets de données pour l'archivage ou le calcul haute performance.*

*Durant son évolution, plusieurs aspects des systèmes de stockage de données ont été étudiés, tels la robustesse et l'efficacité. La robustesse d'un système de stockage tient en sa capacité à limiter voire supprimer l'impact d'une défaillance matérielle. L'efficacité, elle, consiste en la réduction du temps de réponse du système par l'utilisation de technologies plus rapides et/ou du parallélisme d'exécution des requêtes d'entrée/sortie.*

*Dans ce chapitre, un état de l'art est proposé, commençant par la présentation des différentes technologies de stockage à savoir les deux principales technologies de disques : les disques magnétiques et les disques à base de mémoire Flash. S'ensuit une étude des différents mécanismes de gestion permanente des disques magnétiques et Flash. Nous continuerons avec la présentation des configurations de systèmes de stockage utilisées ainsi que leurs apports en terme de robustesse et de performance. Enfin, les travaux antérieurs sur la simulation de systèmes de stockage seront détaillés.*

# 1 Technologies de disques

Le disque est le composant principal du système de stockage, car c'est lui qui aura la charge de stocker les données de l'utilisateur. Actuellement, deux technologies sont majoritairement utilisées chez les particuliers, les entreprises et les centres de données : les disques magnétiques et les disques Flash.

## 1.1 Disques magnétiques – Hard Disk Drive (HDD)

La topographie du disque magnétique, *Hard Disk Drive* (HDD) (Arpaci-Dusseau and Arpaci-Dusseau, 2015), est hiérarchique. La hiérarchie, représentée sur la figure 1.1, a comme premier composant le cylindre, nommé ainsi de par sa forme géométrique. Le cylindre est composé de pistes ayant la même position sur les différents plateaux, placés les uns sur les autres. A chaque plateau est affiliée une tête de lecture afin d'accéder aux données qui y sont stockées. Cet accès est effectué par la combinaison de deux mécanismes : le déplacement de la tête de lecture entre les pistes internes et externes du plateau, et la rotation du plateau, permettant à la tête d'accéder à n'importe quel secteur d'une piste.

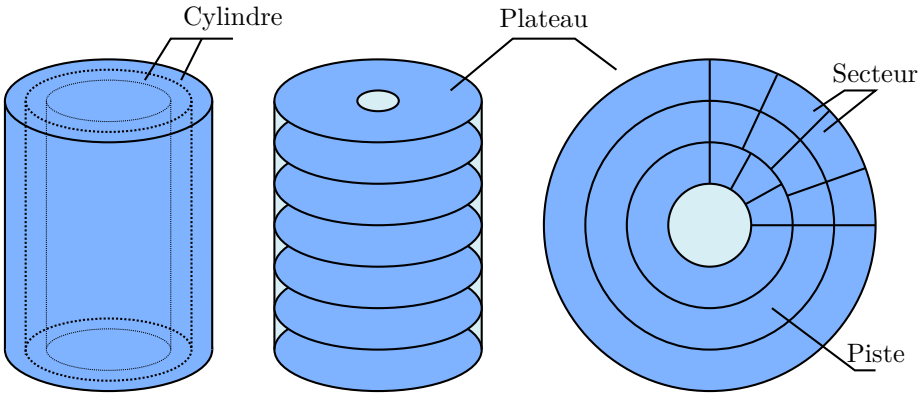



FIGURE 1.1 – Topographie du HDD

Le mapping d'adresses est l'opération qui traduit une adresse logique, connue de l'hôte, en une adresse physique, connue du disque. Plusieurs techniques d'adressage ont été utilisées pour les HDDs. L'adressage *Cylinder Head Sector* (CHS) traduit une adresse logique en une combinaison d'indices de composants (le cylindre, la tête de lecture/écriture et le secteur). Cette combinaison est l'adresse physique du secteur ciblé. La décomposition est donnée dans l'algorithme 1.

Il existe plusieurs manières de découper un plateau de disque magnétique en secteurs : chacune possédant des avantages et des inconvénients. On en distingue deux familles de modèles : les *Constant Angular Velocity* (CAV) et les *Constant Linear Velocity* (CLV). Le CAV utilise une mesure d'angle comme taille de secteur. Dans l'exemple montré figure 1.2a, tous les secteurs ont un angle de 45 degrés, quelque soit la piste à laquelle ils appartiennent. Dans un disque utilisant le modèle CAV, les têtes de lecture bougent à vitesse constante, mais comme chaque secteur contient le même nombre de bits utiles, une partie de l'espace de stockage est inutilisé car les secteurs externes sont plus grands que ceux internes.

**Algorithme 1 :** Décomposition de l'adressage CHS

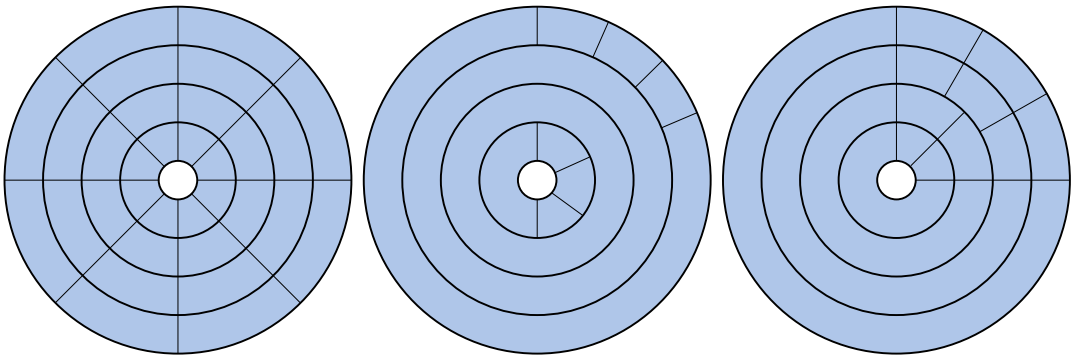
**Entrées :** $adr$ – adresse logique
$n_c$ – nombre de cylindres
$n_h$ – nombre de têtes de lecture par cylindre
$n_s$ – nombre de secteurs par tête de lecture

**Sorties :** $i_c$ – indice de cylindre
$i_h$ – indice de tête de lecture
$i_s$ – indice de secteur

1 $i_c \leftarrow adr/n_c$
2 $adr \leftarrow \text{mod}\ (adr)$
3 $i_h \leftarrow adr/n_h$
4 $adr \leftarrow \text{mod}\ (adr)$
5 $i_s \leftarrow adr/n_s$

Pour pallier ce problème, le modèle CLV donne pour taille de secteur, un nombre de bits constant, quelque soit la piste à laquelle appartient le secteur. On peut observer sur la figure 1.2b, un exemple de modèle CLV. Trois secteurs sont à la fois indiqués sur la piste extérieure et la piste intérieure. Les secteurs étant de taille égale, on peut voir que la piste intérieure contient beaucoup moins de secteurs que la piste extérieure. Contrairement au modèle CAV, ici tout l'espace est utilisé. Mais chaque piste possédant un nombre de secteurs différents des autres, la vitesse des têtes de lecture dépend de la piste ciblée et doit donc être modulée en fonction des données ciblées. Ce fait a pour conséquence d'affecter le temps d'accès aux données : la recherche de données sur les pistes extérieures est plus longue que sur les pistes intérieures, car il y a plus de secteurs à parcourir.

Le multi-zonage est un compromis entre le CLV et le CAV. Représenté sur la figure 1.2c, le modèle *Zoned Bit Recording* (National Semiconductor, 1989) va donner une taille de secteur constante à chaque zone du disque, ie. un ensemble de pistes contiguës. Cette taille de secteur est déterminée par une mesure d'angle. Le disque de la figure possède deux zones, une première constituée des deux pistes extérieures où chaque secteur a un angle de 30 degrés, et une seconde composée des deux pistes intérieures où les secteurs ont un angle de 45 degrés. Le ZBR allie donc une meilleure occupation de l'espace en minimisant la modulation de la vitesse des têtes de lecture.



(a) Modèle CAV (b) Modèle CLV (c) Modèle ZBR

FIGURE 1.2 – Modèles de zoning pour HDD

Avec l'arrivée du multi-zonage et l'augmentation du nombre de composants par disque (nombre de pistes par plateau, nombre de plateaux par cylindre), le CHS a atteint ses limites de par sa représentation dans le BIOS de la machine utilisant le disque. Effectivement, la taille maximale reconnue par le BIOS en CHS est inférieure à 8 Go, et le nombre de secteurs par piste est limité à 63. L'*Enhanced Cylinder Head Sector* (ECHS) a été créé pour fournir une géométrie faussée du disque dur, afin de respecter les normes du CHS. Le principe est d'accroître artificiellement le nombre de têtes de lecture pour réduire les nombres de cylindres et de secteurs, les faisant correspondre aux limites du CHS (1024 cylindres et 63 secteurs maximum).

L'adressage *Logical Block Addressing* (LBA) est introduit en 2002 dans les spécifications ATA-6 (T13 Technical Committee, 2002). Il répond aux nouvelles topographies de HDD, et à l'arrivée des SSDs, et a ainsi remplacé l'adressage CHS. Le LBA consiste en l'association d'un identifiant unique à chaque secteur du disque et à son utilisation pour l'accès au disque. Il s'agit de l'adressage standard selon la norme SCSI.

## 1.2 Disques basés sur la mémoire Flash – Solid State Drive (SSD)

Le disque basé sur la mémoire Flash, ou *Solid-State Drive* (SSD) (Cornwell, 2012) possède aussi une composition hiérarchique, représenté par la figure 1.3. Le premier niveau de la hiérarchie est le *chipset*, une puce électronique. Chaque *chipset* est découpé en *planes* qui sont composés de blocs. Un bloc est lui composé de la plus petite unité de stockage du SSD, la page. Celle-ci est divisée en deux parties : une partie données, et une partie *spare* utilisée pour stocker les métadonnées, le *checksum*, etc. Le disque possède également un contrôleur, gérant l'espace de stockage, et une interface hôte servant de point d'entrée des requêtes d'entrée/sortie. Le contrôleur Flash consiste en un système embarqué composé de divers modules tels celui de la correction d'erreur, l'interface des canaux Flash et le *Flash Translation Layer*, module principal, décrit un peu plus loin. Le contrôleur est relié aux *chipsets* du disque par le biais des canaux Flash.

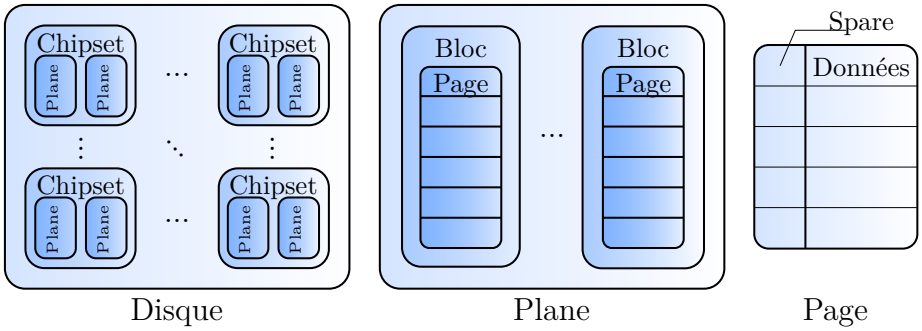



FIGURE 1.3 – Topographie du SSD

Trois opérations sont possibles sur les SDDs : la lecture et l'écriture de données, officiant sur une page et l'effacement des données, ciblant un bloc, et donc toutes les pages de ce bloc. La mise à jour des données sur une page n'est pas possible. L'écriture n'est faisable que si la page a été auparavant effacée. Ce qui signifie que la réécriture des données se fera sur une autre page du disque, la précédente sera marquée comme invalide, et nécessitera d'être effacée pour recueillir à nouveau des données.

Il existe deux types de cellules Flash, dépendant de la porte logique utilisée, les NOR et les NAND. La comparaison entre ces deux types se fera sur la densité de stockage et les performances de chaque cellule.

La taille des cellules NAND est plus petite que celle des cellules NOR d'environ 60% (Micron, 2006). Ce qui implique que la densité de stockage est plus importante sur les cellules NAND, permettant de stocker plus de données pour une puce de même taille physique.

Les performances des cellules dépendent du type de l'opération effectuée. Les cellules NOR ont des lectures sensiblement plus rapides que les NAND (62 Mo/s pour les NOR contre 41 Mo/s), mais possèdent des opérations en écriture (0.25 Mo/s contre 7.5 Mo/s) et en effacement (1 s contre 500 $\mu$s) beaucoup plus lentes. Par contre, là où les cellules NAND effectuent leurs accès aux données par page, les cellules NOR accèdent aux données par bit (Micron, 2006).

Ces caractéristiques permettent d'établir un profil d'utilisation pour les deux types de cellule. Les cellules NOR sont utilisées pour l'exécution de code machine, comme le système d'exploitation par exemple, grâce à leur meilleur débit de lecture et l'accès par bit. En revanche, les cellules NAND sont généralement utilisées pour le stockage de masse, mettant à profit des performances globalement meilleures.

Chaque cellule ne peut être effacée qu'un certain nombre de fois. Cette durée de vie de la cellule dépend non seulement du type de la cellule (NOR ou NAND), mais également du nombre de bits contenu dans la cellule. Ainsi, chaque cellule peut être *Single-Level Cell* (SLC) ou *Multi-Level Cell* (MLC) suivant si elle stocke un ou plusieurs bits d'information. Pour une même capacité, un disque contenant des cellules SLC aura donc deux ou trois fois plus de cellules qu'un disque MLC. Si une cellule possède trois bits d'information, on parle alors de cellules *Triple-Level Cell* (TLC).

Les cellules MLC présentent donc une meilleure densité de données, et un meilleur coût par bit que les SLC. Par contre, les performances en lecture/écriture des MLC sont inférieures. Le tableau 1.1 présentent des ordres de grandeur de performance pour les cellules NAND (AnandTech, 2012).

| | Lecture | Ecriture | Effacement |
|---|---|---|---|
| SLC | 25 $\mu$s | 50 $\mu$s | 75 $\mu$s |
| MLC | 200-300 $\mu$s | 600-900 $\mu$s | 900-1350 $\mu$s |
| TLC | 1.5-2 ms | 3 ms | 4.5 ms |

Table 1.1 – Temps d'exécution des opérations sur les cellules NAND

Les cellules MLC possèdent également une durée de vie moins grande. Le tableau 1.2 recense les durées de vie des cellules Flash. Dans le cas où les cellules deviennent totalement usées, elles ne peuvent plus recevoir de données.

| | NOR | NAND |
|---|---|---|
| SLC | $10^5$-$10^6$ | $10^5$ |
| MLC | $10^5$ | $10^4$ |
| TLC | - | $10^3$ |

TABLE 1.2 – Nombre d'effacements des cellules NOR et NAND

D'où la nécessité du *Flash Translation Layer* (FTL), une couche logicielle du SSD assurant le bon fonctionnement du disque. Le FTL est composé de plusieurs modules, avec chacun un rôle précis, dont la traduction logique/physique des adresses, l'égalisation d'usure, aussi appelée *Wear Leveling* (WL), et le ramassage de miettes ou *Garbage Collection* (GC).

L'égalisation d'usure est le mécanisme chargé d'uniformiser les niveaux d'usure des cellules du disque, afin d'éviter que certaines cellules soient usées beaucoup plus rapidement que les autres, et que le disque perde en capacité. De manière générale, les algorithmes d'égalisation d'usure, décrits dans la section 2.2, vont sélectionner les pages accueillant les données à écrire, en fonction de plusieurs paramètres tels le niveau d'usure du bloc ou le profil des données (souvent lues ou souvent mises à jour).

Le mécanisme de ramassage de miettes intervient lorsqu'un bloc possède en majeure partie des pages invalides ou si le disque n'a pratiquement plus ou peu de pages vierges disponibles. Un exemple de ce mécanisme est décrit dans la figure 1.4. Dans la situation initiale (a), les blocs 1 et 2 possèdent au moins la moitié de leurs pages invalides. Pour récupérer l'espace utilisable par ces pages invalides, il faut effacer le bloc, ce qui nécessite de déplacer les pages valides, contenant des données utiles, vers des pages d'un autre bloc. Dans notre exemple, le bloc 3 ne contient que des pages vierges, et peut contenir les pages valides des deux premiers blocs. La situation (b) est obtenue après la copie des pages valides vers le bloc 3. On peut voir que désormais, toutes les pages des blocs 1 et 2 sont invalides. On peut donc effectuer l'opération d'effacement sur ces blocs, sans perte de données. Le résultat final est donné par la situation (c), où les données sont toujours conservées dans le bloc 3, mais les blocs 1 et 2 contiennent désormais des pages vierges, et pourront être à nouveau utilisées pour l'écriture et le stockage de futures données.

Depuis l'année 2013, de nouveaux SSDs sont apparus sur le marché du stockage. Les SSDs à base de puces NAND 3D, développé entre autres par Samsung (Samsung, 2014) et Intel/Micron (Micron, 2015), et qui consiste en la superposition des cellules. Les disques actuels (Micron, 2016) (Samsung, 2016c) possèdent entre 32 et 48 couches de cellules entreposées. Cette technique particulière apporte un gain de performances et une augmentation significative de la capacité de la puce de l'ordre du nombre de couches la composant.

## 1.3 Interface pour les mémoires non volatiles – Non-Volatile Memory Express (NVMe)

Les premières générations de SSD ont été conçues pour supporter des bus de type SATA ou SAS. Les HDDs étant jusque là le support de stockage massivement utilisé, le bus SATA, connectant ces derniers aux systèmes informatiques, a été adopté d'office

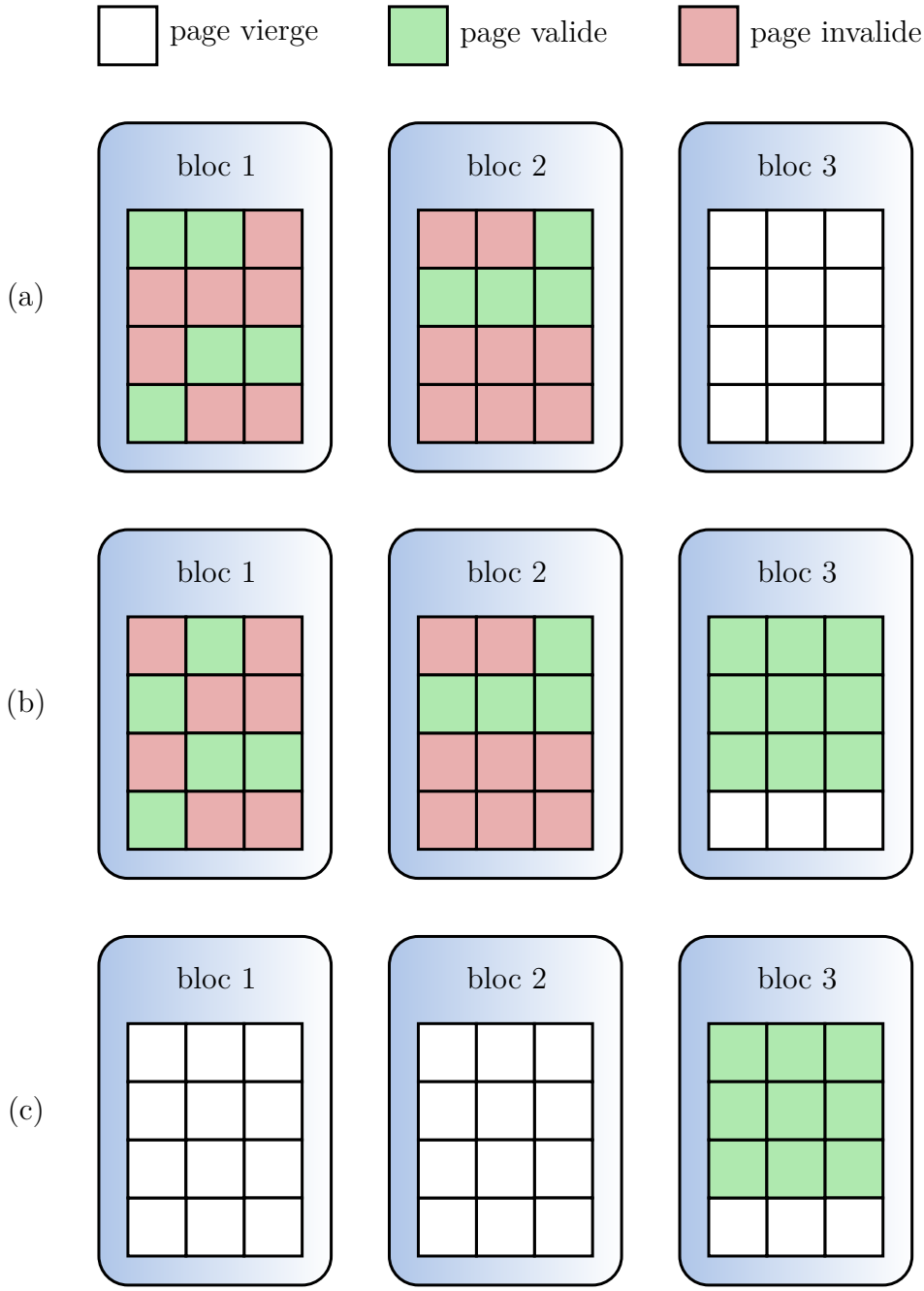



FIGURE 1.4 – Exemple de ramassage de miettes

pour connecter un SSD à un ordinateur. Mais l'interface *Advanced Host Controller Interface* (AHCI), sur laquelle repose SATA, est conçue pour l'interfaçage des disques magnétiques, et de ce fait impose des limitations aux SSDs de nouvelle génération en terme de bande-passante.

L'interface Non-Volatile Memory Express (NVMe) (NVM Express Inc., 2016) a été définie en 2011 pour répondre aux besoins propres des SSDs, en tenant compte de son architecture, et pour dépasser les performances limitées de l'AHCI. Fonctionnelle avec les bus de type PCI Express, qui dispose de bande-passantes jusqu'à cinq fois supérieures à celles de SATA 3.0, NVMe bénéficie également d'un nouveau contrôleur. En effet, le contrôleur AHCI ne possédait qu'une seule file de commandes, en comportant au maximum 32. Les SSDs peuvent exécuter des commandes en parallèle si elles

affectent des *dies* différents, et ne pouvoir utiliser qu'une seule file de commandes ne permet pas d'exploiter au mieux ce parallélisme. L'interface NVMe met à disposition un maximum théorique de $2^{16}$ files pouvant contenir jusqu'à $2^{16}$ commandes chacune. La taille de la file est également importante en raison de la vitesse élevée de traitement des requêtes d'entrée/sortie d'un SSD comparé à celle d'un HDD. De plus, la distinction faite entre les files de commandes soumises et les files de commandes accomplies facilitent l'utilisation des files mono-directionnelles.

Les derniers modèles de SSD (Samsung, 2015) (Samsung, 2016c) permettent alors d'obtenir, si utilisés avec l'interface NVMe, des bande-passantes jusqu'à 4 fois supérieures à celles obtenues avec l'interface AHCI.

# 2 Gestion permanente des disques

Avec l'utilisation, les disques perdent en performance et, pour les SSDs, en durée de vie. Différentes solutions ont été apportées pour limiter ces pertes. On peut en citer les plus importantes :

## 2.1 Gestion permanente des HDDs – Défragmentation

La défragmentation (Capps, 2002) est un mécanisme propre aux disques magnétiques ayant pour but d'améliorer les performances du disque après une utilisation massive de l'espace de stockage. Ce mécanisme est représenté sur la figure 1.5. La situation (a) montre pour 4 fichiers (bleu, rouge, jaune et vert), la répartition de leurs blocs sur le disque. Par exemple, le fichier bleu est découpé en deux parties, une première constituée des trois premiers blocs démarre au début de l'espace, et la seconde reprend après les premiers blocs du fichier rouge. Cette répartition est coutumière d'une utilisation courante, où des blocs sont écrits, puis effacés, laissant alors des espaces vides entre les données restantes.

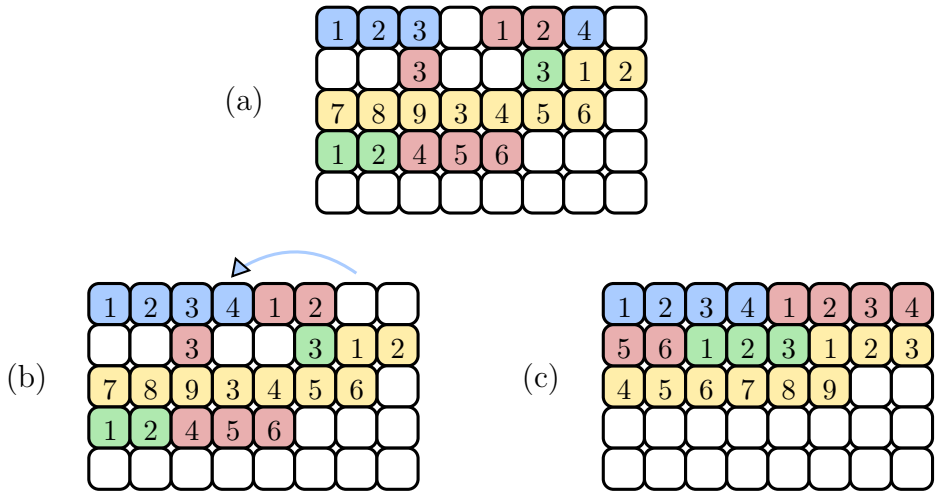



FIGURE 1.5 – Exemple d'exécution de défragmentation

L'opération de défragmentation va donc déplacer les blocs de chaque fichier pour qu'ils soient contigus, ce qui va avoir deux impacts. Le premier étant l'accès au fichier reconstitué, qui sera plus rapide. Le second étant l'optimisation de l'espace libre, qui

sera lui aussi contigu. Dans la situation (b), on peut voir la première étape de cette défragmentation, qui consiste en la reconstitution du fichier bleu : le bloc 4 du fichier est déplacé dans le bloc libre séparant le bloc 3 bleu du bloc 1 rouge. A la fin de la défragmentation, l'espace de stockage sera comme montré dans la situation (c). Tous les fichiers sont reconstitués et l'espace de stockage est en une seule trame.

Un brevet publié en 2008 (Zhang et al., 2008) présente un système de défragmentation de disque utilisant un buffer de type cache couplé au système hôte pour stocker les données ciblées par la défragmentation. Le mécanisme permet le plein accès aux données du disque durant sa défragmentation sans dégradation de performances, l'accès aux données se faisant alors à partir du buffer.

## 2.2 Gestion permanente des SSDs – Flash Translation Layer

Il a été mentionné dans la section 1.2 que les disques à base de mémoire Flash sont soumis à une espérance de vie déterminée liée au nombre fixe d'effacement par cellule. Cette usure est d'autant plus courte que leur utilisation n'est pas optimisée. A cet effet, plusieurs algorithmes d'égalisation d'usure ont été conçus pour repousser cette date de fin de vie. Le ramassage de miettes, mécanisme utilisé principalement pour la libération de pages invalides, peut aussi influer sur la durée de vie du disque, en fonction des blocs utilisés pour l'effacement.

Plusieurs algorithmes ont été proposés pour effectuer la gestion permanente des SSDs. Ci-dessous, une sélection des algorithmes les plus importants.

### 1) Égalisation d'usure

Il existe deux types d'algorithmes d'égalisation d'usure : les statiques, ou synchrones, étant appelés à intervalles réguliers et les dynamiques, ou asynchrones, appelés à chaque allocation de page.

L'égalisation d'usure statique a pour but de contrer l'effet indésirable généré par les réécritures, en effectuant à un instant donné une opération de rééquilibrage des usures des blocs sur l'ensemble composant le chip.

#### a) L'algorithme Hot-Cold Swapping

L'algorithme *Hot-Cold Swapping* (Chang and Kuo, 2005) est un exemple d'égalisation d'usure statique. A chaque période donnée, la différence d'usure entre le bloc le plus vieux (ayant reçu le plus d'effacements) et le plus jeune (ayant reçu le moins d'effacements) est calculée. Si elle dépasse un certain seuil, alors les données des deux blocs sont inter-changées. L'algorithme est basé sur le principe de données chaudes/-données froides : les données sont dites chaudes si elles sont très souvent accédées, à l'inverse des données froides le sont rarement. Si un bloc est vieux (respectivement jeune), il y a de grandes possibilités pour qu'il contienne des données chaudes (respectivement froides). Dans ce cas, déplacer les données froides sur le vieux bloc aura tendance à moins provoquer, voire stopper, les requêtes d'effacement pour concentrer ces dernières sur le jeune bloc. Ainsi, l'écart d'usure entre les deux blocs diminue.

L'efficacité de l'algorithme *Hot-Cold Swapping*, et des mécanismes synchrones de manière générale, dépend de la fréquence à laquelle il est appelé. Si elle est trop élevée, alors le mécanisme est déclenché beaucoup trop de fois, et génèrera un flux constant de requêtes servant à déplacer les données d'un bloc vers un autre. Ceci entraînera une sur-utilisation du disque. A l'inverse, si la fréquence d'appel est trop longue, alors peu de blocs seront traités, et les écarts d'usure ne se réduiront pas assez.

Les autres algorithmes d'égalisation d'usure sont à la fois statiques et dynamiques. Chaque algorithme possède donc deux mécanismes distincts, un appelé à chaque réception de requête, et un autre lancé ponctuellement.

**b) L'algorithme Static Dynamic**

L'algorithme *Static Dynamic* (M-Systems, 1999) est la fusion entre le *Hot-Cold Swapping* et une allocation basée sur un *Round-Robin*. L'allocation en *Round-Robin* signifie que tous les blocs sont placés dans une file, et qu'à chaque allocation, le premier bloc de la file est sélectionné pour accueillir les pages de données, puis il est replacé en queue de file. Cette allocation assure de cibler tous les blocs du disque les uns après les autres. Malgré une augmentation de la durée de vie du disque, l'algorithme présente les mêmes inconvénients que le *Hot-Cold Swapping*, à savoir que selon la fréquence d'appel, soit l'algorithme engendrera trop de requêtes de déplacement, soit les niveaux d'usure n'arriveront pas à s'équilibrer.

**c) L'algorithme Two-Level**

*Two-Level* (STMicroelectronics, 2006) est un algorithme partant d'un prédicat pour une allocation plus restrictive. Toute nouvelle écriture sera redirigée vers le bloc le plus jeune, soit le bloc avec le plus faible niveau d'usure. Cette première différence avec les algorithmes vus précédémment dirige dès l'allocation le disque vers l'égalisation d'usure. Le fait de choisir le bloc le plus jeune, permet de monter son niveau d'usure et donc de diminuer l'écart entre les blocs les plus vieux et ceux plus jeunes.

La partie statique de *Two-Level* utilise le même événement déclencheur que le *Hot-Cold Swapping* : l'écart d'usure entre le bloc le plus jeune et le plus vieux. Dans le cas où cet écart d'usure dépasse le seuil fixé, les pages valides du jeune bloc sont déplacées vers un bloc choisi aléatoirement. Même si le coût temporel est plus faible qu'avec le *Hot-Cold Swapping*, l'efficacité de l'algorithme dépend du choix aléatoire : si un vieux bloc est sélectionné et qu'il doit être effacé car contenant des données valides, l'écart d'usure risque d'augmenter. Dans le cas inverse, si un jeune bloc est sélectionné, alors l'écart d'usure diminuerait s'il est amené à être effacé.

**d) L'algorithme Old-Block Protection**

L'algorithme *Old-Block Protection* (Gleixner et al., 2006) possède un fonctionnement similaire au *Two-Level* en ce qui concerne l'allocation. La sélection parmi les blocs les plus jeunes en est le principe de base. La condition de déclenchement de la partie statique du *Old-Block Protection* calcule non pas l'écart d'effacements entre le plus jeune bloc et le plus vieux bloc, mais entre le plus jeune et celui ayant été effacé le plus récemment, si ce bloc est le plus vieux. Dans ce cas, les données du plus jeune bloc sont déplacées vers le bloc vierge venant d'être effacé. Cette manipulation permet de placer des données peu mises à jour sur un bloc usé, et de permettre au bloc jeune de recevoir de nouvelles données.

La condition de déclenchement de cet algorithme est ici plus forte que celle rencontrée dans les algorithmes précédents. La vérification ne se fait que lorsqu'un vieux bloc vient d'être effacé. Ceci implique que la condition sera vérifiée moins de fois que dans le cas où on calcule la différence d'effacements entre le bloc le plus jeune et le bloc le plus vieux. La partie statique de l'algorithme sera appelé également moins souvent. L'algorithme *Old-Block Protection* possède donc un surcoût temporel moins grand que l'algorithme *Two-Level*.

#### e) L'algorithme Dual Pool

L'algorithme *Dual Pool* (Chang, 2007) sépare les blocs du disques en deux ensembles de manière aléatoire : les blocs actifs et les blocs peu actifs respectivement utilisés pour stocker les données chaudes et les données froides. A chaque écriture, la différence d'usure entre le plus vieux bloc de l'ensemble actif et le plus jeune bloc de l'ensemble peu actif est calculée. Si cette différence dépasse un certain seuil, alors les données sont échangées, et les blocs changent d'ensemble.

D'autres opérations sont faites pour ajuster les ensembles dans deux cas. Le premier cas intervient si la différence d'usure entre le plus vieux et le plus jeune bloc de l'ensemble actif est trop grande. Le second cas intervient si la différence d'usure entre le plus vieux bloc de l'ensemble peu actif et le plus jeune de l'ensemble actif dépasse le seuil. Dans ces cas, le plus jeune bloc est déplacé dans l'ensemble peu actif, et le plus vieux bloc dans l'ensemble actif. Le point fort de la méthode réside dans la non sur-exposition d'un bloc à l'égalisation d'usure. En effet, si le bloc le plus jeune de l'ensemble peu actif bascule dans l'ensemble actif, il ne pourra pas être sélectionné par cette même comparaison car la recherche du plus jeune se fait parmi les non actifs. Ce qui donne à l'algorithme une variance plus faible des compteurs d'usure.

#### f) L'algorithme Rejuvenator

*Rejuvenator* (Murugan and Du, 2011) est un algorithme basé également sur la différenciation données chaudes/données froides et sur le maintien d'un écart d'usure en-dessous d'un seuil défini. Pour chaque valeur d'usure, trois listes de blocs sont construites : une pour les blocs valides (contenant uniquement des données valides), les blocs invalides (contenant au moins une page invalide) et les blocs libres (contenant uniquement des pages vierges). L'algorithme va ensuite migrer les données des blocs les plus jeunes, donc des données froides, vers les blocs libres les plus vieux si aucun bloc libre jeune n'est disponible.

Au détriment d'un stockage des niveaux d'usure et des états de l'ensemble des blocs, *Rejuvenator* garantie une réduction de la variance d'usure des blocs, et également une réduction du nombre de migrations. Celles-ci causent un moindre surcoût car elles ciblent des données peu actives.

### 2) Ramassage de miettes

Dans un SSD, l'écriture de nouvelles données ne peut s'effectuer qu'à destination de pages vierges, ayant été effacées. L'effacement n'étant possible qu'au niveau du bloc, il est important d'optimiser la sélection d'un bloc à recycler lorsque le nombre de pages vierges disponibles dans le disque diminue. Par exemple, la sélection d'un bloc contenant 80% de pages valides aura moins d'impact sur le nombre de pages libérées,

et générera plus d'opérations de copie des données que la sélection d'un bloc contenant 80% de pages invalides. Dans ce cas, l'amplification d'écriture, ou *Write Amplification* (WA), augmente.

L'amplification d'écriture est une métrique indiquant, pour une page de données, le nombre de fois où elle est réécrite sans mise à jour. Une faible amplification signifie que les données subissent peu de déplacements inutiles.

Un algorithme de ramassage de miettes est utilisé pour sélectionner et recycler les blocs selon un ensemble de conditions.

**a) L'algorithme Erase Pool**

L'*Erase Pool* (Toshiba Corporation, 1999) est un algorithme ne présentant pas d'optimisation d'égalisation d'usure : il peut être utilisé comme algorithme de pire cas pour la comparaison. La sélection des blocs à effacer se fait parmi un ensemble de blocs choisis pour être effacés, sans distinction. La variance de l'usure des blocs pour l'algorithme *Erase Pool* est par conséquent très grande.

**b) L'algorithme CAT**

La politique *Cost-Age-Time* (CAT) (Chiang and Chang, 1999) sélectionne les blocs traités pour le ramassage de miettes à l'aide d'un score. Le calcul de ce score utilise plusieurs statistiques collectées au niveau du bloc :

— le coût de recyclage – rapport entre le niveau d'utilisation du bloc (nombre de pages valides dans le bloc) et le nombre de pages invalides ;

— la date de mise à jour – date depuis la dernière modification du bloc ;

— l'âge du bloc – nombre d'effacements effectués sur le bloc.

Plus le coût de recyclage et l'âge du bloc sont élevés, plus le score l'est également. A l'inverse, plus la date de mise à jour est ancienne, plus le score est faible. Le bloc sélectionné pour l'opération de recyclage est celui possédant le score le plus faible. Ceci permet de cibler en priorité des blocs possédant une majorité de pages invalides, et dont les pages valides sont des données froides. Malgré une bonne variance du niveau d'usure des blocs, le surcoût temporel de la politique du choix du bloc est important. En effet, le calcul du score est effectué pour chaque bloc et à chaque appel de l'algorithme, ce qui pose un problème de performance quant à l'utilisation de cet algorithme.

**c) L'algorithme Turn-Based Selection**

*Turn-Based Selection* (Manning and Wookey, 2001) est un algorithme de ramassage de miettes basant l'élection du bloc sur une loi de probabilité. Dans un cas sur $x$, le bloc est ciblé aléatoirement, dans les autres cas, il est choisi selon des règles prédéfinies, comme par exemple un bloc ne contenant que des données valides. Le choix aléatoire est utilisé pour éviter la non sélection des blocs à faible utilisation ne possédant pas de pages invalides et donc de ne pas accroître l'écart d'usure entre les blocs les plus jeunes et les blocs les plus vieux.

L'efficacité de l'algorithme va dépendre ici de la valeur $x$ choisie. Si $x$ est grand, alors peu de blocs sont choisis aléatoirement, et si des blocs à faible utilisation ne possèdent aucune page invalide, alors l'écart d'usure va se creuser. Mais si $x$ est petit, le choix sera dans la majeure partie aléatoire, et l'algorithme serait similaire à *Erase Pool*, car les blocs les plus vieux ont la même probabilité d'être sélectionnés que les blocs les plus jeunes.

#### d) L'algorithme CATA

La politique *Cost-Age-Time with Age-Sort* (CATA) (Han et al., 2006) reprend le principe de CAT, soit l'utilisation d'une formule de calcul de score pour l'élection des blocs. La formule utilisée par CATA est identique à celle de CAT. La différence se passe après la sélection des blocs, où la migration des pages commence par les plus vieilles, ie. celles ayant été modifiées il y a le plus de temps.

De plus, CATA détermine le nombre de blocs à recycler en prédisant le nombre de requêtes d'entrée/sortie qui arriveraient durant l'exécution du ramassage de miettes. Si le nombre prédit de requêtes est haut, alors un bloc est sélectionné au maximum. Si au contraire il est bas, le ramassage de miettes recycle de deux à trois blocs. Cet ajout permet de limiter la surcharge de travail effectué par le disque au cas où il serait sollicité par l'utilisateur. Le surcoût temporel dû à la sélection des blocs reste lui semblable à CAT.

#### e) L'algorithme Sequential Garbage Collection

L'algorithme *Sequential Garbage Collection* (Chung and Hsueh, 2012) (SGC) repose sur l'utilisation d'un vecteur de bits où chaque bit indique si le bloc correspondant doit être effacé. Le vecteur de bits a été sélectionné pour stocker l'information car il s'agit de la structure de données possédant le plus faible coût mémoire : un bit par bloc. SGC considère qu'un bloc est sujet à l'effacement si son ratio de pages invalides dépasse 75% du nombre total de pages du bloc. La valeur du bit est calculée lors de chaque invalidation de page, et mise à jour si besoin.

A chaque appel du ramassage de miettes, le vecteur de bits est parcouru de manière séquentielle et le premier bit marqué à vrai indique le bloc sélectionné. Le vecteur démarrera son parcours suivant à partir de ce bit, pour éviter de re-vérifier l'état de blocs vus peu de temps avant. La sélection n'utilisant que le parcours d'une structure peu coûteuse en espace mémoire, prend peu de temps à s'effectuer. En plus de ce faible coût mémoire et temporel, *SGC* montre de bonnes performances en terme de variance d'usure des blocs.

### 3) Bilan sur le Flash Translation Layer

Plusieurs solutions ont été développées pour l'amélioration de la durée de vie des disques à base de mémoire Flash. L'optimisation de ces solutions est basée sur deux critères : l'équilibrage des niveaux d'usure des cellules Flash, et la diminution du surcoût temporel et d'espace mémoire relatif à l'exécution de ces algorithmes. Parmi ces solutions, les derniers algorithmes d'égalisation d'usure, *Dual Pool* (Chang, 2007) et *Rejuvenator* (Murugan and Du, 2011) et de ramassage de miettes, SGC (Chung and Hsueh, 2012) apportent les meilleurs compromis entre l'efficacité de l'algorithme et le surcoût lié à son exécution. Le tableau 1.3 résume les avantages et inconvénients des algorithmes étudiés dans cette section.

| Type | Nom | Description | Avantages et inconvénients |
|---|---|---|---|
| Egalisation d'usure | Hot-Cold Swapping (HC) | • Différence d'usure entre le bloc le plus vieux et le plus jeune<br>• Si la différence dépasse le seuil alors échange des données entre les deux blocs | • Si fréquence d'appel trop grande alors sur-utilisation du disque<br>• Si fréquence d'appel trop faible alors algorithme inefficace |
| | Static Dynamic (SC) | • Utilisation du HC<br>• Allocation des blocs en Round-Robin | • Mêmes invonvénients que le HC |
| | Two-Level (TL) | • Allocation des blocs parmi les plus jeunes<br>• Différence d'usure entre le bloc le plus jeune et le plus vieux<br>• Si la différence dépasse le seuil alors déplacement des données du plus jeune vers un bloc choisi aléatoirement | • Surcoût temporel plus faible qu'avec le HC<br>• Si le bloc choisi aléatoirement est un vieux bloc alors augmentation de la variance d'usure |
| | Old-Block Protection (OBP) | • Allocation des blocs parmi les plus jeunes<br>• Différence d'usure entre le bloc le plus jeune et celui venant d'être effacé<br>• Si la différence dépasse le seuil alors déplacement des données du plus jeune vers celui venant d'être effacé | • Les blocs les plus vieux continuent d'être mis à jour, non stabilisation de la variance d'usure |
| | Dual Pool (DP) | • Deux pools (actif et peu actif), les blocs sont aléatoirement assignés à l'un d'entre eux<br>• Différence d'usure entre le bloc le plus jeune et le plus vieux<br>• Si la différence dépasse le seuil alors les pages du plus jeune vont vers le plus vieux, celles du plus vieux vont vers des pages libres | • Meilleure gestion de l'usure, variance plus faible<br>• Surcoût faible en temps et en espace mémoire |
| | Rejuvenator (REJ) | • Identification des données actives par une grande fréquence d'écriture de ces données<br>• Maintien d'une liste de bloc, en fonction de leur degré d'usure | • Réduction de la variance d'usure<br>• Réduction de la surcharge due aux migrations des données peu actives |
| Ramassage de miettes | Erase Pool (EP) | • Sélection du bloc à effacer parmi un ensemble de blocs choisis sans distinction | • Variance d'usure très importante |
| | Cost-Age-Time (CAT) | • Effacement du bloc présentant le meilleur score basé sur les degrés d'usure et d'utilisation et du temps passé depuis le dernier effacement | • Exécution très lente due au calcul du score<br>• Si le bloc effacé est un vieux bloc, augmentation de la variance d'usure |
| | Turn-Based Selection (TB) | • Sur $n$ exécutions, $x$ ciblent un bloc aléatoire et $(n - x)$ ciblent un bloc en fonction de règles prédéfinies | • Si $x$ est grand alors moins de probabilité d'effacer des blocs utiles |
| | CAT with Age-Sort (CATA) | • Effacement du bloc le plus vieux parmi ceux présentant le meilleur score basé sur les mêmes paramètres que CAT<br>• Nombre d'effacements dépendant d'une prédiction sur le nombre futur de requêtes à traiter, plus il y aura de requêtes, moins d'effacements seront effectués | • Surcoût temporel identique à CAT<br>• Génère peu d'opérations si l'utilisateur sollicite beaucoup le disque |
| | Sequential Garbage Collection (SGC) | • Chaque bloc possède un bit indiquant si sa part de pages invalides dépasse le seuil ou non<br>• Recherche du bloc à effacer de manière circulaire sur l'ensemble des blocs du disque<br>• Effacement du premier bloc rencontré dont le bit est à vrai | • Faible surcoût temporel et en espace mémoire |

TABLE 1.3 – Bilan des algorithmes de gestion permanente pour SSD

# 3 Configurations des systèmes de stockage

Dans un système de stockage, les disques sont organisés selon une ou plusieurs configurations pour répondre aux besoins des utilisateurs en terme de performances, d'efficacité ou de robustesse. Ci-dessous, un descriptif des différentes configurations pouvant être utilisées.

## 3.1 Just a Bunch of Disks (JBOD)

Le *Just a Bunch Of Disks* (JBOD) est un ensemble de disques ne possédant pas de stratégie de placement particulière : les données sont écrites à la suite, remplissant un disque après l'autre. La figure 1.6 montre un exemple de JBOD, où les premières données (1.1 à 1.m) sont écrites sur le disque $d_1$. A son remplissage, l'écriture des données se poursuit sur le disque $d_2$, et ainsi de suite jusqu'au remplissage complet de tous les disques. Il s'agit de la configuration de base d'un système de stockage. Elle ne présente ni parallélisme de données, ni tolérance à la défaillance.

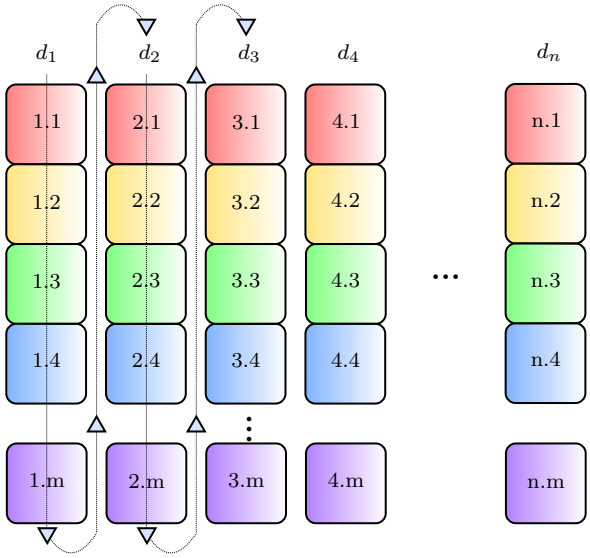



FIGURE 1.6 – Configuration JBOD

## 3.2 Redundant Array of Independent Disks (RAID)

Le *Redundant Array of Independent Disks* (Patterson et al., 1988; Chen et al., 1994) (RAID) est un ensemble de disques dont les données obéissent à un schéma de placement prédéfini. L'utilisation d'un RAID peut amener un gain de performance et/ou de fiabilité.

### a) RAID-0 – Striping

La première configuration RAID utilise le *striping* pour la répartition des données sur les disques. Le *striping* consiste en la décomposition des données en blocs (*stripe units*), et en leur répartition horizontale sur les disques. La figure 1.7 montre un exemple de RAID-0 sur $n$ disques ($d_1$ à $d_n$). Comme on peut le voir, les premiers blocs logiques de données sont respectivement répartis sur les premiers blocs physiques de chaque disque, notés 1.1, 2.1, 3.1, 4.1 jusqu'à n.1. Le processus de remplissage est alors répété pour chaque ligne de données de la configuration. Une ligne de données est le regroupement des blocs de même rang sur chaque disque physique, et est appelée *stripe*.

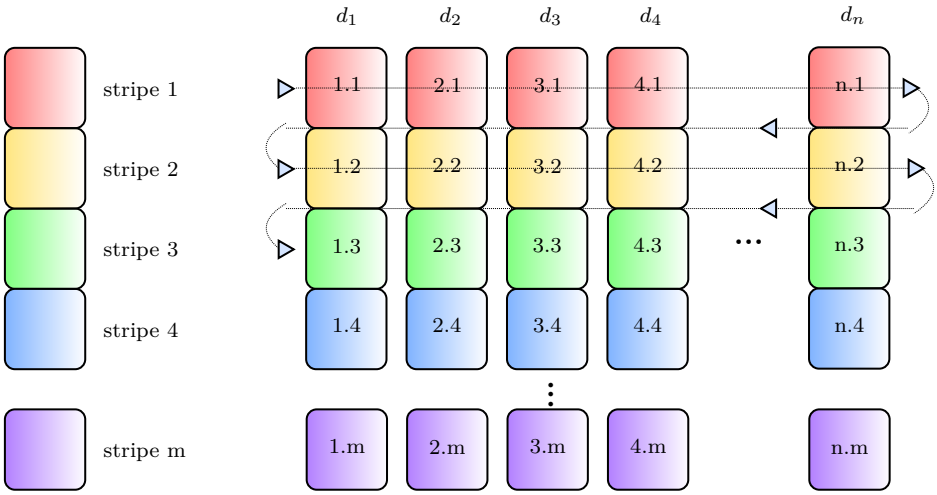



FIGURE 1.7 – Configuration RAID-0

Le RAID-0, par son remplissage horizontal, assure un équilibrage de charge sur les disques. De plus, si une requête vient à cibler une plage de blocs de données logiques successifs, alors son traitement pourra être parallélisé, puisque les blocs sont répartis physiquements sur des disques distincts. La parallélisation du traitement de la requête entraîne un temps de réponse plus court. La configuration ne présente aucun mécanisme de tolérance à la panne, ce qui lui donne un ratio de données utiles maximal, soit 100% de la capacité de stockage.

### b) RAID-1 – Mirroring

La configuration RAID-1 est basée sur le *mirroring*. Le *mirroring* est un mécanisme qui affecte à chaque disque natif, un disque miroir possédant les mêmes blocs de données. Le RAID-1 est représenté sur la figure 1.8. On peut voir que si, par exemple, le bloc 1.2 est écrit sur le disque $d_1$, alors une copie des données, notée $1.2'$, sera écrite sur son disque miroir ($d_2$).

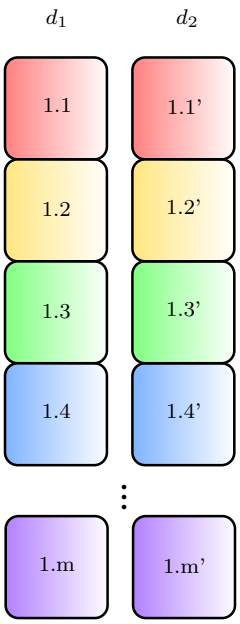



FIGURE 1.8 – Configuration RAID-1

Le *mirroring* est un mécanisme de tolérance à la panne implémentant une redondance intégrale des données. Il implique qu'en cas de défaillance d'un disque, les données se trouvant sur ce disque ne sont pas perdues. En contre-partie de cette robustesse, la part de données utiles n'est plus maximale, elle est à 50% du système global.

Une large requête pourra aussi être parallélisée sur un RAID-1 s'il s'agit d'une lecture, en accédant au disque natif ou au disque miroir pour chaque bloc de données. Les écritures ne peuvent pas être parallélisées car les données doivent être mises à jour sur les deux disques.

Il peut aussi être fait mention de RAID-1 *multi mirroring* (SNIA, 2009), en opposition au *simple mirroring* décrit juste au-dessus, où chaque disque natif de données possèderait deux disques miroir. Dans ce type de configuration, la part de données utiles passe à 33%.

### c) RAID-01 – Striping et mirroring

Le RAID-01 est une configuration mélangeant les principes des deux configurations précédentes, à savoir le *striping* et le *mirroring*. Les disques sont séparés en deux groupes : le groupe natif et le groupe miroir. Les données sont alors écrites sur le groupe de disques natif, dans le sens horizontal, et une copie est effectuée sur le groupe de disques miroir. La configuration est décrite sur la figure 1.9, où les disques 1 à $k$ représentent le groupe natif, et les disques $(k+1)$ à $n$ le groupe miroir, avec $k = n/2$.

L'utilisation des deux mécanismes permet de faire un compromis entre la performance et la robustesse. Ainsi, le système permet de paralléliser les requêtes de lecture et d'écriture, et est tolérant aux pannes. Il peut survivre à plusieurs défaillances si elles n'impactent pas le même couple de disques natif-miroir.

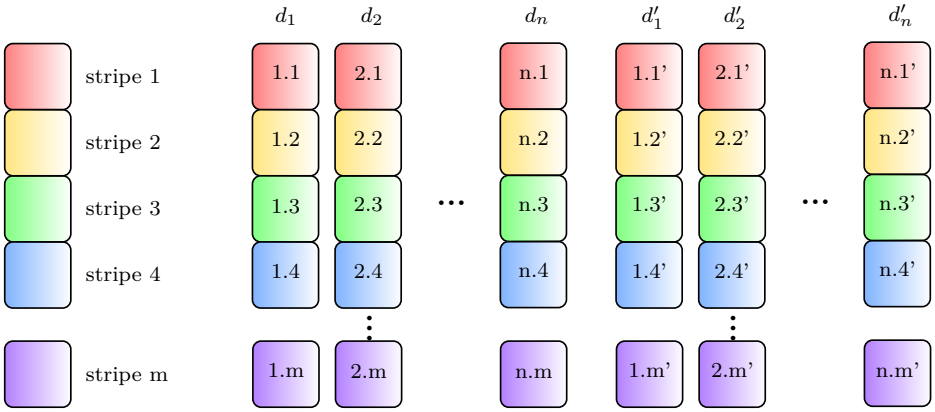



FIGURE 1.9 – Configuration RAID-01

### d) RAID-4 – Parité

La parité est un mécanisme implémentant une redondance partielle. Elle permet la tolérance à la panne avec reconstruction des données perdues à l'aide des données saines, en utilisant l'opération XOR. Soient $b_i$ les blocs de données d'un *stripe* et $p$ le bloc de parité associé à ce *stripe*, le bloc $p$ est construit grâce à l'opération suivante :

$$b_1 \oplus b_2 \oplus b_3 \oplus ... \oplus b_{n-1} = p \tag{1.1}$$

En supposant la configuration décrite sur la figure 1.10, on peut donc dire, pour le premier stripe, que :

$$1.1 \oplus 2.1 \oplus 3.1 \oplus 4.1 = p.1 \tag{1.2}$$

Soit une défaillance ciblant le disque $d_2$, l'opération XOR étant commutative et associative, on peut donc reconstruire le bloc 2.1 grâce à l'opération suivante :

$$1.1 \oplus 3.1 \oplus 4.1 \oplus p.1 = 2.1 \tag{1.3}$$

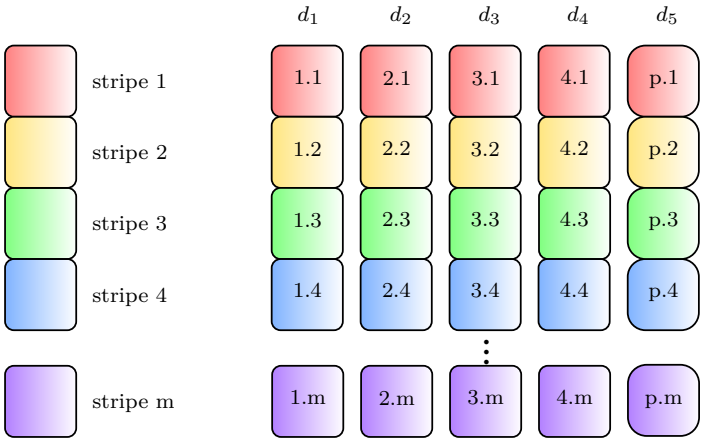



FIGURE 1.10 – Configuration RAID-4

Le placement des blocs de données sur le RAID-4 se fait horizontalement, *stripe* après *stripe*, ce qui permet l'utilisation du parallélisme des accès. La configuration est tolérante à une panne, et optimise la part de données utiles. Selon $n$ le nombre de disques de la configuration, cette part est égale à $1 - (1/n)$.

On définit trois types de requêtes, en fonction de la taille qu'occupent les données dans le *stripe*, représentés figure 1.11 : le *small stripe* est un accès ciblant moins de la moitié des *stripe units*, le *large stripe* cible au moins la moitié des *stripe units* mais pas la totalité et le *full stripe* la totalité du *stripe*.

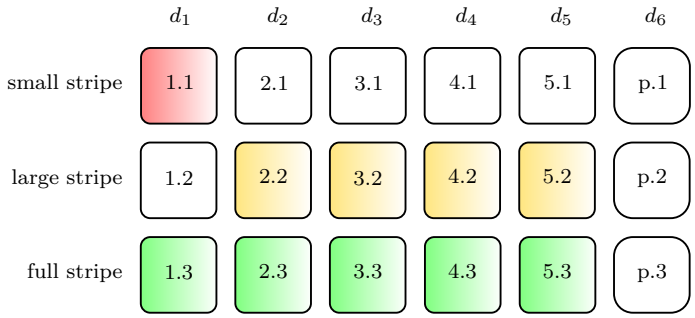



FIGURE 1.11 – Définitions de small, large et full stripe

Dans le cas où une requête d'écriture cible un RAID-4, il y a également besoin de mettre à jour le bloc de parité. L'exécution d'une écriture *full stripe* consiste en l'écriture directe de chaque *stripe unit*, de donnée comme de parité. Si l'écriture est partielle, la mise à jour de la parité nécessite une pré-lecture des *stripe units* de données et éventuellement de parité situés sur les disques. Cette mise à jour peut se faire des deux manières suivantes, en supposant l'écriture *small stripe* de la figure 1.11 :

$$1.1' \oplus 1.1 \oplus p.1 = p.1'$$
$$1.1' \oplus 2.1 \oplus 3.1 \oplus 4.1 \oplus 5.1 = p.1'$$

Dans le premier cas, la mise à jour nécessite 2 pré-lectures, et dans le second, 4 pré-lectures. Pour optimiser la rapidité d'exécution des requêtes, la méthode choisie sera celle engendrant le moins de pré-lectures. Ainsi, pour une requête en *small stripe*, les pré-lectures cibleront les *stripe units* de données mis à jour, ainsi que celui de parité. Pour une requête en *large stripe*, les deux calculs suivants sont possibles :

$$2.2' \oplus 3.2' \oplus 4.2' \oplus 5.2' \oplus 2.2 \oplus 3.2 \oplus 4.2 \oplus 5.2 \oplus p.2 = p.2'$$
$$2.2' \oplus 3.2' \oplus 4.2' \oplus 5.2' \oplus 1.1 = p.2'$$

Le premier calcul nécessite 5 pré-lectures et le second 1 seule. De même, pour la rapidité de l'exécution des requêtes, les pré-lectures cibleront les *stripe units* de données non mis à jour.

Pour un RAID-4 recevant des requêtes d'écriture *small stripe*, les requêtes ne cibleront pas toujours les mêmes disques de données (dépendant de l'adresse logique). Par contre, pour chaque requête, une mise à jour de la parité est nécessaire, et une requête sera toujours à destination du disque de parité. Ce qui génère un déséquilibrage de charges et le disque de parité devient un goulot d'étranglement.

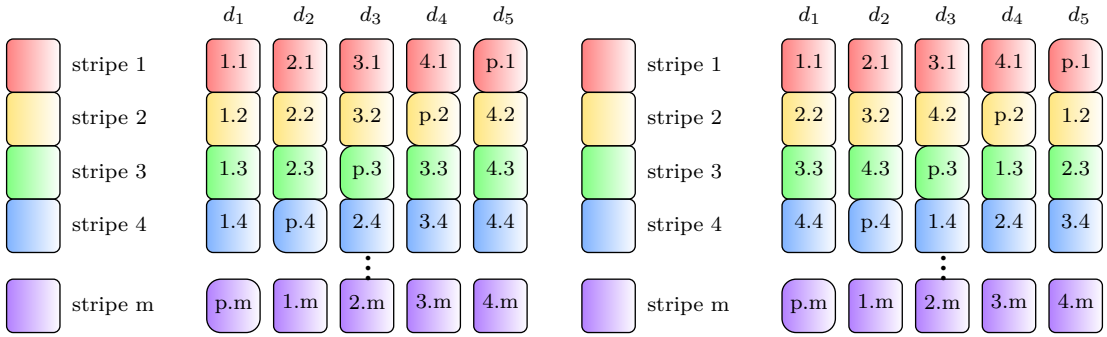
### e) RAID-5 – Parité et declustering

Le RAID-5 est similaire au RAID-4, à l'exception que les *stripe units* de parité sont répartis sur les disques, et de même pour les requêtes les ciblant : les charges sont

donc équilibrées. Le RAID-5 est agrémenté d'un algorithme de *declustering* (Holand and Gibson, 1992) pour la localisation des blocs de parité et/ou de données dans la configuration. Il existe deux types de *declustering* :

— *declustering* de parité : les blocs de parité sont répartis de manière circulaire sur tous les disques, les blocs de données d'un *stripe* sont répartis de la même manière que sur un RAID-0, emplacement du bloc de parité exclu (figure 1.12a) ;

— *declustering* de données : en plus de la répartition des blocs de parité, les blocs de données sont eux aussi répartis de manière circulaire (figure 1.12b).




(a) Declustering de parité (b) Declustering de données

FIGURE 1.12 – Configuration RAID-5 avec declustering

Le *declustering* offre un compromis entre la facilité de la traduction logique/physique des blocs de données et l'équilibrage de charges.

#### f) RAID-NP – Codes d'erreur multiples

Le RAID-NP est la configuration générique du RAID-5, où N est le nombre de disques de données et P le nombre de disques de code d'erreur. Le nombre de disques de code d'erreur influe directement sur la tolérance aux défaillances de la configuration. L'augmentation du nombre de disques de données améliore la part de données utiles dans la configuration. Cette part de données utiles est de : $1 - (P/(N + P))$. Par opposition, l'augmentation du nombre de disques de redondance partielle (code d'erreur) augmente le nombre de défaillances tolérées.

### 3.3 Synthèse

La table 1.4 synthétise les particularités de chaque configuration. Le pourcentage d'utilisation donne la part de données utiles relativement à celles des données stockées, la robustesse le nombre de défaillances tolérées et la performance la présence de parallélisme. On note $n$ le nombre de disques de système, $N$ le nombre de disques de données et $P$ le nombre de disques de codes d'erreur.

## 4 Simulation des systèmes de stockage

Afin d'analyser les performances d'une technologie de disque, d'un algorithme pour sa gestion permanente, de la configuration adoptée ou de manière plus globale, d'un

| Nom | % d'utilisation | Robustesse | Performance |
|---|---|---|---|
| JBOD | 1 | 0 | aucune |
| RAID-0 | 1 | 0 | lecture/écriture |
| RAID-1 | $1/2$ | $n/2$ | lecture |
| RAID-01 | $1/2$ | $n/2$ | lecture/écriture |
| RAID-5 | $1-(1/n)$ | 1 | lecture/écriture |
| RAID-NP | $1-(P/(N+P))$ | $P$ | lecture/écriture |

TABLE 1.4 – Synthèse des configurations

système de stockage complet, la simulation est sans aucun doute le processus à adopter présentant le coût le plus faible. Nous allons dresser dans cette section une liste des simulateurs existants dans ce domaine en précisant leurs choix de conception et leurs apports.

### a) Disksim

La simulation des systèmes de stockage débute dans les années 90 avec DiskSim (Ganger et al., 1998). DiskSim a été créé dans le but de déterminer le profil d'exécution d'une requête au sein de chaque composant du système de stockage : calcul du temps de réponse de la requête, étude de l'impact des caractéristiques du flux de requêtes sur le temps de réponse. Il a également pour objectif l'analyse de différents paramètres du système simulé, tels des algorithmes d'ordonnancement de requête.

DiskSim permet la simulation de systèmes à faible échelle (du simple disque au RAID) composés de disques magnétiques. Chaque système simulé avec DiskSim utilise un pilote, et un maximum de deux contrôleurs. Les contrôleurs et les disques sont reliés par des bus connectant au plus quinze composants. D'après la documentation de la dernière version de DiskSim (Bucy et al., 2008), un contrôleur peut posséder quatre bus de sorties. Ainsi, chaque contrôleur pourrait avoir la charge d'une soixantaine de disques. Il est néanmoins indiqué que les contrôleurs possédant plus d'un bus en sortie n'ont pas été suffisamment testés. Disksim assure donc la simulation d'un système composé d'une quinzaine de disques.

Chaque composant du système simulé, à savoir les bus, les disques ou encore les pilotes, est géré par un module distinct de DiskSim. Écrit en C, et étant *open source*, DiskSim peut être intégré à un simulateur de système plus complexe, en l'utilisant comme une bibliothèque. Ceci permet d'associer la simulation du système de stockage à celle d'un processeur, d'une carte graphique, etc. pour étudier leurs interactions.

DiskSim intègre également deux types d'entrée de traces : des traces présentes dans des fichiers, qui peuvent représenter les requêtes d'entrée/sortie d'applications réelles et un générateur de traces synthétiques. La simulation du système débute avec le pilote d'entrée/sortie, qui envoie les requêtes issues de la trace au contrôleur de disque par le biais du bus. Le contrôleur détermine si les requêtes ciblées sont présentes dans la mémoire cache du disque ou non, et les récupère.

La dernière version de DiskSim (Bucy et al., 2008) ajoute un ensemble de nouvelles fonctionnalités, ainsi que des améliorations et des corrections de bugs. Entre autres, il

gère désormais les systèmes microélectromécaniques (MEMS) et inclut un outil d'extraction de caractéristiques de disques appelé Dixtrac (Schindler and Ganger, 1999).

#### b) Extension SSD pour Disksim

Afin de pouvoir simuler les disques basés sur la mémoire Flash, une première extension de DiskSim a été développée par Microsoft (Agrawal et al., 2008) pour les cellules NAND. Cette extension apporte, au moment où les mémoires Flash s'installaient sur le marché, l'analyse des performances de ces disques disposant d'une nouvelle technologie considérant différents types de *workload*, et d'étudier le comportement des composants du disque en conséquence.

L'extension consiste en une dérivation du module de disque rotatif implémenté nativement dans DiskSim, pour représenter le SSD. Le module est agrémenté d'un niveau de parallélisme afin de gérer les *dies* du disque, avec une file de requêtes chacun. Des structures de données supplémentaires sont également ajoutées afin de stocker le positionnement des blocs logiques, l'état d'utilisation du bloc et le niveau d'usure. Ces derniers ajouts permettent d'observer l'efficacité de mécanismes d'égalisation d'usure et de ramassage de miettes.

#### c) FlashSim

FlashSim (Kim et al., 2009) est aussi une extension de DiskSim, développée pour la gestion des SSDs à base de cellules NAND. L'extension est écrite en C++, pour un schéma conceptuel simple, orienté-objet. Chaque composant du disque est représenté par une classe. Les composants intégrés sont la RAM, le contrôleur avec la gestion des bus internes du disques, le *package* et un dernier module pour les algorithmes du FTL. Le *package* est divisé selon la topologie des SSDs, soit en : *die*, *plane*, bloc et page. L'ensemble des composants font parties d'une classe mère : le SSD.

Le flot d'exécution est semblable à celui de DiskSim : les requêtes sont envoyées du pilote vers le contrôleur SSD, qui à la charge de leurs traitements. Pour déterminer la position physique des données et la décomposition de la requête, le contrôleur utilise le module FTL afin de fournir une liste d'événements de lectures, écritures ou effacements. Ces événements sont ensuite redirigés vers les *packages*.

Comme avec l'extension SSD faite par Microsoft (Agrawal et al., 2008), des statistiques liées à l'égalisation d'usure sont stockées au niveau du SSD, du *package*, du *die*, du *plane* et du bloc. Elles ne sont mises à jour que lors d'événements d'effacement, pour minimiser le surcoût temporel.

Les deux extensions de DiskSim (Agrawal et al., 2008; Kim et al., 2009), bien qu'intégrant un modèle de SSD, ne permettent pas le support de systèmes hétérogènes, mélant aussi bien disques magnétiques et disques basées sur la mémoire Flash. Seule l'une ou l'autre des technologies est supportée pour une simulation.

#### d) SSDSim

SSDSim (Hu et al., 2011) est un simulateur de disques à base de cellules NAND créé pour aider à concevoir des SSDs exploitant au maximum les aspects haute performance

et endurance. Contrairement à DiskSim (Bucy et al., 2008), le système de stockage est ici centré sur un disque seul.

Ecrit en C, SSDSim est un programme divisé en trois niveaux. Le premier niveau, appelé niveau supérieur, a la charge du buffer du disque, et de l'ordonnancement des requêtes. C'est par ce niveau qu'arriveront les requêtes au niveau du disque. Le second niveau est composé du module d'allocation des pages et du FTL. Le dernier niveau, inférieur, gère les composants matériels, ie. les *packages*.

SSDSim permet d'analyser le comportement du moindre composant logiciel ou matériel du SSD. Ainsi, il est possible de paramétrer la topologie du disque ou les algorithmes embarqués pour l'allocation, la gestion des buffers, l'égalisation d'usure ou encore l'ordonnancement des requêtes. De plus, il supporte les commandes avancées propres au SSD :

— le *copyback* – copie des données d'une page vers une autre ;
— le *multiplane* – application d'une même requête sur des *planes* différents ;
— l'*interleave* – exécution simultanée d'une requête sur plusieurs *dies*.

Le simulateur n'intégrant pas de générateur de traces synthétiques, il nécessite d'avoir en entrée, en plus du fichier de paramétrage du disque, un fichier de trace. Il fournit diverses métriques de performances telles les temps d'attente, de service ou de réponse des requêtes, ainsi que le nombre d'effacements effectués ou encore le nombre de *hits* sur le buffer du disque. Le simulateur inclut également le calcul du temps d'exécution des algorithmes internes du SSD pour délivrer des résultats précis.

SSDSim simule de manière précise le comportement d'un SSD, mais est inutilisable pour des systèmes à configurations et nombres de disques variés.

#### e) NANDFlashSim

Le simulateur NANDFlashSim (Jung et al., 2012) est, comme SSDSim (Hu et al., 2011), développé pour analyser le comportement d'un seul disque, composé lui aussi de cellules NAND. NANDFlashSim se focalise sur l'architecture matérielle, où chaque composant de la topologie du disque, du *die* au bloc, est représenté par un module. L'objectif est de capturer le plus fidèlement possible l'état de chaque composant durant la simulation, afin de déterminer le paramétrage du disque menant à des performances optimales.

NANDFlashSim a le même flot d'exécution que SSDSim : le module gérant le bus interne du disque récupère les requêtes d'un module hôte, et traduit les requêtes en commandes à l'aide du contrôleur. Là où FlashSim (Kim et al., 2009) et SSDSim utilisaient les événements comme opération, pour la lecture, l'écriture et l'effacement, NANDFlashSim décompose chaque requête en chaîne de commandes représentant des transactions mémoires, issues de la norme *Open NAND Flash Interface* (ONFI) (ONFI, 2014).

Bien qu'apportant des précisions sur le comportement du SSD, tout comme SSDSim, NANDFlashSim possède le même inconvénient. Le simulateur ne permet pas le support de systèmes de stockage hétérogènes, composés de plusieurs unités de stockage.

#### f) VSSim

VSSim (Yoo et al., 2013) est un simulateur ayant la particularité d'émuler le comportement du SSD à l'aide d'une machine virtuelle. La motivation ayant amené au développement de VSSim est d'observer le comportement de l'hôte face à des prototypes de SSD : nouvelles topologies, nouveaux algorithmes de gestion permanente, etc.

Les machines virtuelles utilisées sont basées sur QEMU (Bellard, 2005), un émulateur de processeur. Il est utilisé pour exécuter des systèmes d'exploitation (Linux, Windows, etc.), et procure des interfaces IDE pour l'émulation de disques. VSSim intègre un modèle SSD pour QEMU, qui se lie à la machine virtuelle à l'aide d'une interface IDE.

Le schéma conceptuel de VSSim est constitué de quatre modules : le module FTL, l'émulateur d'entrées/sorties, le générateur de délai et le moniteur SSD. Le module FTL est semblable à ceux utilisés dans FlashSim (Kim et al., 2009) ou SSDSim (Hu et al., 2011), à savoir exécuter les algorithmes de traduction des adresses logiques/physiques, de ramassage de miettes et d'égalisation d'usure. L'émulateur d'entrées/sorties récupère les opérations NAND à effectuer de la part du FTL, et calcule le délai correspondant à l'opération. Le générateur de délai introduit le délai respectif à l'exécution de l'opération NAND, en prenant en considération le surcout de VSSim dû au traitement de la requête. Le moniteur SSD génère des informations en temps réel sur le traitement des requêtes, telles le nombre de requêtes en lecture et écriture, le nombre d'exécution de ramassage de miettes effectuées ou encore le nombre de commandes TRIM traitées.

VSSim est lui aussi spécialisé dans l'étude du comportement d'un disque seul, considérant un flux de requêtes. Son utilisation pour la simulation de systèmes de stockage à large échelle n'est donc pas possible.

De manière générale, on distingue deux types de simulateurs : ceux déterminant les performances d'un système de stockage (DiskSim et ses extensions), et ceux analysant le comportement d'un disque seul (NANDFlashSim, SSDSim et VSSim). Même si les systèmes supportés par DiskSim peuvent atteindre théoriquement 120 disques, nombre de disques largement inférieur à celui des systèmes à large échelle, l'absence d'hétérogénéité de systèmes est un frein à la simulation de systèmes actuels, utilisant pour certains les deux technologies de disques.

# 5 Conclusion

A partir de l'étude faite de l'évolution des différents aspects du stockage, on se rend compte que l'utilisateur consomme toujours plus de données, en un temps toujours plus court. De plus, si différentes pannes venaient à surgir, le système doit toujours répondre aux besoins de l'utilisateur, et continuer de fournir l'accès aux données stockées. La recherche de la robustesse d'un système de stockage date déjà de plus d'une vingtaine d'années, et essaye d'allier parallélisme des accès aux données, avec le *striping*, et tolérance à la défaillance, avec le *mirroring* et l'utilisation de codes d'erreur.

L'analyse des deux grandes technologies de disque, HDD et SSD, nous montre que les disques ne cessent de croître en capacité et en complexité matérielle et logicielle. De plus, un compromis est sans cesse réalisé entre les performances que possède le support de stockage et son côut. Pour les disques à base de mémoire Flash, une dimension supplémentaire est à prendre en considération : la durée de vie des cellules du disque.

Enfin, l'ensemble des simulateurs existants dans la littérature ne permettent pas l'analyse du comportement des systèmes hétérogènes à large échelle. Les systèmes utilisés dans les centres de données actuels utilisent des technologies de disque différentes, et un nombre de disques de l'ordre du millier. Cette étude nous mène à la conclusion que la simulation de tels systèmes nécessite le développement d'un nouvel outil tenant compte de leurs spécificités.

# Chapitre 2

# OGSSim : motivation et schéma conceptuel



*OGSSim, pour Open and Generic Storage system Simulation tool, est un outil de simulation générique et ouvert pour systèmes de stockage de données à large échelle. Il est constitué de plusieurs modules, tous personnalisables, offrant la possibilité de spécifier pour chaque composant de la simulation un comportement précis.*

*Nous l'avons développé dans l'optique d'être ouvert et générique. A cet effet, OGSSim permet l'ajout simple de nouveaux modules implémentant des technologies de stockage ou des stratégies de placement. L'implémentation se fait à l'aide d'une interface minimisée pour faciliter l'usage à quelques fonctions pour la bonne intégration de l'ensemble des modules déjà existants. Par exemple, l'ajout d'une nouvelle technologie, pouvant représenter n'importe quel type de support de stockage, s'effectue par la définition d'un nouveau pilote de support de stockage, et d'un modèle de calcul pour ce support. Une nouvelle stratégie de placement peut être définie au sein du pilote de volume, dans la chaîne de décomposition des requêtes.*

*Ce chapitre présente l'outil de simulation, en commençant par expliquer les besoins et répondre aux diverses questions :*

- *Quels sont les systèmes que nous souhaitons simuler ?*
- *Les outils existants peuvent-ils prendre en charge ces types de système ?*

*Ces questions guideront notre proposition. Nous terminerons par l'étude du schéma conceptuel général d'OGSSim, en présentant les bibliothèques utilisées et les fichiers nécessaires à sa bonne exécution.*

# 1 Motivation

Notre objectif est de pouvoir simuler des systèmes de stockage de données à large, voire très large, échelle afin d'en évaluer les performances. La taille des systèmes ne cessant de croître, et leur coût avec, le processus de simulation permettra de prédire et de comparer la performance, la robustesse et la rentabilité de ces systèmes. Par exemple, le centre de données de l'Utah du NSA (Wikipedia, 2016) serait capable de stocker entre 3 et 12 exaoctets de données, selon un article de Forbes paru en 2013 (Hill, 2013). Les simulateurs présentés dans le chapitre 1 ne nous permettent pas d'opérer la simulation de tels systèmes et ceci pour les raisons suivantes :

1. Les systèmes pris en compte ne peuvent conjuguer plusieurs technologies. En exemple Disksim (Ganger et al., 1998) et ses extensions (Agrawal et al., 2008; Kim et al., 2009) qui simulent le comportement soit d'un ensemble de HDD, soit de SSD, mais pas des deux en même temps. Or, la cohabitation est inévitable car les systèmes actuels et futurs sont déjà hétérogènes.
2. Ensuite, aucun des simulateurs présents dans la littérature ne peut gérer des systèmes de plus de quelques dizaines de disques. Généralement, ils vont se limiter à la simulation précise d'un seul disque (Hu et al., 2011), ou alors d'une configuration de type RAID d'au plus 15 disques (Bucy et al., 2008).
3. Enfin, la dernière raison est la performance de l'outil de simulation. Nous devons soumettre à un système composé de centaines voire milliers d'unités de stockage, des traces pouvant contenir de l'ordre du million de requêtes. Des simulations de cet ordre de grandeur génèrent une grande quantité de calcul, que ce soit pour la redirection et la décomposition des requêtes ou le calcul des métriques liés à leur exécution. L'outil doit donc être capable de délivrer des résultats très rapidement.

Les simulateurs existants ne répondant pas à nos besoins, nous avons donc décidé de concevoir un nouvel outil de simulation performant, pouvant prendre en charge des systèmes larges et hétérogènes. Nous avons nommé cet outil OGSSim pour **O**pen and **G**eneric **S**torage system **Sim**ulation tool, avec comme principales qualités l'ouverture et la généricité :

- ouverture : l'aspect modulaire facilite l'ajout de nouvelles fonctionnalités, stratégies de placement, algorithmes de gestion permanente ou modèles de calcul en respectant les spécifications d'une interface minimisée ;
- généricité : l'utilisation de ces fonctionnalités peut se conjuguer avec n'importe quel autre module déjà existant, permettant ainsi la simulation de toutes combinaisons de schémas de placement, de technologies de disque, etc.

# 2 Schéma conceptuel général d'OGSSim

La première étape de la conception d'OGSSim est l'élaboration d'une architecture logicielle offrant le support nécessaire à son développement.

## 2.1 Informations techniques

Afin d'atteindre ces objectifs d'ouverture, de généricité et de performance, OGSSim a été conçu dès le départ comme un outil modulaire pour fonctionner dans un environnement *multi-threadé*. Il est développé en C++ pour l'efficacité du langage, tant

au niveau de l'écriture que des performances d'exécution, et son accès simple aux appels systèmes. La norme utilisée est C++11 car elle apporte plusieurs fonctionnalités utiles telles que le support des *threads* dans la bibliothèque standard et les pointeurs intelligents.

La figure 2.1 montre les modules principaux d'OGSSim ainsi que leurs liens. Chaque lien est représenté par un canal de communication décrit dans la section 2.2. En entrée du logiciel, le fichier de configuration de ce dernier, écrit en XML, possédant les chemins d'accès aux fichiers contenant les paramètres de la simulation (trace et architecture). En sortie, différents fichiers de résultats et graphes de visualisation.

Le logiciel est décomposé en plusieurs modules, chacun remplissant un rôle spécifique. Un module est associé soit à un processus de la simulation (extraction des fichiers d'entrée, calcul des métriques de performance, création des fichiers de sortie), soit à un composant du système de stockage (disque, configuration). Chaque module peut lui-même se composer de sous-modules afin de personnaliser au mieux le comportement du système simulé. Par exemple, au niveau du module d'exécution, il est possible de choisir entre deux sous-modules représentant le modèle de calcul de transfert : le premier étant un modèle naïf où toutes les données sont transférées immédiatement sans tenir compte des délais d'attente de bus et le second est un modèle de files d'attente prenant en considérant la congestion des données au sein des bus.

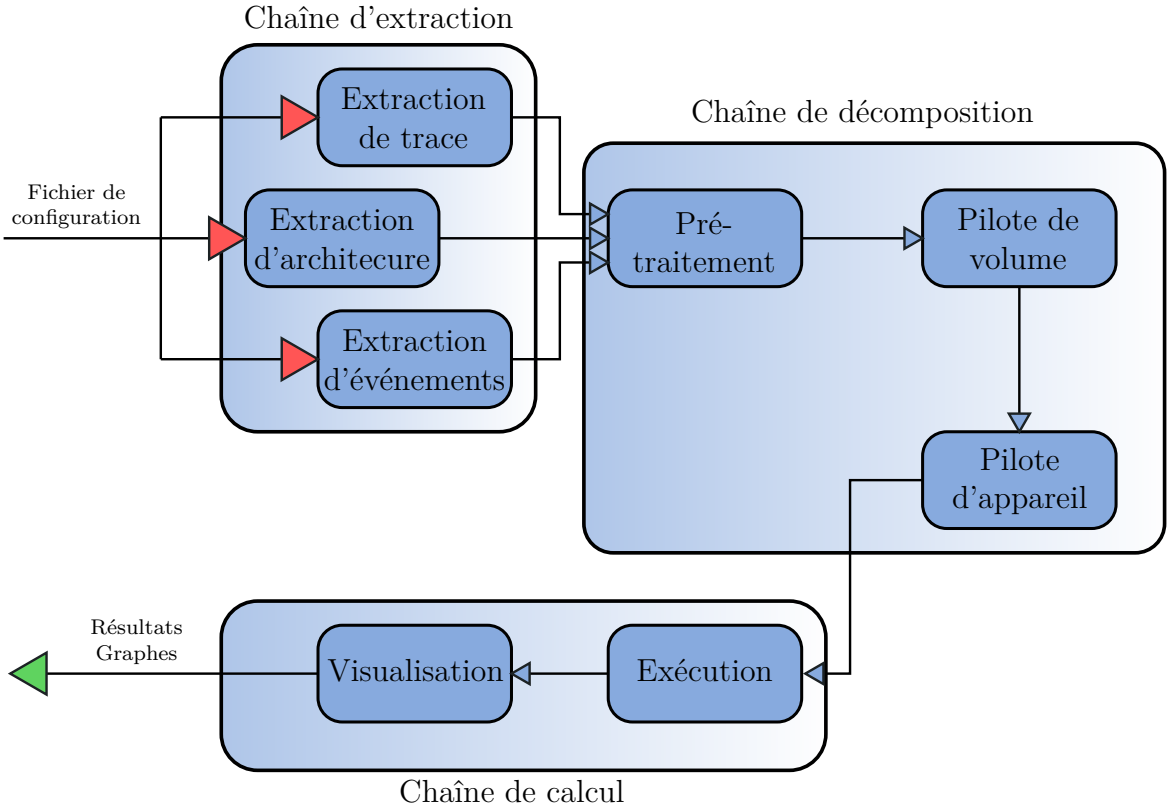



FIGURE 2.1 – Modèle général d'OGSSim

La version actuelle d'OGSSim comporte 17'000 lignes de code pour 112 fichiers. La répartition des lignées de code par module est spécifiée sur la figure 2.2. On peut observer que le tiers des lignes de code sont utilisées pour le module du pilote de volume,

où tous les algorithmes de placement sont décrits. Les autres modules importants du simulateur sont celui des utilitaires (mécanismes d'extraction/sérialisation XML, de synchronisation, de tests unitaires, etc.), des structures de données et d'exécution (description des modèles de calcul).

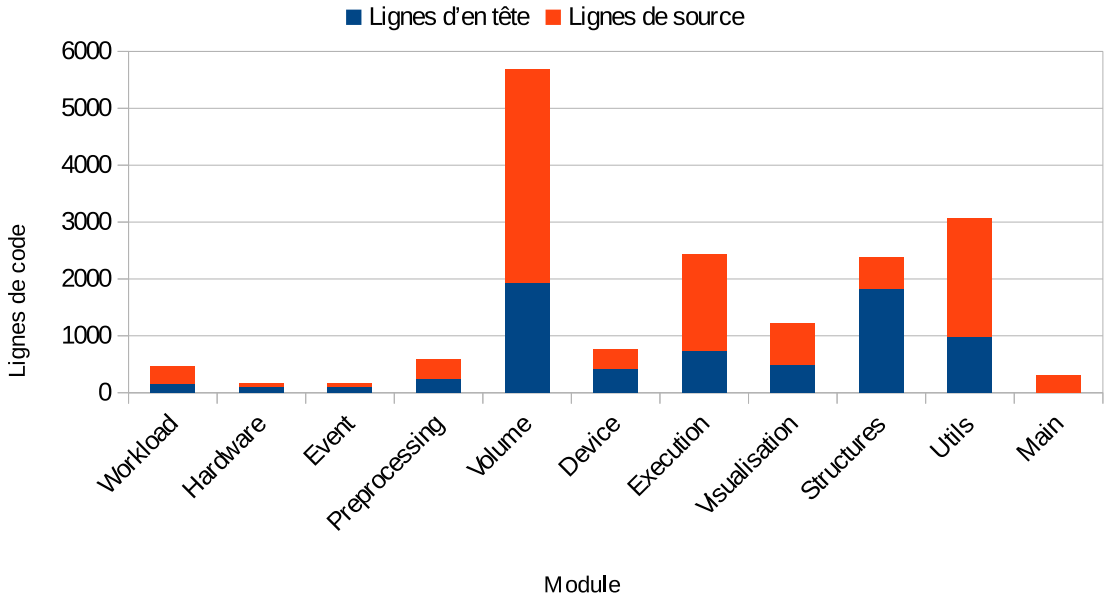



FIGURE 2.2 – Répartition des lignes de code dans OGSSim

## 2.2 Bibliothèques utilisées

Plusieurs produits professionnels mis à la disposition des développeurs ont été intégrés à OGSSim. Les bibliothèques intégrées sont séparées en deux catégories : celles utilisées en dehors de l'exécution d'OGSSim, pour la compilation et la documentation, et celles utilisées durant la simulation.

### a) CMake – outil de compilation portable

L'outil CMake (Kitware Inc, 2011) facilite la compilation d'OGSSim. A l'aide de la commande `cmake`, le script de compilation est généré en suivant les modalités renseignés dans un fichier de description. Un exemple de ce type de fichier est donné par le code 2.1. On peut donc décrire le paramétrage de la compilation, comme l'emplacement des fichiers d'en-tête (lig. 13), des fichiers sources (lig. 19) mais également les bibliothèques à inclure dans le projet (lig. 16). Il est aussi possible de définir plusieurs profils de compilation, comme ici les profils *debug* (lig. 23) et *release* (lig. 22), en activant ou non les options de débogage et d'optimisation.

L'utilisation de CMake automatise la génération du script de compilation en tenant compte de l'architecture physique de la machine et du système d'exploitation utilisés.

### b) Doxygen – documentation

Doxygen (van Heesch, 2016) est l'outil standard de documentation de projet C++, mais fonctionne aussi pour un grand ensemble de langages (C, python, Java, etc.). A

```
# CMake Header
cmake_minimum_required (VERSION 2.8)
project (OGSSim)

# Version Number
set (OGSSim_VERSION_MAJOR 1)
set (OGSSim_VERSION_MINOR 2)

# Modules
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} ./)

# Includes
include_directories (../include)

# Libraries
set (EXTRA_LIBS ${EXTRA_LIBS} glog gflags pthread xerces-c zmq zmqpp mgl
    )

# Sources
file (GLOB SRC *.cpp ComputationModels/*.cpp DecRAIDSchemes/*.cpp
    Drivers/*.cpp GraphGeneration/*.cpp LayoutModels/*.cpp Modules/*.cpp
    Structures/*.cpp Utils/*.cpp XMLParsers/*.cpp)

# Definitions
set (CMAKE_CXX_FLAGS_DEBUG ${CMAKE_CXX_FLAGS} "-std=gnu++11 -fext-
    numeric-literals -std=c++11 -O0 -g")
set (CMAKE_CXX_FLAGS_RELEASE ${CMAKE_CXX_FLAGS} "-std=gnu++11 -fext-
    numeric-literals -std=c++11 -O3 -DNDEBUG")

# Executable
message (STATUS ${EXTRA_LIBS})

add_executable (OGSSim ${SRC})
target_link_libraries (OGSSim ${EXTRA_LIBS})
install (TARGETS OGSSim DESTINATION .)
```

Code 2.1 – Exemple de fichier de description CMake

l'aide de balises insérées dans le code source, Doxygen génère la documentation du projet dans le format HTML ou Latex. Étant donné l'aspect ouverture d'OGSSim, cela permet à n'importe quel utilisateur souhaitant intégrer un nouvel algorithme de placement, de gestion permanente ou un nouveau modèle de calcul, d'avoir les informations suffisantes pour une implémentation rapide et simple.

Le code 2.2 montre un exemple de documentation Doxygen pour la fonction principale d'un modèle de calcul de temps d'exécution. En prenant l'exemple du modèle de calcul des disques magnétiques, le temps retourné serait le temps de service de la requête donnée en paramètre sur le HDD lié à l'instance du modèle.

### c) xerces-c – extraction/sérialisation de fichier XML

XML (Extensible Markup Language) est un langage informatique proposant, entre autres, un stockage d'informations de manière hiérarchisée au sein de fichiers. La bibliothèque xerces-c (Apache Software Foundation, 2016) offre des mécanismes d'extraction et de sérialisation d'informations au format XML. Comparée aux autres bibliothèques

```
/**
 * Function which computes the execution time of a given
 * request for the simulated component of the system.
 *
 * @param   idxRequest          Request index.
 * @return                      Execution time.
 */
    double compute (
        const long              idxRequest);
```

Code 2.2 – Exemple d'utilisation de balises Doxygen

existantes, elle est celle offrant le meilleur respect du format, et possède également une syntaxe plus proche du C++ que Libxml2 (Veillard, 2016) par exemple.

### d) google glog – journalisation

La journalisation est un mécanisme intéressant à ajouter dans le développement de notre outil de simulation car il permet de suivre le déroulement de l'exécution, et de vérifier le bon comportement des modules implémentés en surveillant l'état des données. La journalisation est gérée dans OGSSim avec la bibliothèque glog (Google, 2016) de Google. Cette bibliothèque permet une journalisation sur plusieurs niveaux (debug, information, avertissement, erreur ou fatal), une redirection des messages dans un fichier de sortie ou encore des messages conditionnés. La différence entre les niveaux erreur et fatal vient de la nécessité ou non d'arrêter l'exécution du simulateur. Par exemple, le niveau erreur peut être utilisé lorsqu'une erreur de calcul est détecté, mais sans impact sur le fonctionnement d'OGSSim tandis que le niveau fatal est utile lorsqu'un problème plus important intervient, comme une mauvaise allocation mémoire, et qui nécessite l'arrêt de l'exécution.

La syntaxe de création des messages est semblable à celle de l'utilisation de la sortie standard, et se fait par la macro donnée dans le code 2.3. Dans cet exemple, le message `"test"` sera journalisé avec le niveau d'information.

```
LOG(INFO) << "test";
```

Code 2.3 – Exemple de macro de journalisation

Un exemple de fichier de journalisation est donné dans le code 2.4. On peut y voir qu'en plus des informations donnés par la macro `LOG()`, la date de création du message, le niveau de journalisation du message, l'identifiant du thread appelant et le numéro de ligne d'appel de la macro dans l'application sont également renseignés. Le but est de fournir un maximum de détails sur le contexte lié au message de journalisation.

### e) ZeroMQ – communication

Chaque module de l'outil de simulation est géré de manière indépendante dans un environnement multi-threadé. Ceci oblige l'utilisation d'un mécanisme de communication pour relayer les informations entre ces différents modules. Le mécanisme choisi est celui offert par la bibliothèque ZeroMQ (iMatix Corporation, 2016) qui implémente

```
Log file created at: 2016/06/01 13:35:17
Running on machine: sego-OptiPlex-9020
Log line format: [IWEF]mmdd hh:mm:ss.uuuuuu threadid file:line] msg
I0601 13:35:17.513254 20921 performanceevaluation.cpp:19] Perf module is on!
I0601 13:35:17.513854 20916 workload.cpp:47] Get 1 requests
I0601 13:35:17.699247 20920 main.cpp:229] Launching simulation
I0601 13:35:17.699250 20919 preprocessing.cpp:151] Launch simulation for 1 req
I0601 13:35:17.699265 20919 preprocessing.cpp:158] All requests were distributed
I0601 13:35:17.699267 20919 preprocessing.cpp:170] Send termination messages
I0601 13:35:18.053114 20934 ivolume.cpp:254] Receive event for dev #3 for date 130
I0601 13:35:18.053143 20934 ivolume.cpp:160] Need to launch event: 130 < 146.408
I0601 13:35:18.053218 20940 devicedriver.cpp:114] Dev #3 will be faulty at 130
I0601 13:35:18.107903 20934 decraiddriver.cpp:197] Need rebuilding 50919 blocks
I0601 13:35:18.108726 20934 decraiddriver.cpp:223] Write --> 15
I0601 13:35:18.108811 20981 devicedriver.cpp:126] Device #15 receives read request
I0601 13:35:18.108870 20920 execution.cpp:119] Receives #2 from dev #15 of type 4
I0601 13:35:18.108888 20920 execution.cpp:230] Device #15 computation in 16.8024ms
```

Code 2.4 – Exemple de fichier de journalisation

des sockets de communication selon plusieurs modèles comme le fil d'abonnement ou le producteur/consommateur.

Les points positifs de cette bibliothèque sont : une facilité d'utilisation ainsi qu'une grande efficacité. L'utilisation des ZeroMQ au sein d'OGSSim, et des communications qu'elles engendrent, sera détaillée dans le chapitre 4.

#### f) MathGL – création de graphes

Les graphes de visualisation peuvent être directement créés avec OGSSim. La génération est faite à l'aide de la bibliothèque MathGL (Balakin, 2016), pouvant construire des graphiques en deux ou trois dimensions, des courbes ou des histogrammes, et gérant plusieurs format, dont le PNG et le SVG (images vectorielles).

L'utilisation de la bibliothèque peut se faire avec une syntaxe C++, où chaque composant est un membre de la classe représentant le graphe. Le code 2.5 montre un exemple d'utilisation de cette bibliothèque, avec l'affectation du point d'origine du graphe, de la longueur de ses axes et de l'ajout de sa légende.

## 2.3 Fichiers d'entrée-sortie d'OGSSim

OGSSim récupère les paramètres de la simulation à partir de fichiers d'entrée. Les paramètres lus sont ceux concernant directement le simulateur, la trace des requêtes et l'architecture du système simulé. Les résultats sont eux exportés dans d'autres fichiers, et présents sous différentes formes.

#### a) Fichiers d'entrée

- Fichier de configuration d'OGSSim

Formaté en XML, ce fichier possède les informations nécessaires à la bonne invocation d'OGSSim. Il apporte entre autres le paramétrage des sockets de communication ZeroMQ entre chaque module, celui de la journalisation (niveau d'information, fichier

```
// Set the graph parameters
graph.SetOrigin (0, 0, 0);
graph.SetRanges (0, numColumns + 1, 0, 120);

graph.AddLegend ("work", "r");
graph.AddLegend ("idle", "n");

graph.SetTicksVal ('x', labelsDT, labelStream.str () .
    c_str () );
graph.Box ();
graph.Bars (ticksDT, valuesDT, "arn");
graph.Axis ();
graph.Grid ("y", "k;");
graph.Legend ();

graph.Label ('x', "devices", 0);
graph.Label ('y', "time (%)", 0);

graph.Title (titleStream.str () .c_str () );
```

CODE 2.5 – Exemple de code de création de graphe

de sortie) ainsi que la taille de la structure utilisée pour stocker les requêtes temporaires. On appelle requête temporaire, ou sous-requête, une requête issue de la décomposition d'une requête utilisateur. C'est également dans ce fichier que sont renseignés les événements ciblant le système simulé, les demandes de graphes de visualisation ainsi que les chemins de l'ensemble des fichiers d'entrée et de sortie.

Dans l'exemple donné par le code 2.6, on peut distinguer les informations de la configuration du simulateur, comme les chemins d'accès aux autres fichiers d'entrée et de sortie (lignes 5 à 9), le réglage de la journalisation (ligne 11), des files de communication (lignes 14 et 20) ou encore des graphes de visualisation (lignes 36 à 37).

- Fichier de trace

Le fichier de trace, formaté en RAW, contient les champs des requêtes utilisés durant la simulation. Plusieurs formats de requêtes sont possibles. Ces formats sont décrits dans la table 2.1. Le premier contient uniquement les données nécessaires à l'exécution de la requête, à savoir la date d'arrivée de la requête depuis le début de la simulation, le type de la requête, l'adresse logique de début des données et leur taille. Le second format contient, en plus des champs obligatoires détaillés précédemment, un identifiant regroupant, au choix de l'utilisateur, des ensembles de requêtes. Ces groupes peuvent dépendre du type d'application utilisée (bureautique, calcul, etc.) ou de l'importance des données ciblées par les requêtes (cryptées, en clair). Le dernier format ajoute aux champs obligatoires deux identifiants permettant un profilage plus détaillé de la simulation : le numéro de l'hôte et du processus.

Un exemple de fichier de trace est donné par le code 2.7, où un ensemble de requêtes sont décrits suivant le format n°1.

Pour simuler des requêtes successives, il est possible d'indiquer pour une plage de requête la même date d'arrivée. Ces requêtes seront alors ordonnées selon leur estam-

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Parameter file for OGSSim -->
<config>
    <path>
        <workloadfile>env/trace/trace.data</workloadfile>
        <hardwarefile>env/arch/arch-ssdtest.xml</hardwarefile>
        <resultfile>env/result/result.data</resultfile>
        <subresultfile>env/result/sres.data</subresultfile>
        <unitarytestfile>ut-output.xml</unitarytestfile>
    </path>
    <general>
        <log mlvl="0" file="log-ssdprobe-" />
    </general>
    <event>
        <zeromq intr="preproc" prot="tcp" addr="localhost" port="5559"/>
        <entry dev="6" date="120" type="fault" />
        <entry dev="2" date="130" type="fault" />
        <entry dev="2" date="520" type="replace" />
    </event>
    <workload>
        <zeromq intr="preproc" prot="tcp" addr="localhost" port="5555"/>
        <subreq bsiz="50000" />
        <reqdut size="512" />
    </workload>
    <hardware>
        <zeromq intr="preproc" prot="tcp" addr="localhost" port="5556"/>
    </hardware>
    <preproc>
        <zeromq intr="workload" prot="tcp" addr="*" port="5555" />
        <zeromq intr="hardware" prot="tcp" addr="*" port="5556" />
        <zeromq intr="event" prot="tcp" addr="*" port="5559" />
    </preproc>
    ...
    <visualization>
        <zeromq intr="preproc" prot="tcp" addr="*" port="5560" />
        <zeromq intr="execution" prot="tcp" addr="*" port="5561" />
        <graph type="devbehavior" volume="all" output="gr1.png" />
        <graph type="reqpercentile" size="20" output="gr2.png" />
    </visualization>
</config>
```

Code 2.6 – Exemple de fichier de configuration d'OGSSim

```
#Trace file
#Date Type Address Size
1.14156 0 1631855330 4
1.24346 0 3544738046 4
1.84049 0 3287788364 4
2.1625 1 3643476217 4
2.7827 0 1641661534 4
2.8482 0 190842183 4
2.93569 0 652436336 4
6.65846 0 1437469849 4
```

Code 2.7 – Exemple de fichier de trace

| Format | | | Champ | Description |
|---|---|---|---|---|
| 1 | 2 | 3 | | |
| X | X | X | Date | Date d'arrivée depuis le début de la simulation en ms |
| X | X | X | Type | Type de la requête, 0 pour lecture, 1 pour écriture |
| X | X | X | Adresse | Adresse logique des données ciblées en du |
| X | X | X | Taille | Taille des données ciblées en du |
| | X | | Groupe | Numéro du groupe d'applications demandant la requête |
| | | X | Hôte | Numéro de l'hôte demandant la requête |
| | | X | Processus | Numéro du processus demandant la requête |

TABLE 2.1 – Formats des requêtes disponibles dans OGSSim

pille (leur ordre dans le fichier). L'unité pour l'adresse et la taille des données est le *data unit* (du). Il s'agit d'une unité propre à OGSSim utilisée pour quantifier l'espace disponible sur le système de stockage. Cette unité est également utilisée dans le fichier de configuration architecturale, pour exprimer les tailles des *stripe units* par exemple.

- Fichier de configuration architecturale

Le fichier de configuration architecturale apporte le détail des composants du système simulé, à savoir les bus, les configurations et les disques. Les informations sur les disques utilisés sont donnés dans leur fichier de description respectif.

Un exemple de fichier de configuration architecturale est donné dans le code 2.8. Le fichier est composé de deux parties distinctes, celle réservée aux bus délimitée par la balise `<buses>` et celle pour la configuration générale du système délimitée par la balise `<system>`.

Le fichier de configuration architecturale décrit pour chaque élément du modèle, le nombre de sous-éléments qu'il contient et les bus utilisés grâce à leur nom logique. Chaque volume possède deux balises. La balise `<config>` donne des informations sur la configuration du volume. Il renseigne le type de schéma de placement utilisé et, si nécessaire, le nombre de disques de parité, le *declustering* utilisé ainsi que la taille des *stripes* de données. La balise `<device>` donne un pointeur sur le fichier décrivant les paramètres du disque utilisé par le volume, lui aussi formaté en XML. Un exemple de fichier de description de disque est donné code 2.9. Chaque fichier est composé de six parties distinctes :

— `<information>` : informations nominales du disque et capacité ;

— `<geometry>` : informations sur la composition du disque (nombre de secteurs, de pages, etc.) ;

— `<technology>` : informations sur les interfaces utilisables sur le disque (ATA, SATA, PCIe, etc.) ;

— `<performance>` : informations sur la performance (bande-passante, nombre d'opérations I/O, etc.) ;

```xml
<?xml version="1.0" encoding="utf-8"?>
<!--
     Parameter file for architecture
-->
<architecture>
    <buses nbbuses="4">
        <bus name="B0" nbports="17" bandwidth="640" type="SCSI" />
        <bus name="B1" nbports="17" bandwidth="640" type="SCSI" />
        <bus name="B2" nbports="17" bandwidth="640" type="SCSI" />
        <bus name="B3" nbports="17" bandwidth="640" type="SCSI" />
    </buses>
    <system nbtiers="1" bus="B0">
        <tier nbvolumes="2" bus="B1">
            <volume nbdevices="5" bus="B2">
                <config type="JBOD" stripeunitsize="8388608" />
                <device file="env/disk/hdd_hitachi_A7K1000.xml" />
            </volume>
            <volume nbdevices="5" bus="B3">
                <config type="JBOD" />
                <device file="env/disk/ssd_transcend_370.xml" />
            </volume>
        </tier>
    </system>
</architecture>
```

CODE 2.8 – Exemple de fichier de configuration architecturale

— `<reliability>` : informations sur la fiabilité du disque ;

— `<feature>` : informations sur les algorithmes de gestion permanente du disque.

### b) Fichiers de sortie

- Fichier de résultats

Le fichier principal de sortie d'OGSSim est le fichier de résultats. Il donne entre autres pour chaque requête utilisateur son temps de réponse et sa décomposition (temps d'attente, de transfert, de service). C'est à partir des informations contenues dans ce fichier que les différentes métriques de performance et de robustesse peuvent être calculées, telles le taux d'utilisation des disques, la distribution des requêtes sur l'espace de stockage et la proportion des requêtes non exécutées en mode défaillant.

- Fichier de résultats intermédiaires

Ce fichier propose une version détaillée des résultats contenus dans le premier fichier de sortie en indiquant les temps de réponse, les disques ciblés et autres informations relatives aux sous-requêtes utilisateur et système. Une sous-requête est créée lorsqu'une requête utilisateur cible des plages de données non contigües et/ou plusieurs disques.

- Graphes de visualisation

```xml
<?xml version="1.0" encoding="utf-8"?>
<!--
    Parameter file for SSD Transcend SSD370
-->
<device type="ssd">
    <information>
        <name>Transcend SSD370</name>
        <capacity>256</capacity>
        <arch>MLC</arch>
        <producer>Transcend</producer>
        <year>2015</year>
    </information>
    <geometry>
        <pagesize>16384</pagesize>
        <pagesbyblock>512</pagesbyblock>
        <blocksbydie>2048</blocksbydie>
        <nbdies>16</nbdies>
    </geometry>
    <technology>
        <ata extended="off" ncq="on" security="off" smart="off" />
        <sata bandwidth="6000000000" revision="3.0" />
    </technology>
    <performance>
        <randread unit="iops" size="4096">70000</randread>
        <randwrite unit="iops" size="4096">70000</randwrite>
        <seqread unit="mbps">560</seqread>
        <seqwrite unit="mbps">320</seqwrite>
        <erase>3.8</erase>
        <buffersize>32</buffersize>
    </performance>
    <reliability>
        <nberase>3000</nberase>
    </reliability>
    <feature>
        <allocator>default</allocator>
        <probe>default</probe>
        <garbagecollector>default</garbagecollector>
        <wearlevelling>default</wearlevelling>
    </feature>
</device>
```

Code 2.9 – Exemple de fichier de description de disque

Demandés par l'utilisateur, les graphes de visualisation sont des fichiers PNG créés à l'aide de la bibliothèque MathGL. Trois types de graphes sont disponibles dans la version actuelle d'OGSSim : graphe du temps de réponse moyen des requêtes, visualisation de l'utilisation des disques et visualisation de l'utilisation des bus par le biais d'histogrammes ainsi que la proportion des requêtes non exécutées et celle des données reconstruites dans le cas du mode défaillant.

- Fichier de journalisation

Le fichier de journalisation, rédigé avec la bibliothèque glog, regroupe des informations sur le déroulement de la simulation. Ces informations sont demandées au sein du code source de l'application.

# 3 Conclusion

OGSSim est un outil de simulation pour des systèmes de stockage hétérogènes à large échelle. Il a été développé dans l'optique de proposer une solution ouverte et générique, pour tenir compte de l'avancée technologique actuelle et future du stockage. OGSSim supporte ainsi des systèmes composés de centaines de disques, HDD ou SSD. Des technologies plus récentes, telles les SSD NVMe, sont prévues pour également être supportées dans un futur proche.

Son architecture modulaire permet l'ajout simple de nouveaux modèles de calcul ou de schémas de placement des données. Chaque module, spécifiant un rôle précis dans la simulation, est attaché à un thread. L'utilisation du multi-threading résulte en un parallélisme des tâches ayant pour but le traitement de centaines de milliers de requêtes d'entrée/sortie en un temps acceptable.

Plusieurs fichiers sont nécessaires à la correcte initialisation d'OGSSim. Rédigés principalement en XML, ils permettent le paramétrage des différentes bibliothèques utilisées, chargées entre autre de la communication inter-thread, de la journalisation ou encore de la génération de graphes de visualisation. Ces fichiers servent également à la définition de la configuration architecturale et du jeu de requêtes utilisés. L'implémentation d'OGSSim sera détaillée dans le chapitre suivant.

# Chapitre 3

# OGSSim : implémentation et validation



*OGSSim est une application modulaire qui octroie à chaque module un rôle précis. Comme observé sur la figure 2.1 qui présente le schéma conceptuel d'OGSSim, on peut distinguer trois catégories de modules. Au lancement de l'outil de simulation, les modules d'extraction permettent de récupérer les données d'initialisation et de positionner l'environnement. Les modules de la chaîne de décomposition permettent de transformer les requêtes logiques issues de la trace en sous-requêtes physiques ciblant les disques du système simulé. Les modules de calcul et de sortie servent à calculer temps de réponse des requêtes et indicateurs de performances, afin de délivrer les fichiers de résultats.*

*Ce chapitre détaille les modules composant les trois chaînes d'OGSSim. Puis nous discuterons de la phase de validation d'OGSSim, en confrontant notre logiciel à deux SSDs et deux HDDs de constructeurs différents. Enfin, dans les perspectives, nous parlerons de la prochaine approche d'implémentation d'OGSSim ainsi que la validation de notre outil contre des systèmes de stockage réels.*

# 1 Chaîne d'extraction

## 1.1 Extraction de trace

A l'initialisation du programme, le module d'extraction de trace (*Workload*) récupère le chemin du fichier de trace à partir du fichier de configuration. Le fichier de trace est alors ouvert par le module et les informations sur les requêtes extraites.

Les informations extraites sont stockées dans une structure de données dont seule l'adresse mémoire est ensuite communiquée au module de pré-traitement.

## 1.2 Extraction d'architecture

Le module d'extraction d'architecture (*Hardware Configuration*) agit également au lancement du programme. Son rôle est de construire la structure de données contenant l'architecture du système simulé. Le chemin du fichier, formaté en XML, est indiqué dans le fichier de configuration du simulateur.

L'extraction des fichiers XML se fait par l'intermédiaire de la classe *XMLParser*, qui est une surcouche de l'API proposée par la bibliothèque xerces-c. L'extraction d'informations peut se faire de deux manières. Soit lors d'une extraction ponctuelle, en indiquant les paramètres de l'information recherchée, tels que le nom de la balise, ou celui de son argument si besoin. Soit lors d'une extraction continue, comme c'est le cas ici. Dans ce cas, l'arbre des balises est lu en profondeur, nœud après nœud.

Chaque bus est décrit par son nom logique, le nombre de connexions qu'il peut supporter, sa bande-passante ainsi que son type. Ce dernier champ est informatif dans la version actuelle du simulateur. Mais il pourra être utilisé ultérieurement dans le cas où des modèles de calcul pour le transfert ont besoin d'être spécifiés selon le type de canal employé. Par exemple, avec l'arrivée des NVMe, le modèle de calcul pour le transfert PCIe est différent du modèle implémenté pour le SCSI de par la différentiation entre les buffers pour les requêtes et la prise en compte des confirmations.

Le système architectural est ensuite décrit de manière hiérarchique selon la topologie tier/volume/device. Cette topologie, représentée figure 3.1, est constitué de différents niveaux, représentant un élément logique ou physique. Le premier niveau (a), appelé *tier*, est utilisé pour séparer les volumes d'un système. Le second niveau (b) contient les volumes. Chaque volume possède un nombre donné de disques et un schéma de placement des données sur ces disques. Les schémas de placement possibles dans OGSSim sont décrits dans la section 2. Les disques d'un même volume sont tous identiques. Le dernier niveau (c) du modèle, appelé *device*, représente le support physique de stockage, à savoir actuellement le disque, qui peut être soit un HDD, soit un SSD. Les bus relient les éléments entre eux, à savoir : l'hôte avec l'ensemble des *tiers*, chaque *tier* avec ses volumes et chaque volume avec ses supports physiques.

A la fin de l'extraction, les informations sont stockées dans la structure de données `Architecture` présentée dans le code 3.1. La classe `Geometry` indique le nombre de bus, tier, volume et device définis pour le système simulé. Les autres pointeurs sont des tableaux indicés entre 0 et le nombre d'éléments de chaque type, possédant les

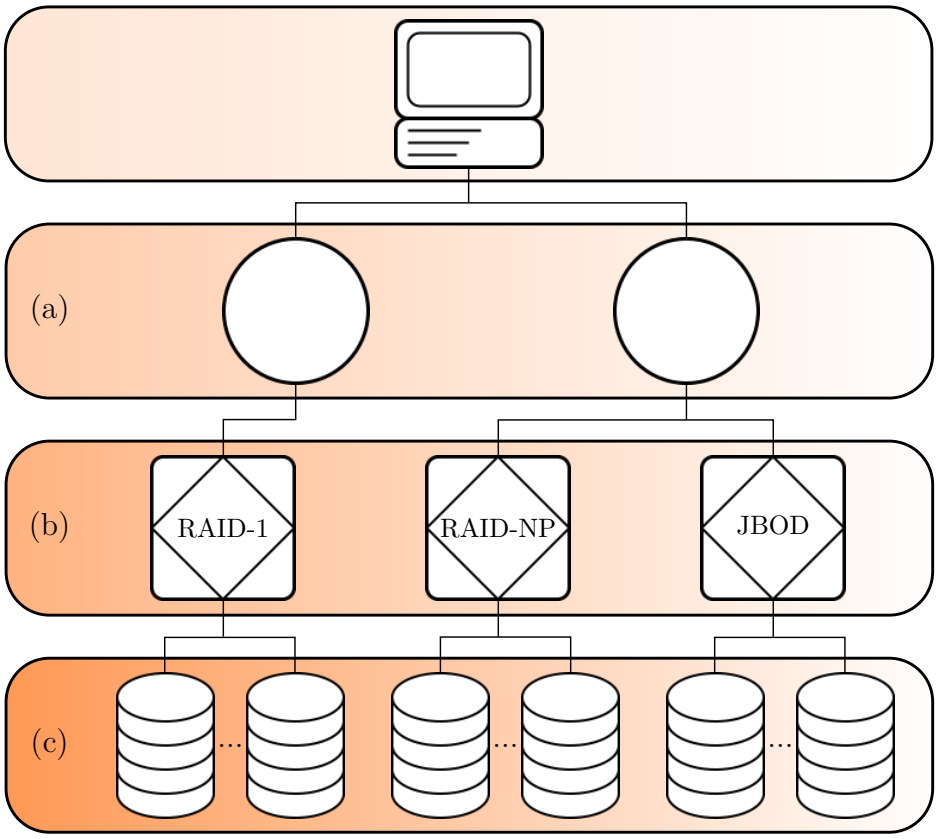



FIGURE 3.1 – Modèle tier (a) / volume (b) / device (c)

paramètres des éléments du système. Chaque élément possède un pointeur vers son élément père dans la hiérarchie symbolisé par son indice dans le tableau. Toutes les informations sur la configuration architecturale sont alors réunies au sein d'une même structure, dont le pointeur est communiqué au module de pré-traitement.

```
1 struct Architecture {
2     Geometry        * _geometry;
3     Bus             * _buses;
4     Tier            * _tiers;
5     Volume          * _volumes;
6     Device          * _devices;
7 };
```

CODE 3.1 – Structure Architecture

## 1.3 Extraction d'événements

L'extraction d'événements (*EVENT*) est le dernier mécanisme à agir avant le lancement de la simulation. Il va extraire du fichier de configuration d'OGSSim, formaté en XML, les informations concernant les événements arrivant durant la simulation. Les informations sur les événements sont contenus dans la balise `<event>`. Un exemple de cette balise est donné par le code 3.2. En plus de contenir les informations sur les files de communication, chaque événement est renseigné à l'aide de la balise `<entry>`. Un événement se caractérise par son type et la date à laquelle il se produit durant la simulation.

```
<event>
    <entry dev="6" date="120" type="fault" />
    <entry dev="2" date="130" type="fault" />
    <entry dev="2" date="520" type="replace" />
</event>
```

Code 3.2 – Partie événements dans le fichier de configuration du simulateur

Les événements sont stockés dans une structure de données dont le pointeur est communiqué au module de pré-traitement. La gestion des événements dans le processus de simulation sera détaillé dans le chapitre 5.

Actuellement, seuls les événements de type défaillance sont implémentés. Pour une défaillance, le disque ciblé doit aussi être indiqué. D'autres types d'événements peuvent facilement être ajoutés, tel le remplacement de disque. L'ajout d'un type d'événement se caractérise essentiellement par des surcharges de fonctions au sein des modules de la chaîne de décomposition, permettant de traiter l'information. Par exemple, là où la défaillance intervenait au sein du pilote de disque pour mettre à jour l'état du support de stockage en l'indiquant comme défaillant, le remplacement interviendrait au même endroit pour remettre l'état à sain.

# 2 Chaîne de décomposition

## 2.1 Pré-traitement

Le module de pré-traitement (*Preprocessing*) a pour tâche de rassembler l'ensemble des données d'initialisation de la simulation, d'instancier les modules de pilote de volume et de lancer le processus de simulation, comme montré dans la figure 2.1.

Les données d'initialisation sont envoyées par les modules d'extraction de trace, d'architecture et d'événements une fois le parcours des fichiers d'entrée effectué. Ces données seront par la suite transmises à l'ensemble des modules du simulateur par leurs pointeurs respectifs, afin que les communications soient les plus rapides possibles.

Une fois la réception des données faite, le module de pré-traitement va instancier un pilote de volume pour chaque volume du système. L'instance créée dépend de la configuration associée au volume. Par exemple, si la configuration est de type RAID-NP, alors le pilote de RAID-NP est sélectionné.

Au lancement de la simulation, le module de pré-traitement parcourt le tableau des requêtes, trié par date d'arrivée, et redirige les requêtes vers le pilote du volume ciblé. Cette redirection se calcule à l'aide d'une table indiquant pour chaque volume, sa taille. Un exemple est donné table 3.1, pour un système à quatre volumes : les adresses accessibles pour le premier volume de l'adresse 0 jusqu'à l'adresse 512M ; le second volume de 512M jusqu'à 2048M ; le troisième volume de 2048M jusqu'à 8912M et le quatrième volume de 8912M jusqu'à 10960M.

| $i$ | adresse de fin |
|---|---|
| 1 | 512M |
| 2 | 2 048M |
| 3 | 8 912M |
| 4 | 10 960M |

TABLE 3.1 – Exemple de table de redirection du module de pré-traitement

L'algorithme 2 retranscrit le mécanisme de redirection. Pour une adresse donnée, la table de redirection est parcourue jusqu'à ce que l'adresse de la requête soit inférieure à la taille indiquée dans le tableau. L'indice trouvé correspond au numéro du volume ciblé par la requête.

**Algorithme 2 :** Redirection des requêtes lors du pré-traitement

**Entrées :** $r$ – requête
$T$ – table de redirection
$n_v$ – nombre de volumes dans le système
**Sorties :** $v$ – volume ciblé

```
1 pour v allant de 1 à n_v faire
2     si T[v] > r.adresse alors
3         retourner
4     fin
5     v ← v + 1
6 fin
```

## 2.2 Pilote de volume

Le module du pilote de volume (*VOLUME DRIVER*) a pour tâche le calcul de la redirection des requêtes arrivant dans une configuration de disques donnée. Les requêtes reçues du module de pré-traitement seront analysées, et si besoin décomposées, puis redirigées vers les pilotes des disques ciblés.

Chaque configuration ayant son propre schéma de placement, il est nécessaire de spécialiser le pilote de volume en fonction du système simulé. L'interface du pilote est définie dans le code 3.3. La majorité des fonctions de la classe sont communes à toutes les configurations : `executeSimulation()` est la fonction générale du pilote qui, durant la simulation, va recevoir les indices de requêtes à traiter avec `receiveRequest()`, les décomposer avec `decomposeRequest()` puis les envoie au pilote de disque avec `sendRequest`. Une fois toutes les requêtes terminées, à la réception de la requête de terminaison, le pilote de volume envoie cette même requête à tous les pilotes de disque qui le composent à l'aide de `terminateTreatment()`.

Les algorithmes 14 à 24 présentent les schémas de redirection utilisés pour chacune des configurations rencontrées dans OGSSim et présentées dans le chapitre 1. Ils sont disponibles dans l'annexe 5, par souci de lisibilité.

L'algorithme 14 détaille la redirection effectuée dans un JBOD. Le comportement du JBOD est de stocker les données sur un disque après l'autre. Ainsi, l'algorithme va

```
class IVolume {
public:
    void executeSimulation ();
    virtual void decomposeRequest (
        const long          idxRequest,
        std::vector <long>  & subrequests) = 0;

protected:
    void terminateTreatment ();
    inline bool receiveRequest (
        long                & idxRequest);
    inline void sendRequest (
        const long          idxRequest);
};
```

Code 3.3 – Interface du pilote de volume

rechercher, à partir du disque ciblé par l'adresse de départ de la requête, quels sont les disques qui vont contenir les données ciblées. Cette recherche est faite à l'aide de la boucle ligne 5 basée sur la taille de la plage de données restante.

La redirection pour les RAID-0, inclut l'utilisation du *striping* pour la répartition des données. Chaque disque est donc divisé en lignes de données, appelées *stripes*, et chaque *stripe* doit être entièrement plein avant de passer au suivant. La redirection d'une requête dans cette configuration est expliquée dans l'algorithme 15. La majeure différence avec la redirection pour les JBOD est le pas de la boucle : au lieu d'itérer sur les disques, l'itération est faite sur les unités de *striping*.

Le RAID-1 utilise le *mirroring*, ce qui signifie que chaque disque de donnée est associée à un disque miroir possédant une copie de ces mêmes données. En lecture, le disque natif ou miroir est sélectionné pour y délivrer les données. Dans le cas d'une écriture, les données doivent être écrites sur les deux disques. Ce comportement est décrit dans l'algorithme 16.

Un RAID-01 est la fusion des concepts de *striping* et de *mirroring*, la configuration est découpée en deux sous-ensembles et pour chaque disque du premier sous-ensemble, un disque dans le second sous-ensemble possède les mêmes données. A cet effet, la redirection effectuée est identique à celle du RAID-0, à ceci près qu'en cas d'écriture, les données sont envoyées sur les deux sous-ensembles. Si la requête est une lecture, alors elle est envoyée selon la stratégie adoptée, soit sur les disques de données natives, soit sur les disques miroirs. L'algorithme 17 décrit ce comportement. La stratégie de redirection des lectures peut être aléatoire, ou circulaire. Le choix fait dans OGSSim est de les rediriger de manière circulaire.

Enfin, la dernière spécialisation concerne les RAID-NP, utilisant les codes d'erreur pour la robustesse des données. Nous allons décrire ici les algorithmes en considérant un RAID-5, une configuration avec un disque de parité, et du declustering de parité. Pour plus de lisibilité, les redirections des requêtes de lecture et d'écriture sont séparées.

L'algorithme de redirection des lectures est donné dans l'algorithme 18. Tout d'abord, il va déterminer le nombre de *stripes* à traiter. Si les données ne sont présentes que

sur un seul *stripe* (lignes 5 à 10), alors la requête est décomposée en déterminant les disques ciblés. La seconde partie de l'algorithme traite le cas où la requête cible plusieurs *stripes* (lignes 12 à 31). On va déceler ici deux cas : soit la lecture ne concerne qu'une partie du *stripe* et on utilise `readPartStripe()`, soit le *stripe* est lu dans sa totalité, et dans ce cas on va utiliser la méthode `readFullStripe()`. Les définitions des différents types de traitement du *stripe* sont disponibles dans le chapitre 1.

Les méthodes `readPartStripe()` et `readFullStripe()` sont décrites respectivement dans les algorithmes 19 et 20. Si la lecture est partielle, nous calculons alors la plage de disques ciblés grâce à l'adresse de début et la taille des données sur le *stripe*. Dans le cas de la lecture du *full stripe*, tous les *stripe units* sont les cibles d'une nouvelle sous-requête.

La redirection de la requête d'écriture, donnée par l'algorithme 21 diffère de la lecture par la mise à jour de la parité. Même si l'algorithme est toujours séparé en deux parties, une pour le cas où la requête cible des données d'un seul *stripe* (lignes 5 à 12) et l'autre lorsque plusieurs *stripes* sont concernés (lignes 14 à 39), le traitement d'un *stripe* partiel va maintenant dépendre de la taille des données ciblés sur le *stripe*. Soit la taille des données est inférieure à la moitié de la taille d'un *stripe*, et dans ce cas nous utilisons la méthode `writeSmallStripe()`. Soit cette taille est supérieure à la moitié de la taille d'un *stripe* et nous appelons la méthode `writeLargeStripe()`. Le cas où toutes les données du *stripe* sont ciblés est toujours géré par une seule méthode, `writeFullStripe`.

La méthode `writeSmallStripe()`, décrite dans l'algorithme 22, se décompose en deux parties : une première où le bloc de parité est lu puis écrit (lignes 3 à 9), puis une seconde où les données ciblées sont traitées de la même manière (lignes 10 à 23).

Le traitement du *large stripe* est différent du *small stripe* : les données du *stripe* lues pour la mise à jour de la parité sont celles qui ne sont pas ciblées par la requête. L'algorithme 23 décrit ce traitement. On distingue ici quatre parties : la lecture et l'écriture du bloc de parité (lignes 3 à 9), la lecture des blocs antérieurs aux données ciblées (lignes 12 à 19), l'écriture des nouvelles données (lignes 21 à 26) et enfin la lecture des blocs postérieurs aux données ciblées (lignes 28 à 34).

Le traitement en écriture d'un *full stripe*, comme indiqué dans l'algorithme 24 se résume en l'écriture de tous les blocs du *stripe*, données comme parité.

## 2.3 Pilote de disque

Le module du pilote de disque (*Device Driver*) est le dernier module de la chaîne de décomposition des requêtes. Son principal rôle est le maintien de l'état du disque assuré par un ensemble de sous-modules de gestion permanente : allocateur, sonde, défragmentation, égalisation d'usure et ramassage de miettes.

Les sous-modules de l'allocateur et de la sonde sont propres à l'ensemble des disques. L'allocateur s'occupe de la traduction logique/physique de la plage de données. La sonde met à jour, pour chaque requête traitée, les informations du disque demandées par l'utilisateur, comme par exemple le nombre d'accès à une page de données.

Les autres sous-modules sont spécifiques à un type de disque. Deux choix s'offrent alors à nous en ce qui concerne l'implémentation du module. Soit nous décidons de garder un pilote de disque générique auquel se greffe les sous-modules de gestion sans spécification, ce qui signifie que nous ne faisons pas de distinction entre les différents types de sous-modules. Soit nous spécifions le pilote de disque par deux classes filles, une pour les HDDs et une pour les SSDs afin de leur associer directement les sous-modules de gestion permanente correspondants.

Notre choix s'est porté sur la seconde possibilité. En privilégiant la spécialisation des pilotes, et la non généricité des sous-modules de gestion, nous avons la possibilité de définir pour chaque type de sous-module, une interface propre. Par exemple, les appels à un mécanisme d'égalisation d'usure, qui peuvent être synchrones (appelé à chaque période) ou asynchrones (pour chaque allocation de page) ne sont pas identiques à ceux d'un mécanisme de défragmentation qui sont complètement asynchrones (généralement demandés par l'utilisateur). Ce choix permet également de renforcer le côté ouverture d'OGSSim, en facilitant l'intégration de nouvelles technologies de disque accompagnées de leurs mécanismes de gestion permanente.

L'implémentation de la gestion permanente des disques ne dépend pas uniquement des algorithmes propres à ces mécanismes. La figure 3.2 montre la décomposition du module de pilote de disque. Pour rappel, ce module fait le lien entre le pilote de volume, qui se charge de la décomposition des requêtes de l'utilisateur en sous-requêtes, dépendant de la configuration du volume, et le module d'exécution, calculant les métriques de performance.

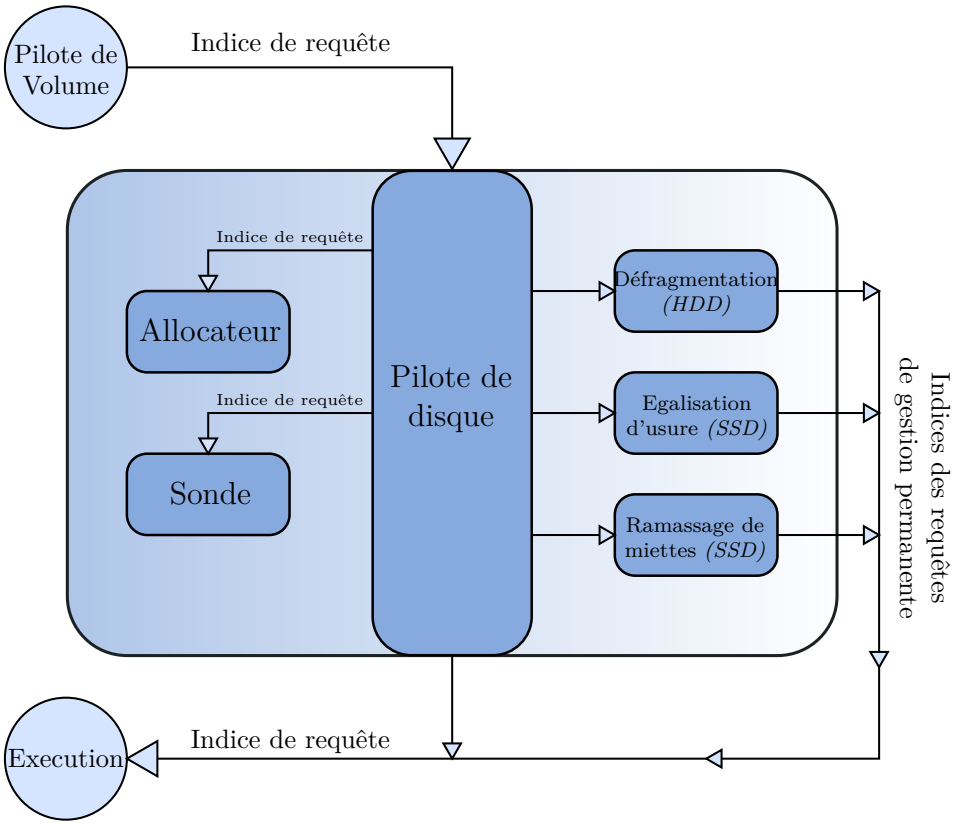



FIGURE 3.2 – Modèle du pilote de disque d'OGSSim

A la réception d'un indice de requête de la part du pilote de volume, le pilote de disque va désormais appeler l'allocateur (*Allocator*), le sous-module chargé de la traduction logique/physique des adresses des requêtes. Une fois l'adresse physique obtenue, l'index de la requête va être donnée au sous-module de sonde (*Probe*) pour la mise à jour de l'état du disque. Ces deux sous-modules sont utilisables par les deux types de pilotes implémentés actuellement dans OGSSim.

Les autres sous-modules sont eux spécifiques à la technologie du disque. Ainsi, le sous-module de défragmentation (*Defragmentation*) est utilisable par les pilotes de HDD, et les sous-modules d'égalisation d'usure (*WearLeveling*) et de ramassage de miettes (*GarbageCollector*) sont propres aux pilotes de SSD. Chaque instance de ces sous-modules sera appelée soit de manière synchrone, soit de manière asynchrone. Il est également possible d'avoir, pour un même sous-module, un comportement synchrone et un comportement asynchrone distincts.

- Allocateur

Le sous-module d'allocation est appelé par le pilote de disque pour traduire les adresses logiques en adresses physiques. Les allocateurs fournis avec OGSSim utilisent l'adressage LBA décrit dans le chapitre 1.

Dans le cas d'un algorithme plus avancé, ou couplé avec des mécanismes de gestion permanente, l'allocateur va devoir pour chaque nouvelle écriture, trouver un nouvel emplacement pour les données. Une fois l'allocation faite, la nouvelle adresse physique peut être insérée dans une table de traduction propre au pilote de disque. Cette table sera alors accessible par tous les sous-modules.

- Sonde

La sonde est le sous-module chargé de la prise d'informations concernant l'utilisation du disque durant la simulation, et de la mise à jour de son état. En fonction des besoins requis par les sous-modules de gestion permanente, elle peut par exemple tenir le compte du nombre d'accès en lecture ou en écriture, le nombre de pages contenant des données invalides ou encore le nombre d'effacements des blocs d'un SSD. L'appel au mécanisme de mise à jour de la sonde se fait une fois la traduction de l'adresse et/ou l'allocation de la page faite.

- Gestion permanente

On compte dans la version actuelle d'OGSSim trois sous-modules de gestion permanente :

1. Défragmentation pour les HDDs – La défragmentation est un mécanisme rassemblant les données éparpillées sur le disque pour qu'elles soient contigües. Ceci permet d'améliorer les performances du disque lors de la lecture de données, mais également lors de l'écriture de par le rassemblement des secteurs vierges. Les appels à la fonction principale du sous-module sont faits de manière asynchrone, définis par l'utilisateur.

2. Égalisation d'usure pour les SSDs – L'égalisation d'usure a pour but l'extension de la longévité du disque en équilibrant les niveaux d'usure de ses blocs. Deux types d'appel sont disponibles pour ce sous-module, l'appel asynchrone et l'appel synchrone. L'appel asynchrone est lié à une fonction de test, pour déterminer si l'appel doit se faire ou non.
3. Ramassage de miettes pour les SSDs – Le ramassage de miettes consiste en un recyclage des blocs possédant une majorité de pages invalides, généralement suite à des réécritures. Le mécanisme va alors copier les pages valides sur un autre bloc, puis effacer le bloc contenant les pages invalides pour qu'il puisse être de nouveau utilisé. Comme pour l'égalisation d'usure, le sous-module de ramassage de miettes peut être configuré avec un appel asynchrone et un appel synchrone.

Deux pilotes de disque sont donc utilisables : un pilote pour les disques magnétiques (HDD), et un pour les disques Flash (SSD). Dans une version ultérieure, d'autres pilotes pourront être ajoutés pour des nouveaux supports de stockage.

Une fois le traitement de la requête terminé, le pilote de disque communique au module d'exécution l'indice de la requête utilisateur traitée et les indices des requêtes générées par les sous-modules de gestion permanente.

# 3 Chaîne de calcul

## 3.1 Exécution

Le module d'exécution (*Execution*) est le noyau de calcul d'OGSSim. Après la réception, de la part des pilotes de disques, des requêtes décomposées, il va calculer pour chacune d'entre elles leur temps de réponse. D'autres métriques de performance sont également calculées au fur et à mesure de la simulation, telle le taux d'utilisation des disques. Le temps de réponse $T_{rep}$ se calcule comme suit :

$$T_{rep} = T_{srv} + T_{tsf} + T_{att} \tag{3.1}$$

avec $T_{srv}, T_{tsf}, T_{att}$ respectivement les temps de service, de transfert et d'attente

Les calculs du temps de service et du temps de transfert sont effectués par des sous-modules de l'exécution. Dans ce qui suit, nous allons détailler ces sous-modules. Nous noterons $s$ la taille des données ciblées par la requête que nous cherchons à traiter.

- Temps de service

Le temps de service est équivalent au temps que prend le disque pour lire/écrire des données. Le calcul de ce temps est dépendant de la technologie utilisée, puis des caractéristiques physiques du support. Il nous faut donc séparer ici le cas du HDD de celui du SSD.

Pour les disques magnétiques, le service d'une requête est composé de trois phases : la recherche de la piste, la rotation du disque et le transfert des données. Les temps correspondant à ces trois phases sont respectivement appelés temps de recherche $T_{rec}$, temps rotationnel $T_{rot}$ et temps de transfert interne $T_{tsi}$ :

$$T_{srv} = T_{rec} + T_{rot} + T_{tsi} \tag{3.2}$$

La recherche de la piste ciblée consiste en un déplacement latéral de la tête de lecture le long du plateau. Ce temps de recherche est calculé selon la formule de Lee (Lee, 1993; Chen and Lee, 1995), décrite dans l'algorithme 3.

**Algorithme 3 :** Formule de Lee pour le calcul du temps de recherche dans les disques magnétiques

**Entrées :** $p$ – piste précédente
$p'$ – piste courante
$n_{cyl}$ – nombre de cylindres
$t_{min}$ – temps de recherche minimal
$t_{moy}$ – temps de recherche moyen
$t_{max}$ – temps de recherche maximal
**Sorties :** $T_{rec}$ – temps de recherche

1 $d \leftarrow \text{abs}\ (p - p')$
2 **si** $d = 0$ **alors**
3 | $T_{rec} = 0$
4 **sinon**
5 | $a \leftarrow (-10 \times t_{min} + 15 \times t_{moy} - 5 \times t_{max})/(3 \times \text{ sqrt } (n_{cyl}))$
6 | $b \leftarrow (7 \times t_{min} - 15 \times t_{moy} + 8 \times t_{max})/(3 \times n_{cyl})$
7 | $T_{rec} \leftarrow a \times \text{ sqrt } (d) + b \times (d - 1) + t_{min}$
8 **fin**

Le temps rotationnel représente le temps que met la tête de lecture à parcourir la piste pour trouver le secteur ciblé puis lire les données. Il est calculé à l'aide du nombre de rotations par minute et du nombre de secteurs par piste, avec $n_{spp}$ le nombre de secteurs par piste et $n_{rpm}$ le nombre de rotations par minute qu'effectue la tête de lecture/écriture :

$$T_{rot} = \frac{n_{spp}/2 + s}{n_{spp}} \times \frac{60}{n_{rpm}} \tag{3.3}$$

Le facteur $n_{spp}/2$ représente le temps que met la tête de lecture à se positionner sur le premier secteur. Nous avons choisi de représenter ce temps par une valeur moyenne car nous ne pouvons assurer la position de la tête à un instant donné, et donc calculer le temps exact de rotation.

Le temps de transfert interne correspond au temps que mettent les données à être transférées du disque vers le buffer interne de ce dernier. Ce temps est calculé comme suit, avec $bp_i$ la bande-passante interne du disque :

$$T_{tsi} = taille \times bp_i \tag{3.4}$$

Pour le calcul du temps de service sur les SSDs, il est décomposé en deux parties : le temps d'accès $T_{acc}$ et le temps de transfert interne $T_{tsi}$. Le calcul est donc le suivant :

$$T_{srv} = T_{acc} + T_{tsi} \tag{3.5}$$

Le temps de transfert interne est calculé de la même manière que pour les HDDs, en fonction de la bande-passante interne du disque. Le calcul du temps d'accès dépend du type d'accès initié par le traitement de la requête. Si l'accès aux données est séquentiel, alors la formule est la suivante, avec $t_{seq}$ le temps d'accès séquentiel à une page :

$$T_{acc} = taille \times t_{seq} \tag{3.6}$$

Dans le cas où l'accès aux données est aléatoire, la formule devient, avec $t_{alea}$ le temps d'accès aléatoire à une page :

$$T_{acc} = t_{alea} + (taille - 1) \times t_{seq} \tag{3.7}$$

- Temps de transfert

Le modèle de calcul pour le temps de transfert est basé sur la bande-passante maximale du bus utilisé. L'équation utilisée pour ce calcul est la suivante, avec $bp$ la bande-passante du bus :

$$T_{tsf} = s \times bp \tag{3.8}$$

- Temps d'attente

Il existe dans notre cas deux types de temps d'attente : le temps d'attente avant la prise en charge par le disque, une fois la requête arrivée à ce dernier, et le temps d'attente avant le transfert des données sur les bus. Leur calcul dépend du disque ciblé, des bus empruntés, et de l'ordre dans lequel les requêtes sont traitées. Or, notre simulateur est asynchrone, ce qui sous-entend que l'ordre des requêtes utilisateur tel que donné dans le fichier de trace n'est pas toujours respecté. Effectivement, il arrive souvent, voire dans la quasi totalité des cas, que les requêtes ciblant un disque $\alpha$ soient plus rapidement traitées par la chaîne de décomposition que celles ciblant un disque $\beta$. Ceci dépend du type de configurations auxquelles appartiennent ces disques, et de la quantité de requêtes qu'ils doivent respectivement traiter.

Pour recalculer le bon ordre d'arrivée des requêtes, il y a donc besoin d'utiliser des mécanismes de synchronisation. Ces mécanismes et le calcul du temps d'attente des requêtes est expliquée dans la section 4.

## 3.2 Visualisation

Le module de visualisation (*Visualization*) est chargé de la création des graphiques et diagrammes demandés par l'utilisateur en début de simulation. Les demandes de graphes sont renseignées dans le fichier de configuration d'OGSSim, dans la balise `<visualization>` réservée au module. Un exemple de demande de graphe est montré sur le code 3.4.

```
1    <visualization>
2        <graph type="devbehavior" volume="all" output="gr1.png" />
3        <graph type="reqpercentile" size="20" output="gr2.png" />
4    </visualization>
```

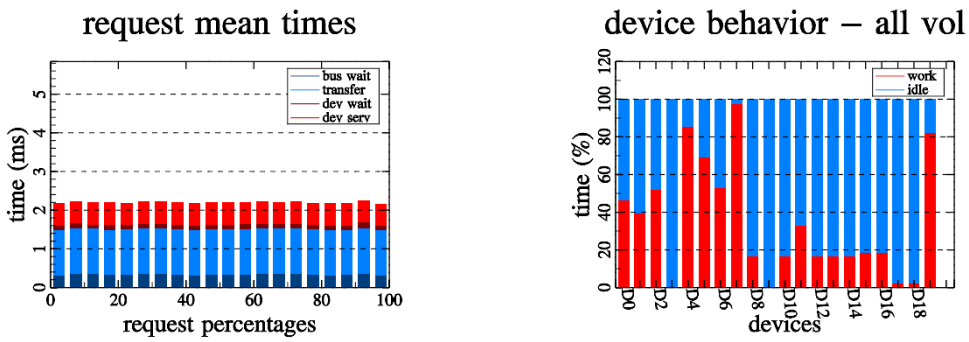
Code 3.4 – Exemple de demande de graphe de visualisation

A l'aide des informations recueillies durant la simulation dans les différentes structures liées aux requêtes et à l'architecture simulée, une fonction de création fait le tri des résultats utiles pour fournir le graphe demandé. L'interface de la classe de création de graphe est présentée dans le code 3.5. Elle possède un pointeur sur les structures de données, `m_requests` et `m_architecture`, ainsi que la fonction génératrice de graphe `makeGraph()`.

```
class GraphGeneration {
public:
    virtual ~GraphGeneration () {  };
    virtual void makeGraph () = 0;

protected:
    GraphGeneration (
        RequestArray            * requests,
        Architecture            * architecture,
        const std::string       outputFilename);

    std::string                 m_outputFile;

    RequestArray                * m_requests;
    Architecture                * m_architecture;
};
```

Code 3.5 – Interface de création de graphe

La figure 3.3 montre un exemple de graphes de sortie d'OGSSim. Le premier (3.3a) montre l'évolution du temps d'attente et de service moyen des requêtes durant la simulation. Le second (3.3b) donne pour chaque disque du système, la part de temps durant laquelle il est en service, en train de traîter une réquête, et celle durant laquelle il est inactif.




(a) Evolution du temps moyen de réponse

(b) Part d'utilisation des disques

Figure 3.3 – Exemples de graphes de visualisation

# 4 Validation d'OGSSim

Nous considérons pour première étape de la validation d'OGSSim, la validation au niveau du composant unitaire, soit le disque.

## 4.1 Contexte d'expérimentation

Nos expérimentations suivent le protocole suivant :

a) Installation du disque sur la machine de test.

b) Exécution sur le disque d'une série de *benchmarks* en utilisant Vdbench (Oracle, 2016).

c) Récupération des paramètres du disque pour la création du fichier de paramétrage utilisé dans OGSSim.

d) Exécution avec OGSSim de la même série de *benchmarks* que celle exécutée sur le disque testé.

e) Comparaison des deux séries de résultats.

### a) Installation du disque sur la machine de test

Nous considérons pour ces expérimentations les quatre disques suivants (2 SSDs et 2 HDDs), dont les paramètres sont décrits dans le tableau 3.2 :

— SSD Transcend 370 (Transcend, 2015) ;

— SSD Samsung 850 EVO (Samsung, 2016b) ;

— HDD Seagate ST 500 (Seagate, 2011) ;

— HDD HGST Travelstar Z7K500 (HGST, 2016).

| Capacité | 256 Go |
|---|---|
| Taille de page | 4ko |
| Lecture séquentielle | 560 Mo/s |
| Ecriture séquentielle | 320 Mo/s |
| Lecture aléatoire (4Ko) | 70000 IOPS |
| Ecriture aléatoire (4Ko) | 70000 IOPS |

(a) Transcend 370

| Capacité | 256 Go |
|---|---|
| Taille de page | 16ko |
| Lecture séquentielle | 540 Mo/s |
| Ecriture séquentielle | 520 Mo/s |
| Lecture aléatoire (4Ko) | 97000 IOPS |
| Ecriture aléatoire (4Ko) | 88000 IOPS |

(b) Samsung 850 EVO

| Capacité | 500 Go |
|---|---|
| Taille de secteur | 512o |
| Recherche min. (R/W) | 1.0 / 1.2 ms |
| Recherche moy. (R/W) | 11 / 12 ms |
| Rotation | 7200 RPM |
| Débit interne | 212 Mo/s |

(c) Seagate ST 500

| Capacité | 500 Go |
|---|---|
| Taille de secteur | 4ko |
| Recherche min. (R/W) | 1.0 / 1.1 ms |
| Recherche moy. (R/W) | 13 / 13 ms |
| Rotation | 7200 RPM |
| Débit interne | 171 Mo/s |

(d) HGST Travelstar Z7K500

TABLE 3.2 – Spécifications des disques utilisés pour la validation d'OGSSim

La machine utilisée pour les tests réels est un processeur octo-core Intel "Haswell" 3.6 GHz, couplé à 16 Go de Ram. Le système d'exploitation est Ubuntu 14.04 64-bit.

### b) Exécution avec Vdbench

L'envoi de requêtes au disque est effectué à l'aide de Vdbench (Oracle, 2016). Vdbench est un logiciel de *benchmarking* fonctionnant aussi bien avec les HDDs qu'avec les SSDs. Il peut être paramétré par ligne de commande ou par fichier d'entrée. Un exemple de fichier d'entrée est donné par le code 3.6. Le fichier est découpé en trois parties que nous pouvons identifier par le premier mot-clé de chaque ligne : `sd`, `wd` et `rd`.

```
sd=sd1,lun=/data/vdbench_file,size=8G
wd=wd1,sd=sd1,xfersize=512,rdpct=0,seekpct=0,openflags=o_direct
wd=wd2,sd=sd1,xfersize=1k,rdpct=0,seekpct=0,openflags=o_direct
wd=wd3,sd=sd1,xfersize=2k,rdpct=0,seekpct=0,openflags=o_direct
wd=wd4,sd=sd1,xfersize=4k,rdpct=0,seekpct=0,openflags=o_direct
wd=wd5,sd=sd1,xfersize=8k,rdpct=0,seekpct=0,openflags=o_direct
rd=run1,wd=wd1,iorate=max,elapsed=200,interval=10
rd=run2,wd=wd2,iorate=max,elapsed=200,interval=10
rd=run3,wd=wd3,iorate=max,elapsed=200,interval=10
rd=run4,wd=wd4,iorate=max,elapsed=200,interval=10
rd=run5,wd=wd5,iorate=max,elapsed=200,interval=10
```

Code 3.6 – Exemple de fichier d'entrée de Vdbench

La ligne `sd` (*source description*) permet d'indiquer le disque ciblé par l'exécution du *benchmark*. Dans le cas où le disque possède un système de fichiers, nous renseignons également un chemin et une taille de fichier tampon, sur lequel les requêtes seront envoyées. Ce qui est le cas pour le troisième disque, le Seagate ST 500, qui est utilisé par la machine de test comme espace de stockage. Ce fichier tampon est rempli de manière aléatoire.

La partie `wd` (*workload description*) détaille les types de traces créés pour l'exécution du *benchmark*. Chaque ligne décrit ici une trace référencée par un nom logique (`wd1`, `wd2`, etc.). Chaque trace est ensuite spécifiée avec divers paramètres. Pour nos tests, nous avons utilisés les paramètres suivants :

- `xfersize` : taille des requêtes envoyées ;
- `rdpct` : pourcentage de lecture ;
- `seekpct` : pourcentage de requêtes aléatoires ;
- `openflags` : paramètres d'ouverture du fichier.

Nous ouvrons le fichier avec le mode `o_direct` pour minimiser les effets de cache. En effet ce mode permet le transfert synchrone des données, afin qu'elles ciblent directement le disque.

Enfin, la description des tests avec `rd` (*run description*) permet d'indiquer le temps d'exécution des traces, le débit d'envoi de requêtes ainsi que l'intervalle de temps entre deux mesures.

Le tableau 3.3 recense les valeurs des paramètres du *benchmark* pour nos jeux de test.

| *Benchmark* | `xfersize` | `rdpct` | `seekpct` |
|---|---|---|---|
| Lecture séquentielle | 512 à 8M (HDD) | 100 | 0 |
| Lecture aléatoire | 1k à 128M (SSD) | 100 | 100 |
| | 512 à 8M (HDD) | | |
| Ecriture séquentielle | 512 à 8M (HDD) | 0 | 0 |
| Ecriture aléatoire | 1k à 128M (SSD) | 0 | 100 |
| | 512 à 8M (HDD) | | |

TABLE 3.3 – Paramètres des jeux de test

#### c) Paramétrage des fichiers de caractérisation des disques pour OGSSim

A partir des spécifications des disques, nous récupérons les paramètres concernant la géométrie (taille de page/secteur, nombre de secteurs par piste pour les HDDs, nombre de pages par bloc pour les SSDs, etc.) et la performance théorique annoncée par le constructeur (temps d'accès minimum, moyen et maximum au secteur pour les HDDs, débit en accès aléatoire et séquentiel pour les SSDs).

#### d) Exécution avec OGSSim

Nous simulons avec OGSSim, un système composé d'un seul disque, représenté par le fichier de caractérisation établi à l'étape précédente. La trace utilisée pour la simulation est un ensemble de 50k requêtes, dépendant de la taille de la requête, dont les dates d'arrivée sont égales. Ceci pour stresser le système.

#### e) Comparaison entre le système réel et le système simulé

Une fois les deux séries de résultat obtenues, nous effectuons une comparaison à l'aide de la bande-passante. D'autres métriques sont disponibles en sortie de Vdbench, telles le temps de réponse moyen de la requête ou le nombre d'IOPS.

## 4.2 Validation contre les SSDs

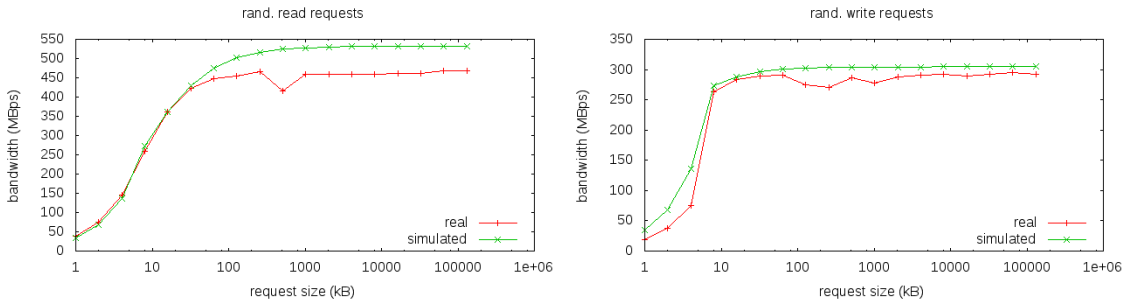
La première validation effectuée avec OGSSim est celle ciblant les disques Flash. Nous avons mené nos expérimentations avec deux disques de constructeurs différents : le Transcend 370 et le Samsung 850 EVO. Les évaluations sont ici faites avec des traces contenant soit des lectures aléatoires, soit des écritures aléatoires.

#### Transcend 370

Les résultats de la comparaison sont présentés sur la figure 3.4. On représente la bande-passante en fonction de la taille des requêtes de la trace, entre 1Ko et 128Mo. Dans l'ensemble, les courbes réelles et simulées possèdent le même profil et des valeurs très proches.

Pour ce qui est de la lecture aléatoire (figure 3.4a), la pente (entre 1ko et 32ko) des deux courbes est quasiment la même. Le plateau lui (entre 32ko et 128Mo) possède un écart d'en moyenne 20%, avec la bande-passante simulée supérieure à la bande-passante mesurée (réelle).

Les résultats du système simulé avec l'écriture aléatoire (figure 3.4b) présentent un écart plus marqué au niveau de la pente, avec un maximum pour des requêtes de 4Ko, où la bande-passante simulée est supérieure de 70%. Par contre, les tendances des courbes pour le plateau sont identiques, avec au plus une augmentation de 10% pour le système simulé. Ces écarts s'expliquent par la baisse de performance des disques réels dû à l'exécution répétée des *benchmarks*. Plus il y a d'écritures sur le disque, plus il y a de blocs invalides et plus les performances se dégradent à cause du mécanisme d'égalisation d'usure et du ramassage de miettes.




(a) Lecture aléatoire (b) Ecriture aléatoire

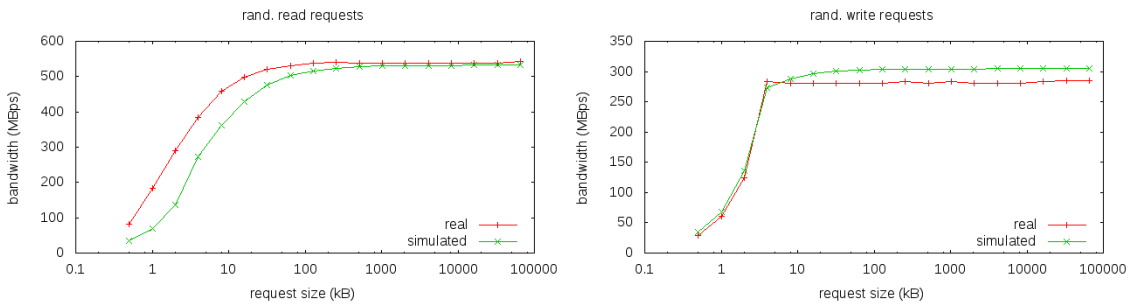
FIGURE 3.4 – Validation contre le Transcend 370

### Samsung 850 EVO

Les courbes résultantes de la validation de ce disque, présentées sur la figure 3.5, montrent également des résultats très proches entre la bande-passante du système simulé et celle du système réel.

En lecture aléatoire (figure 3.5a), les courbes au niveau du plateau (à partir de 128ko) sont quasi confondues tandis que la pente (entre 512o et 128ko) accuse un écart maximal de 50 à 60%.

Pour l'écriture aléatoire (figure 3.5b), les pentes des courbes sont confondues. Le plateau de la courbe simulée est 8% supérieure à celui de la courbe réelle. La raison de cet écart est identique à celle évoquée pour le Transcend 370 : la dégradation des performances du disque réel suite à une utilisation intensive.




(a) Lecture aléatoire (b) Ecriture aléatoire

FIGURE 3.5 – Validation contre le Samsung EVO 850

L'expérimentation menée pour la validation du modèle de calcul pour les SSDs présente des résultats cohérents. Sur les deux disques testés, les tendances des courbes sont identiques, ce qui signifie que les phases de pente et de plateau sont délimitées par les mêmes bornes. Pour des requêtes supérieures à 4ko, l'écart de performance mesuré entre le système réel et le système simulé est entre 0 et 20%, dépendant du disque et du type d'opération.

## 4.3 Validation contre les HDDs

La seconde validation d'OGSSim cible les disques magnétiques. Les expérimentations ont été menées avec également deux disques de constructeur différent : le Seagate ST500 et le HGST Travelstar Z7K500.

### Seagate ST500

La série de graphes présentée sur la figure 3.6 montre les résultats des expérimentations de validation sur le disque pour le système réel et le système simulé. Nous avons ici deux courbes différentes pour le système simulé : une où le débit considéré pour le transfert interne des données est fixé à 212 Mo/s, valeur donnée par le constructeur, et l'autre où ce débit est à 151 Mo/s. Cette seconde valeur correspond au débit assuré par le constructeur. Cette valeur de débit interne correspond à la valeur convergente de la bande-passante du disque lorsque la taille de la requête croît, car les temps rotationnel et de recherche deviennent négligeables.

Nous distinguons pour la suite de la discussion, trois intervalles distincts : le plateau bas, entre 512o et 4ko pour les accès séquentiels et entre 512o et 64ko pour les accès aléatoires ; la pente, entre 4ko et 256ko pour les accès séquentiels et entre 64ko et 4Mo pour les accès aléatoires ; et le plateau haut au-dessus de 256ko pour les accès séquentiels et 4Mo pour les accès aléatoires.

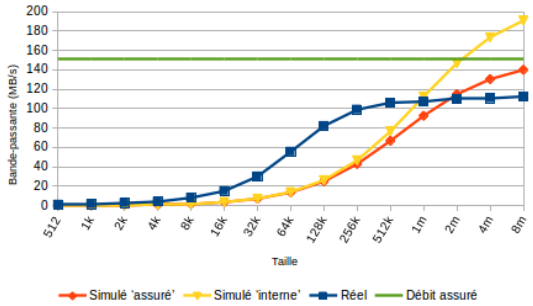
Nous pouvons observer pour les traces d'accès séquentiels (figure 3.6a et figure 3.6c) que l'allure des courbes est similaire, mais elles ne sont pas assez proches. Dans le cas des accès aléatoires (3.6b et 3.6d), les tendances des courbes sont plus proches.

Les résultats observés ne sont pas concluants, ceci étant en majeure partie dû à un manque d'informations sur la géométrie interne du disque.

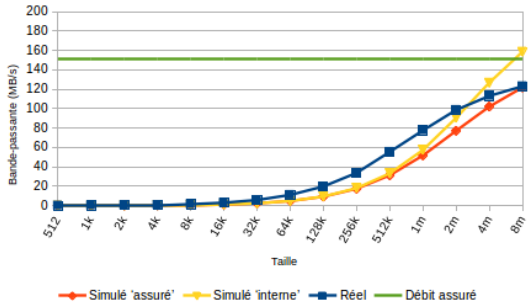
### HGST Travelstar Z7K500

Les courbes présentées sur la figure 3.7 concernent la validation d'OGSSim contre le disque HGST Travelstar Z7K500. On observe que les courbes des accès séquentiels (figures 3.7a et 3.7c) ne suivent pas la même tendance. En revanche, les courbes des accès aléatoires (figures 3.7b et 3.7d) ont des tendances similaires pour des tailles de requêtes allant jusqu'à 2Mo. L'écart maximal constaté pour cet intervalle est de 40% pour les requêtes de 512ko.

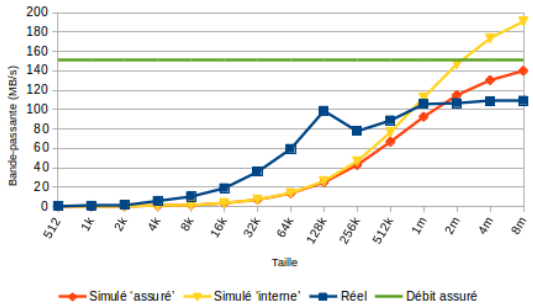
Nous poursuivons notre validation contre les disques magnétiques. Les tendances des courbes des accès aléatoires sont semblables pour les deux disques, ce n'est pas le cas pour les accès séquentiels. La piste de recherche actuelle est portée sur l'intégration de la gestion du buffer interne au disque dans notre modèle de calcul.



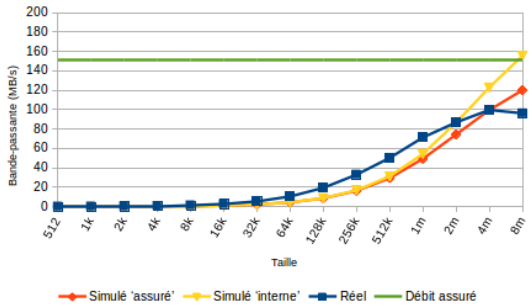

(a) Lecture séquentielle




(b) Lecture aléatoire

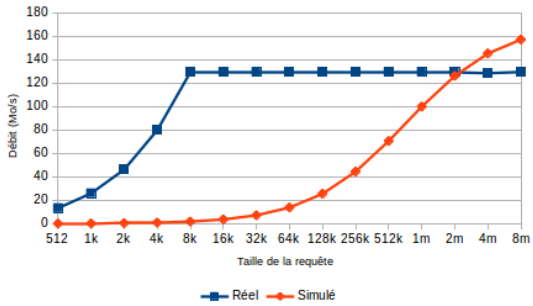



(c) Ecriture séquentielle

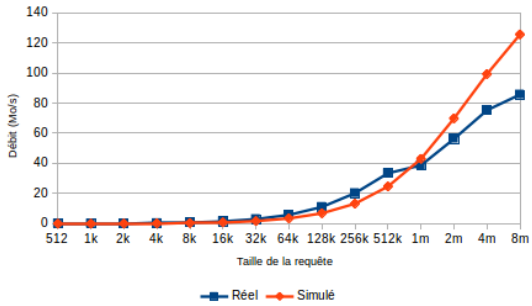



(d) Ecriture aléatoire

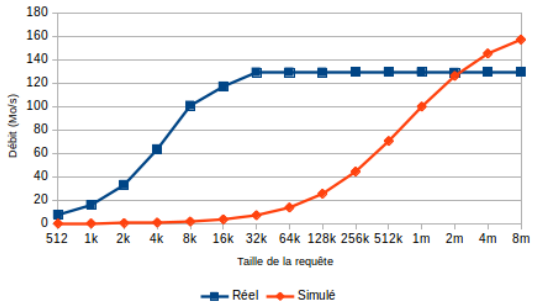
FIGURE 3.6 – Validation contre le Seagate ST500



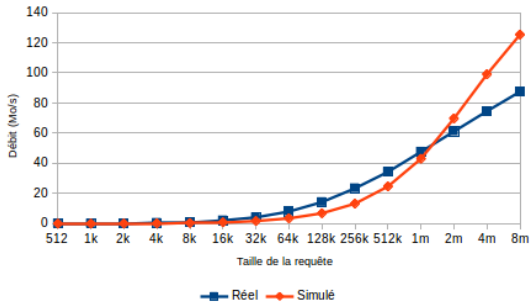

(a) Lecture séquentielle




(b) Lecture aléatoire




(c) Ecriture séquentielle




(d) Ecriture aléatoire

FIGURE 3.7 – Validation contre le HGST Travelstar Z7K500

## 4.4 Validation contre les systèmes réels

Pour que la validation de l'outil soit complète, nous prévoyons de mener des expérimentations contre des systèmes réels (systèmes RAID, baies de stockage). Il nous faudra pour ceci avoir à notre disposition ce système réel et avoir pleine connaissance

de sa topologie, à savoir le paramétrage des disques, leur agencement et les connectiques utilisés, puis la reporter dans OGSSim. Cette expérimentation nous permettra également de vérifier notre gestion des bus externes. Cette étape de validation devrait se faire dans un futur proche.

# 5 Conclusion et perspectives

L'implémentation des modules d'OGSSim offre donc la possibilité à n'importe quel utilisateur d'ajouter, par le biais d'interfaces simples d'utilisation, ses propres modèles de calcul et autres algorithmes.

A l'heure actuelle, OGSSim fonctionne uniquement en environnement multi-threadé, dans un contexte de mémoire partagée. Une évolution future de notre logiciel serait de le porter dans un environnement parallèle multi-noeud, en mémoire distribuée. Cette modification nécessitera l'ajout d'une interface pour la gestion des structures de données, afin de pouvoir basculer facilement du mode mémoire partagée au mode mémoire distribuée en fonction du système utilisé pour la simulation.

La phase de validation contre les SSDs a été complétée et satisfaisante à l'état actuel, et celle contre les HDDs nécessite plus d'investigation et de test. Il est également prévu de confronter notre outil de simulation à des systèmes de stockages, d'une ou de plusieurs configurations.

OGSSim a fait l'objet de deux publications dans des conférences internationales. La première à SIMUtools 2015 (Gougeaud et al., 2015a), présentant l'outil dans une première version où seul le mode normal d'exécution était disponible, et une seconde à SPECTS 2016 (Gougeaud et al., 2016b), montrant l'ajout du mode défaillant à OGSSim. D'autres sont en cours pourtant tout spécialement sur l'implémentation de la communication et de la synchronisation au sein d'OGSSim.

# Chapitre 4

# Communication et synchronisation



*L'exploitation du parallélisme dans une application permet de réduire son temps d'exécution par un traitement simultané des instructions. Il existe alors deux types de parallélisme. Le premier, aussi appelé scatter and gather, est le parallélisme de données. Le processus consiste en une découpe du jeu de données à l'initialisation, puis en une répartition de ces données sur les threads exécutants. Chaque thread va exécuter, à quelques conditions près, la même suite d'instructions. A la terminaison du programme, les résultats sont alors réunis et transmis à l'utilisateur. Le second type de parallélisme est le parallélisme de tâches, où chaque thread va exécuter une suite différente d'instructions. Dans ces programmes, des moyens de communication sont utilisés durant l'exécution pour la transition de données.*

*OGSSim utilise un parallélisme de tâches, et par conséquent la communication occupe une place très importante au sein de l'application. En plus de gérer le transfert et/ou le partage de données durant l'exécution, elle peut également servir à garantir une synchronisation entre les différents threads.*

*Dans ce chapitre, nous allons présenter les différents mécanismes de communication utilisés, de par l'intégration des ZeroMQ, ainsi que de la synchronisation mise en place. Nous soulignerons plus particulièrement quatre problèmes survenus lors de l'implémentation de notre outil de simulation et détaillerons la manière dont ils ont été résolus. Nous terminerons la partie décrivant OGSSim par quelques mesures de performances de notre outil de simulation.*

# 1 Utilisation des ZeroMQ

OGSSim est un logiciel modulaire, où chaque module implémente une phase de l'exécution d'une requête d'entrée/sortie, et chaque module est traité par un thread. Toutes les communications inter-thread sont gérées par des ZeroMQ (iMatix Corporation, 2016). Comme décrit dans le chapitre 2, chaque canal de communication nécessite d'être détaillé dans le fichier de configuration du simulateur. Cette description est composée des 4 paramètres suivants :

— l'interlocuteur – le module communicant avec le 'demandeur' ;

— le protocole – protocole de communication utilisé, ici tcp ;

— l'adresse de destination – adresse de l'interlocuteur, ici en `localhost` ;

— le port de communication – port du canal de communication.

Le modèle de communication choisi dans OGSSim est celui du producteur/consommateur. Un ou plusieurs producteurs, aussi appelés expéditeurs, envoient des messages dans le canal de communication, qui seront récupérés par un consommateur ou destinataire. Nous allons décrire le fonctionnement de ce modèle, en séparant le cas où il n'y a qu'un producteur de celui où il y en a plusieurs.

La figure 4.1 montre un canal pour deux interlocuteurs. On suppose donc ici un expéditeur souhaitant remettre 5 messages, numérotés de 1 à 5, à son interlocuteur, le destinataire. La situation (a) décrit la phase initiale, où les 5 messages sont possédés par l'expéditeur. Dans la situation (b), l'expéditeur a transmis par le biais du canal de communication les deux premiers messages 1 et 2. Dans la situation (c), deux autres messages ont été envoyés par l'expéditeur, 3 et 4, et le message 1 a été reçu par le destinataire. Enfin, dans la dernière situation, la (d), tous les messages ont été reçus par le destinataire. Cet exemple montre que les messages sont reçus dans l'ordre où ils sont envoyés par l'expéditeur.

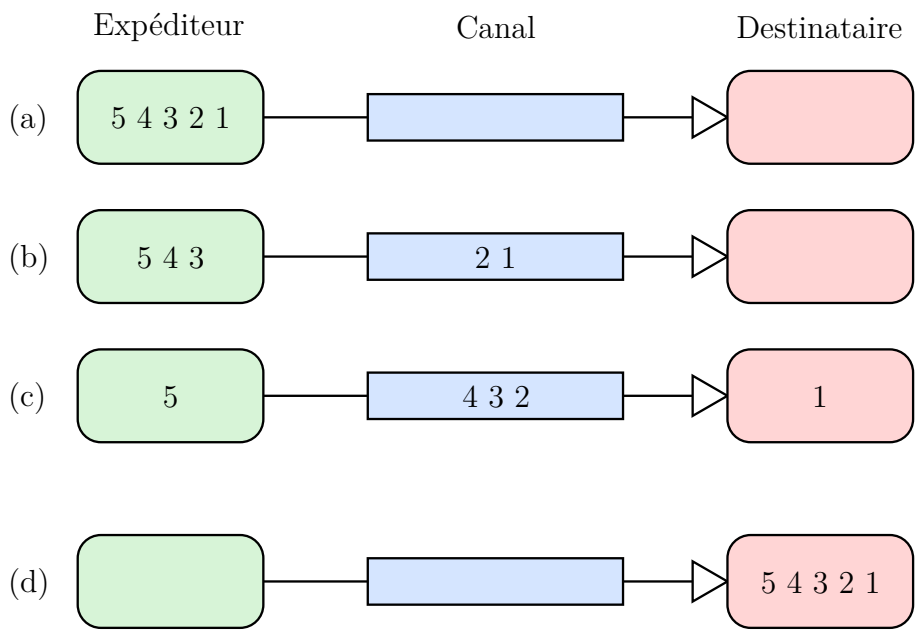



FIGURE 4.1 – ZeroMQ 1-1

L'autre type de communication est un canal à $n$ expéditeurs et un destinataire. Il est représenté sur la figure 4.2. L'exemple montre trois expéditeurs souhaitant envoyer respectivement les messages 1 à 3, 4 et 5 et 6 à 8 par l'unique canal de communication, comme décrit dans la situation initiale (a). Aux premiers messages envoyés (b), le canal de communication se remplit avec les messages 1, 4, 6 et 7. On observe ici un cas d'asynchronisme, où les messages ne sont plus dans l'ordre initial (le message 4 est envoyé dans le canal avant le message 2). Dans la situation (c), les premiers messages sont récupérés par le destinataire, à savoir les messages numéro 1, 4 et 6. De même, les expéditeurs continuent d'envoyer leurs messages dans le canal de communication, soient les messages 8, 2 et 3, toujours en perdant la notion d'ordre dans le temps. Dans la situation finale (d), tous les messages ont été reçus par le destinataire. On peut déduire à partir de l'ordre de réception, les deux faits suivants :

— à cause de l'asynchronisme généré à l'envoi des messages, l'ordre de réception est différent de la numérotation initiale ;

— l'ordre des messages envoyés par un seul expéditeur est respecté lors de la réception, par exemple, les messages 1 à 3 sont récupérés dans cet ordre par le destinataire, le message 1 avant le 2, et le 2 avant le 3.

L'utilisation de la variante du modèle de communication avec $n$ expéditeurs est nécessaire au regroupement des données à la fin de la chaîne de décomposition. L'asynchronisme généré par l'ensemble de la chaîne est donc dû :

1. au traitement parallèle de la décomposition, à chaque pilote son propre thread ;
2. au comportement imprédictible du canal de communication.

Pour la suite, nous allons définir les deux fonctions suivantes, liées au canal de communication :

— `envoi (canal, message)` – envoi d'un message par le canal de communication donné en paramètre ;

— `reception (canal, message)` – réception d'un message par le canal de communication donné en paramètre.

# 2 Synchronisation des créations des sous-requêtes

Durant l'étape de décomposition des requêtes, lors d'une simulation, le pilote de volume est amené à créer des requêtes intermédiaires, appelées sous-requêtes. Ceci est d'autant plus important dans le cas où la requête utilisateur cible plusieurs disques ou a besoin de générer une pre-lecture des données (comme dans un RAID-5).

Les sous-requêtes sont stockées dans un espace mémoire de taille fixe. Au lancement de la simulation, le module d'extraction de trace, chargé de l'allocation de cet espace mémoire, récupère dans le fichier de configuration d'OGSSim, l'information renseignant sur le nombre de sous-requêtes pouvant être contenues dans l'espace mémoire à l'aide de la balise `<subreq>`.

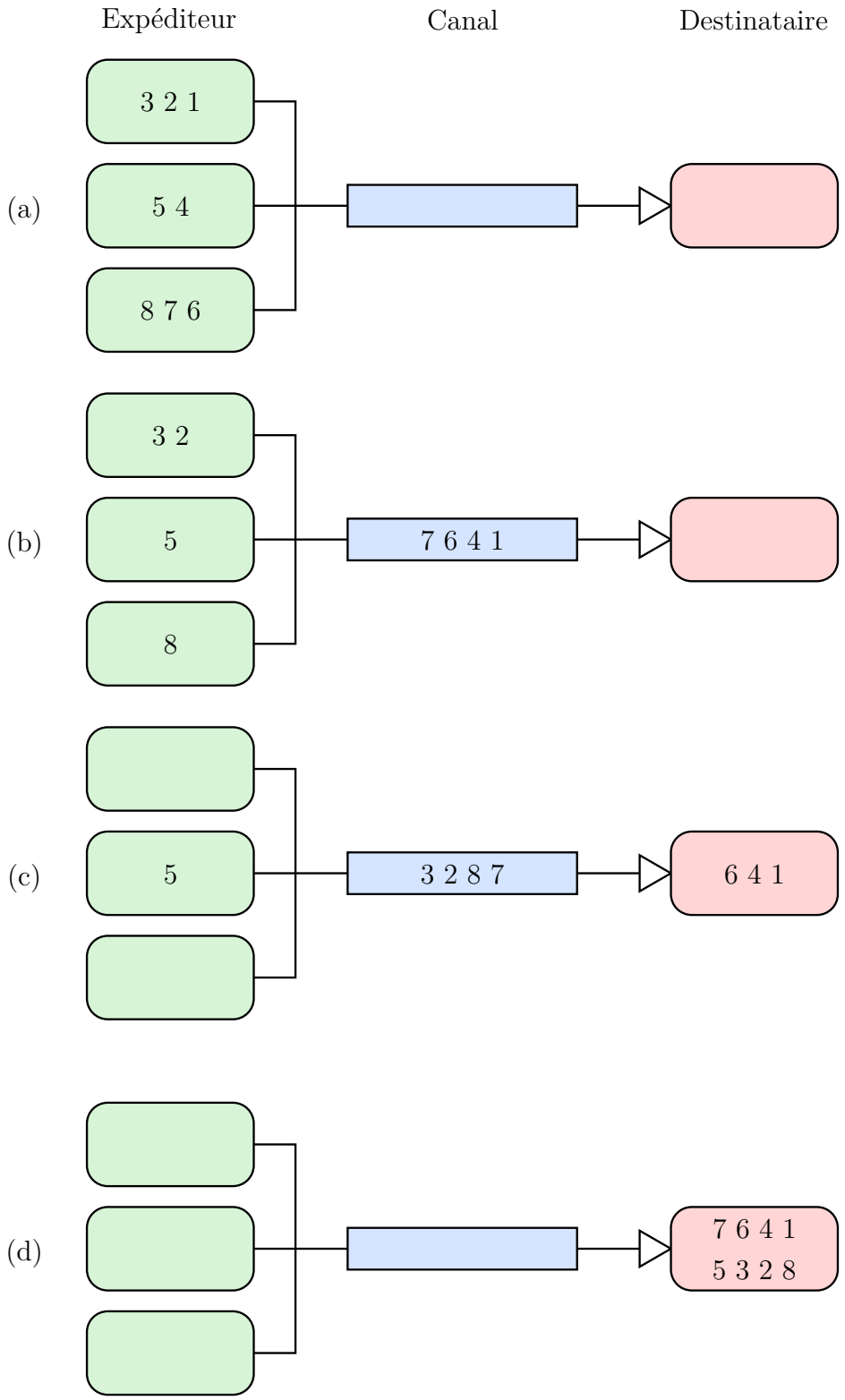



FIGURE 4.2 – ZeroMQ n-1

Chaque sous-requête possède un champ, représenté par un booléen, indiquant son état :

— vrai – la sous-requête est actuellement en cours de traitement ;

— faux – soit l'emplacement n'est pas utilisé, soit la sous-requête correspondante a déjà été traitée.

Lors de la création des sous-requêtes, le pilote de volume va parcourir le tableau de sous-requêtes jusqu'à trouver un emplacement dont l'état est à faux. Une fois trouvé, le pilote va utiliser cet emplacement pour y mettre les informations de la sous-requête créée, puis envoyer l'indice de la sous-requête au pilote de disque.

L'étape de création de sous-requêtes nécessite l'utilisation d'un verrou sur le tableau, pour éviter que deux pilotes ne sélectionnent le même emplacement disponible au même moment. Un verrou, ou exclusion mutuelle, est un mécanisme de synchronisation assurant que des ressources partagées ne soient pas accédées au même instant par deux processus différents. Un verrou s'utilise à l'aide des deux actions suivantes :

1. `prendre()` : Entrer dans l'exclusion mutuelle, pour avoir accès au tableau de sous-requêtes partagé. Si un autre pilote utilise déjà le verrou, alors ce pilote se met en attente de libération du verrou.
2. `relâcher()` : Sortir de l'exclusion mutuelle, en libérant le verrou.

Le comportement du pilote de volume lors de la recherche de l'emplacement est donné dans l'algorithme 4, selon lequel il va chercher de manière exclusive un emplacement disponible. L'indice $i_{prec}$ est utilisé pour éviter de toujours rechercher parmi les premiers éléments de la table, et éviter les emplacements récemment utilisés pour de nouvelles sous-requêtes.

**Algorithme 4 :** Recherche d'emplacement disponible dans le tableau de sous-requêtes

**Entrées :** $r$ – requête
$T$ – tableau des sous-requêtes
$n$ – taille de $T$
$i_{prec}$ – indice de la dernière sous-requête créée
$v$ – verrou sur $T$

**Sorties :** $i$ – indice de la sous-requête créée

```
1  prendre (v)
2  i ← i_prec
3  tant que T[i].etat est vrai faire
4  |   i ← i + 1
5  |   si i ≥ n alors
6  |   |   i ← 0
7  |   fin
8  fin
9  T[i].etat ← vrai
10 relacher (v)
```

Avec l'asynchronisme généré par le multi-threading et les communications inter-modules, il devient possible qu'au moment où une requête est décomposée par un pilote de volume, plusieurs sous-requêtes postérieures ont déjà été reçues par le module d'exécution. Dans le cas où ce nombre de sous-requêtes postérieures dépasse la taille du tableau alloué avant le lancement de la simulation, alors le pilote de volume chercherait indéfiniment un emplacement disponible dans ce même tableau. La raison réside dans le fait que pour qu'une sous-requête soit traitée par le module d'exécution, il faut calculer entre autre son temps d'attente, qui dépend de celui des requêtes antérieures.

La figure 4.3 présente la dépendance entre les requêtes, et ce besoin de synchronisation. Supposons que le module d'exécution souhaite calculer le temps de réponse de la requête $i$, et que toutes les requêtes antérieures ont été traitées, sauf la requête $(i-1)$, comme montré dans la situation (a). Les deux requêtes sont à destination de disques différents, mais empruntent les mêmes bus. Le temps de réponse théorique de la requête $i$ consisterait en la somme du temps de transfert de $t_5$ à $t_7$ puis du temps de service entre $t_7$ et $t_{11}$. Seulement, la requête $(i-1)$ étant antérieure, son temps de réponse peut influer sur celui de la requête $i$. A la réception de la requête $(i-1)$ dans la situation (b), le module d'exécution calcule son temps de transfert, allant de $t_4$ à $t_6$, puis son temps de service de $t_6$ à $t_{10}$. Les deux requêtes ne ciblant pas le même disque, les services des deux requêtes peuvent se chevaucher. Ce qui n'est pas le cas des transferts. La requête $i$ doit donc attendre la fin du transfert de la requête $(i-1)$ avant de pouvoir démarrer son traitement, ce qui génère un temps d'attente de $t_5$ à $t_6$. Le temps de réponse de la requête $i$ passe alors de 6 à 7 unités de temps.

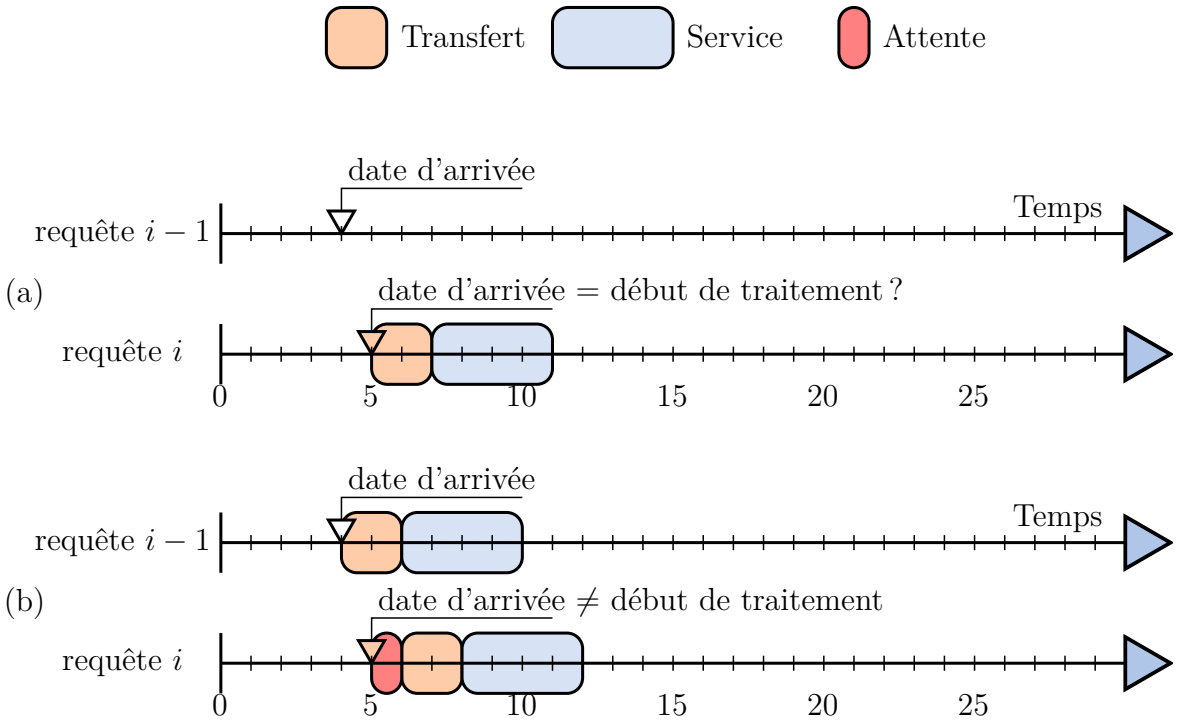



FIGURE 4.3 – Dépendance entre les requêtes pour le calcul du temps de réponse

Durant les premiers tests effectués, le phénomène d'inter-blocage (*deadlock*) bloquant alors l'exécution du programme était possible. Dans ce cas, le pilote de volume attendant une libération d'un emplacement du tableau par le module d'exécution, exécutait en boucle les instructions de l'algorithme 4, tandis que le module d'exécution

attendait la réception de nouvelles sous-requêtes à traiter. L'ordonnanceur donnant la priorité aux tâches utilisant le processeur, le module d'exécution n'était jamais sélectionné pour exécuter ses instructions.

La solution adoptée pour gérer ce phénomène est d'utiliser des canaux de communication afin de mettre en attente les pilotes de volume ne trouvant pas d'emplacement libre dans le tableau de sous-requêtes. La nouvelle version de l'algorithme de recherche est donnée dans l'algorithme 5.

**Algorithme 5 :** Recherche d'emplacement disponible dans le tableau de sous-requêtes (sans deadlocks)

**Entrées :** $r$ – requête
$T$ – tableau des sous-requêtes
$n$ – taille de $T$
$i_{prec}$ – indice de la dernière sous-requête créée
$v$ – verrou sur $T$
$c$ – canal de communication vers le module d'exécution
$c'$ – canal de communication depuis le module d'exécution

**Sorties :** $i$ – indice de la sous-requête créée

```
1  prendre (v)
2  i ← i_prec
3  tant que T[i].etat est vrai faire
4  |  i ← i + 1
5  |  si i ≥ n alors
6  |  |  i ← 0
7  |  fin
8  |  si i = i_prec alors
9  |  |  relacher (v)
10 |  |  envoi (c, "tableau sous-requetes plein")
11 |  |  reception (c', message)
12 |  |  verrou (v)
13 |  fin
14 fin
15 T[i].etat ← vrai
16 relacher (v)
```

Le comportement des modules lors de cette synchronisation est représenté par la figure 4.4. Ainsi, si le pilote de volume a parcouru l'ensemble des emplacements du tableau sans avoir trouvé d'espace libre (1), il va envoyer un message au module d'exécution afin de lui indiquer qu'il a besoin que l'espace soit libéré avant qu'il ne puisse reprendre la décomposition des requêtes. Ceci tout en ayant libéré le verrou, afin que d'autres pilotes de volume, s'exécutant en même temps, ne soient pas bloqués. A la réception de ce message, le module d'exécution associe au pilote de volume un compteur initialisé à un certain pourcentage (par exemple 20%) correspondant à la taille du tableau de sous-requêtes (2). A chaque fin de traitement de sous-requêtes, le compteur est décrémenté. Une fois le compteur à zéro, le module d'exécution envoie un signal au pilote de volume (3), pour lui signaler que suffisamment d'emplacements

ont été libérés et que le traitement peut reprendre. Le pilote de volume, à la réception du signal, reprend donc son exécution (4).

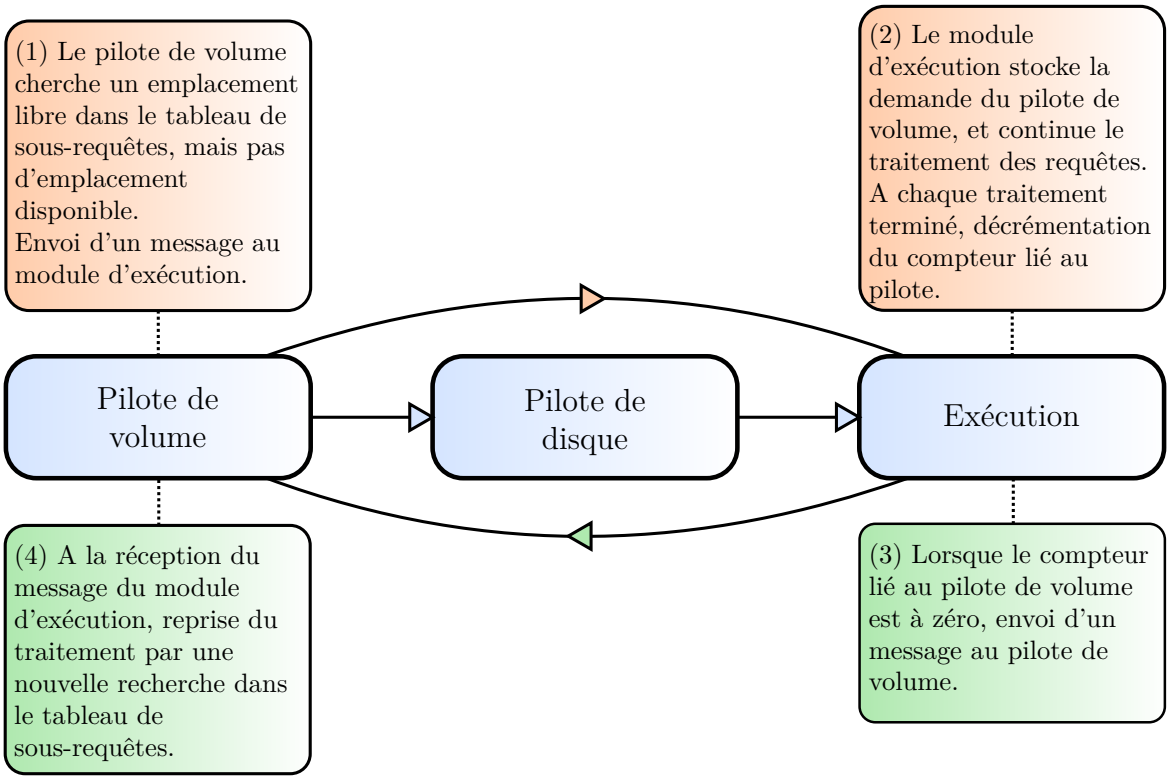



FIGURE 4.4 – Synchronisation pour la création des sous-requêtes

# 3 Synchronisation lors du traitement des pré-lectures

Lors de la décomposition de requêtes d'écriture à destination d'une configuration RAID-NP, utilisant alors des codes d'erreur comme la parité, une suite de sous-requêtes de pré-lecture sont générées. Ces sous-requêtes servent à récupérer les informations nécessaires à la mise à jour du code d'erreur, comme expliqué dans le chapitre 1.

Les requêtes décomposées en pre-lectures et écritures de code d'erreur obéissent à une contrainte simple qui est que l'écriture de code d'erreur ne peut être traitée par le disque qu'une fois l'ensemble des pré-lectures nécessaires à sa mise à jour effectuées. Le respect de cette contrainte, dans notre exécution asynchrone, ne peut se faire qu'à l'aide d'un nouveau mécanisme de synchronisation, utilisant, comme pour la création des requêtes, des canaux de communication.

Les acteurs de cette synchronisation sont les pilotes de disque et le module d'exécution. L'objectif de la synchronisation est donc de mettre en attente les pilotes de disque devant traiter des requêtes d'écriture de code d'erreur, jusqu'à la réception d'un message du module d'exécution, lui annonçant que les pré-lectures correspondantes ont été effectuées. Ce signal permettra alors de déclencher le calcul de la date à laquelle le service de l'écriture du code d'erreur peut démarrer, à partir du temps de réponse des sous-requêtes de pré-lecture. L'algorithme 6 décrit ce traitement. Le pilote de disque envoie au module d'exécution l'indice de la requête parente de la sous-requête afin

d'être réveillé uniquement si les pré-lectures issues de cette même requête parente sont terminées.

**Algorithme 6 :** Traitement d'une requête d'écriture de code d'erreur par un pilote de disque

**Entrées :** $r$ – requête
$c$ – canal de communication vers le module d'exécution
$c'$ – canal de communication depuis le module d'exécution

1 **si** *$r$ est une écriture de code d'erreur* **alors**
2 envoi ($c$, "attente pré-lecture $r$.parent")
3 reception($c'$, message)
4 **fin**
5 ...
6 traitement
7 ...
8 envoi ($c$, indice de $r$)

Du côté du module d'exécution, l'utilisation de structures de données supplémentaires sont nécessaires pour la mise en œuvre de cette synchronisation. En premier lieu, chaque demande de réveil de la part des pilotes de disque est stockée dans des files, indicées par l'estampille de la requête parente. Ainsi, lorsqu'une série de pré-lectures est traitée, le module consulte la file correspondante à l'indice de la requête parente, et envoie un message de réveil à chaque pilote de disque se trouvant dans la file. Le second ajout de données se situe au niveau de la structure de données des requêtes, par l'ajout d'un champ représentant le nombre de pré-lectures qui restent à traiter. Le compteur est initialisé par le pilote de volume, lors de la décomposition de la requête parente, et décrémenté par le module d'exécution à chaque fois qu'une pré-lecture est terminée.

A la réception d'une demande de réveil, le module d'exécution va avoir deux choix :

— soit les pré-lectures ne sont pas terminées (compteur non nul), et l'indice du pilote de disque est ajouté à la file correspondante ;
— soit les pré-lectures sont terminées (compteur nul), et le message de réveil est directement envoyé.

Une version de l'algorithme 6 moins coûteuse en communication peut alors être utilisée, grâce au champ de la requête, mis à jour par le module d'exécution. Cette version est décrite dans l'algorithme 7. Ici, il y a communication uniquement si le nombre de pré-lectures en attente de traitement est strictement positif.

La figure 4.5 résume le comportement des modules de pilote de disque et d'exécution.

# 4 Synchronisation pour l'ordonnancement dans les bus

La première version du modèle de calcul de bus ne prend pas en compte l'ordonnancement des requêtes au sein des bus de communication, ni même la congestion

**Algorithme 7 :** Traitement d'une requête d'écriture de code d'erreur par un pilote de disque version 2

**Entrées :** $r$ – requête
$c$ – canal de communication vers le module d'exécution
$c'$ – canal de communication depuis le module d'exécution

1 **si** *$r$ est une écriture de code d'erreur et $r.cptPreLectures \neq 0$* **alors**
2 envoi ($c$, "attente pré-lecture $r$.parent")
3 reception($c'$, message)
4 **fin**
5 ...
6 traitement
7 ...
8 envoi ($c$, indice de $r$)

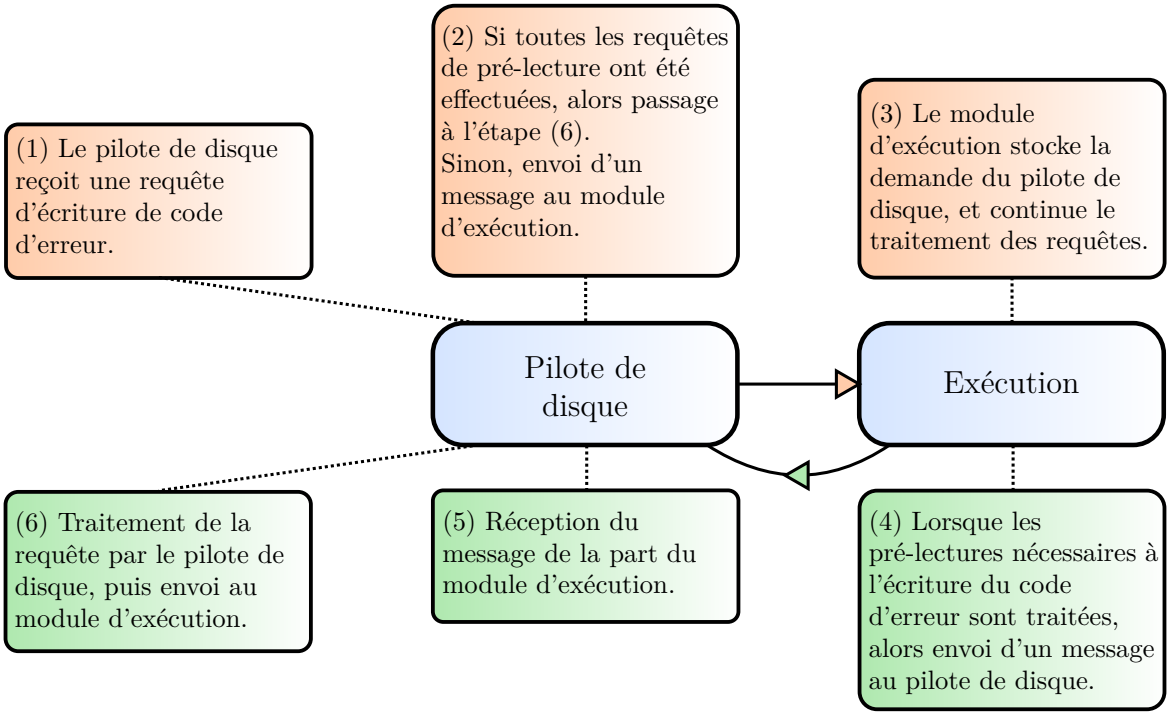



FIGURE 4.5 – Synchronisation pour le traitement des pré-lectures

de ces bus. L'ajout d'un modèle intégrant ces deux aspects nécessite de pouvoir traiter les requêtes selon leur ordre d'arrivée, fait non assuré, voire jamais avéré, de par l'asynchronisme de notre exécution dans OGSSim.

Deux possibilités d'implémentation sont alors possibles pour la synchronisation des données :

1. faire attendre les pilotes de disque avant chaque envoi de requête, que le module d'exécution ait traité toutes les requêtes antérieures ;
2. mettre dans des files d'attente les requêtes ne pouvant encore être traitées au niveau du module d'exécution.

La première implémentation revient à ignorer le parallélisme de traitement des requêtes par les pilotes de disque au profit de leur synchronisation. Effectivement, alors qu'un pilote de disque pourrait traiter les différentes requêtes envoyées par le pilote de volume, il est mis en attente le temps que le module d'exécution ait terminé le traitement des requêtes antérieures à celle possédée par le pilote de disque. C'est pour cette raison, et ce malgré l'obligation d'utiliser un ensemble supplémentaire de structures de données, que nous avons opté pour la seconde proposition. De plus, l'utilisation de files d'attentes ne nécessite pas de génération de communications supplémentaires : l'ensemble des opérations sont uniquement faites au niveau du modèle de calcul, donc du module d'exécution.

Chaque bus du système de stockage simulé est associé à une file d'attente, possédant les requêtes souhaitant emprunter le bus de communication. On définit alors une nouvelle structure de données, appelée unité de transfert (`TransferUnit`). La structure est décrite dans le code 4.1.

```
1 struct TransferUnit {
2     double      date;
3     long        idxRequest;
4     long        size;
5     short       type;
6     short       step;
7     bool        toDevice;
8     bool        user;
9 };
```

Code 4.1 – Structure TransferUnit

La description des champs de l'unité de transfert est la suivante :

— `date` – date à laquelle l'unité de transfert est prête à être transférée par le bus de communication ;

— `idxRequest` – indice de la requête ciblée par l'unité de transfert, pour faire le lien avec le tableau pour avoir accès au détail de la requête ;

— `size` – taille de l'unité de transfert ;

— `type` – type de l'unité de transfert, choix entre soumission (`SUB`), données (`DATA`) et acquittement (`ACK`) ;

— `step` – niveau hiérarchique de la localisation de l'unité de transfert, 0 pour l'hôte, 1 pour le tier, 2 pour le volume et 3 pour le disque ;

— `toDevice` – sens de communication de l'unité de transfert, vrai si de l'hôte au disque, faux si du disque à l'hôte ;

— `user` – priorité de l'unité de transfert, vrai si requête utilisateur, faux si requête système.

La taille de l'unité de transfert dépend de son type. Pour des requêtes ou des acquittements, la taille est fixe (pour nous, à 128 octets). La taille des données transférées dépend de la requête. Le type de l'unité de transfert dépend du sens de communication, et du type de la requête. Il est expliqué plus en détail dans le tableau 4.1. Le dernier champ, correspondant à la provenance de la requête, si elle est issue d'une

demande utilisateur ie. du fichier de trace, ou d'un processus indépendant de l'utilisateur, que l'on qualifie de système, tel la reconstruction d'un disque ou l'exécution d'un mécanisme de gestion permanente du disque.

| Type / Sens | Vers le disque | Vers l'hôte |
|---|---|---|
| Lecture | SUB | DATA & ACK |
| Ecriture | SUB & DATA | ACK |

TABLE 4.1 – Types possibles de l'unité de transfert

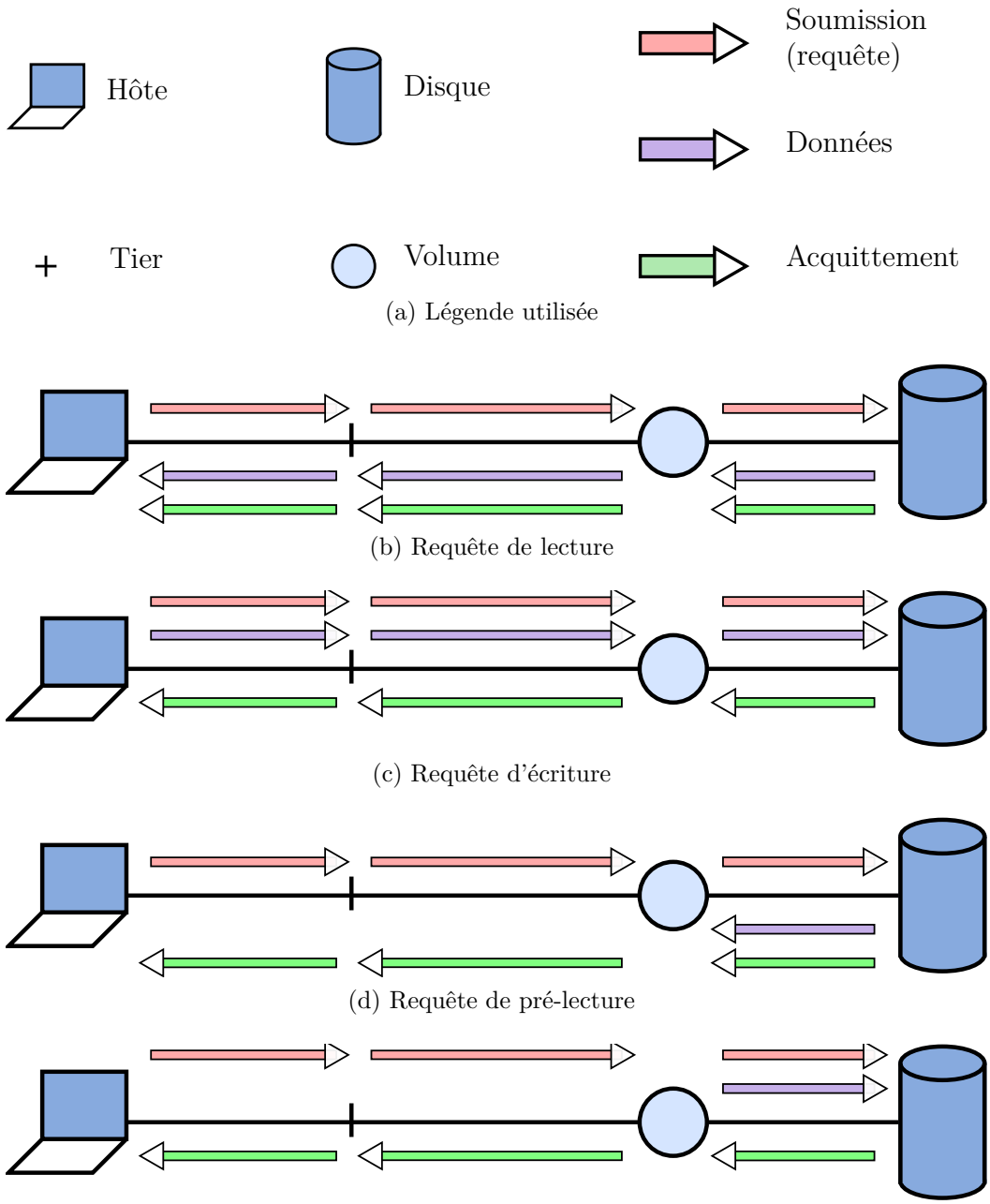
La figure 4.6 montre les cheminements des unités de transfert à travers les bus, en fonction du type des requêtes qu'elles représentent. La légende des schémas est représentée sur la figure 4.6a. Pour tous les types de requêtes, on considère que la soumission de la requête provient de l'hôte, qu'elle est transférée jusqu'au disque, afin qu'il puisse exécuter la requête demandée, puis l'acquittement fait le chemin inverse. La différence provient du parcours fait par les données. Dans le cas de requêtes utilisateur, les données sont transférées de l'hôte jusqu'au disque en cas de lecture (figure 4.6b), et remontent du disque vers l'hôte en cas d'écriture (figure 4.6c). S'il s'agit de requêtes système, les données ne sont déplacées qu'entre le contrôleur de volume et le disque. Pour des pré-lectures, les données transitent depuis le disque (figure 4.6d). Pour les écritures de code d'erreur, les données sont envoyées sur le disque (figure 4.6e).

L'étude des cheminements de chaque type de sous-requête permet de mettre à jour les unités de transfert après chaque passage dans un bus. Par exemple, l'unité de transfert d'une requête en lecture arrivant au disque est mis à jour comme indiqué dans l'algorithme 8. On observe que la date à laquelle l'unité de transfert est prête à traverser le bus est calculée suite aux temps d'attente pour l'accès au bus et au disque, ainsi que le précédent temps de transfert et le temps de service par le disque. La taille de l'unité est également modifiée, pour correspondre à la taille des données lues sur le disque, à laquelle est ajoutée la taille de l'acquittement. Le type de l'unité de transfert devient donc, comme indiqué dans le tableau 4.1 la concaténation des données et de l'acquittement. Enfin, le sens de communication de l'unité de transfert est dans la phase montante, du disque vers l'hôte, et cette phase démarre au niveau le plus bas de la hiérarchie du système.

# 5 Gestion de la priorité au niveau du bus

La dernière partie dédiée à la gestion de la synchronisation concerne la sélection des unités de transfert à traiter par le module d'exécution, à savoir quelle est la prochaine unité à pouvoir être transférée. Chaque file d'attente est ordonnée selon deux critères : la date à laquelle l'unité de transfert est prête à être transférée, et la priorité de la requête. Si deux unités de transfert possèdent la même date, alors celle possédant la priorité la plus haute est transférée en premier, c'est à dire la requête de type utilisateur. La sélection est décrite dans l'algorithme 9. La file d'attente sélectionnée est alors celle possédant l'unité de transfert avec la date la plus ancienne.

La dernière condition de l'algorithme (ligne 7) permet d'assurer la synchronisation du traitement. L'antériorité d'une requête par rapport à une autre dépend de sa date




(a) Légende utilisée

(b) Requête de lecture

(c) Requête d'écriture

(d) Requête de pré-lecture

(e) Requête d'écriture de code d'erreur

FIGURE 4.6 – Cheminement des requêtes dans les bus de communication

**Algorithme 8 :** Traitement d'une requête d'écriture de code d'erreur par un pilote de disque

**Entrées :** $u$ – unité de transfert
$b$ – bus situé entre le contrôleur de volume et le disque
$T$ – tableau des requêtes
**Sorties :** $u'$ – unité de transfert mise à jour

1 $u'.date \leftarrow u.date+$ temps d'attente pour l'accès au bus $b$ + temps de transfert par $b$
2 $u'.date \leftarrow u'.date+$ temps d'attente pour l'accès au disque + temps de service de la requête
3 $u'.idxRequest \leftarrow u.idxRequest$
4 $u'.size \leftarrow T[u'.idxRequest].taille + 128$
5 $u'.type \leftarrow DATA \wedge ACK$
6 $u'.step \leftarrow 3$
7 $u'.toDevice \leftarrow faux$
8 $u'.user \leftarrow u.user$

**Algorithme 9 :** Sélection de l'unité de transfert à traiter

**Entrées :** $F$ – ensemble des files d'attente
**Sorties :** $u$ – unité de transfert sélectionnée

1 $t_{min} \leftarrow$ inf **pour chaque** *file d'attente* $f \in F$ **faire**
2 | $u' \leftarrow$ premier élément de f
3 | **si** $u'.date < t_{min}$ **alors**
4 | | $u \leftarrow u'$
5 | | $t_{min} \leftarrow u.date$
6 | **fin**
7 | **si** *toutes les requêtes antérieures à u.idxRequest ont été traitées* **alors**
8 | | **retourner**
9 | **fin**
10 | $u \leftarrow$ vide
11 **fin**

d'arrivée théorique (ou de son estampille si les dates d'arrivées sont identiques), qui doit être inférieure. Il faut donc vérifier, pour chaque requête antérieure à celle que l'on souhaite sélectionner, son état qui peut indiquer :

— soit elle est terminée – chaque unité de transfert composant la requête a été traitée, et les acquittements ont été reçus par l'hôte ;

— soit elle est en cours de traitement – chaque unité de transfert composant la requête a été traitée ou est en cours de traitement. Ceci est vérifié par leur présence dans les files d'attente dans les buffers associés aux composants (disque, contrôleur volume, hôte) ;

— soit son traitement n'a pas démarré – au moins une unité de transfert composant la requête n'a pas encore été insérée dans une file d'attente.

Si la requête antérieure est dans le premier cas, alors elle a été traitée, et n'empêche pas le traitement de l'unité de transfert. Si la requête antérieure est dans le second cas, alors la date de disponibilité des unités de transfert encore présentes dans les files

d'attente sont supérieures à celle de l'unité sélectionnée, ce qui n'empêche pas son traitement. Par contre, si la requête antérieure se trouve dans le troisième cas, alors il est impossible de déterminer la date de disponibilité réelle de ses unités de transfert, et le traitement de l'unité de transfert sélectionné ne peut pas s'effectuer.

Pour éviter que toutes les requêtes antérieures ne soient vérifiées à chaque sélection d'unité de transfert, une solution a été proposée et implémentée. Elle consiste en l'utilisation d'un compteur, indiquant la première requête utilisateur non terminée. L'incrémentation de ce compteur intervient lors de la finalisation du traitement d'une requête, si son indice est égal au compteur.

L'utilisation de ce mécanisme de synchronisation englobe celui expliqué dans la section 3, en ajoutant lors de la vérification des requêtes antérieures de l'algorithme 9, une contrainte sur les pré-lectures. Une unité de transfert représentant une sous-requête d'écriture de code d'erreur atteindra le contrôleur du volume lors de sa phase descendante. Puis elle attendra que les pré-lectures ne reviennent lors de la phase montante à ce même contrôleur pour pouvoir continuer. Ceci permet donc d'éviter une sur-utilisation des canaux de communication, et de mettre systématiquement en attente les pilotes de disque, exploitant au mieux le parallélisme.

# 6 Mise à l'échelle d'OGSSim

L'étude de la scalabilité d'OGSSim consiste en l'observation du comportement du logiciel lorsque la taille du système simulé augmente. OGSSim utilise deux formes de parallélisme. Tout d'abord, un parallélisme de tâches est effectué, où chaque module de l'outil est associé à un thread. Ensuite, dans la chaîne de décomposition des requêtes, les données sont parallélisées entre les pilotes de volume d'une part, puis les pilotes de disques d'autre part. Ce parallélisme est possible car chaque exécution d'un pilote est indépendante de celle d'un autre pilote. Effectivement, la décomposition d'une requête $i$ est uniquement liée à l'architecture du système simulé.

Le tableau 4.2 détaille le nombre de threads générés par module, à chaque exécution d'OGSSim. On peut donc observer que ce nombre est directement proportionnel au nombre de configurations et de disques composant le système simulé.

| Module | Nombre de threads |
|---|---|
| Module de chaîne d'extraction | 1 |
| Pré-traitement | 1 |
| Pilote de volume | 1 par volume |
| Pilote de disque | 1 par disque |
| Module de chaîne de calcul | 1 |

Table 4.2 – Nombre de threads générés en une exécution d'OGSSim

Un phénomène indésirable était néanmoins observé lors des premières exécutions d'OGSSim simulant des systèmes de stockage à large échelle. Lorsque le système pos-

sédait plus de 300 disques, notre outil avait des difficultés à initialiser le contexte de simulation.

Pour chaque pilote de disque, un canal de communication est créé afin que le pilote de volume lui transmette les indices de requêtes qu'il a à traiter. Ce nombre de canaux est limité, dans un premier temps par le formatage du fichier de configuration d'OGSSim. Le fichier de configuration décrit le paramétrage des canaux ZeroMQ par, notamment, le port de communication. Les ports utilisés sont ici choisis directement par l'utilisateur, et sont compris entre 0 et 65'535. Sachant que le nombre de pilotes de volume et de disque dépend du système simulé, le port indiqué correspond à la première valeur utilisable. Par exemple, si le fichier de configuration possède la ligne de paramétrage décrite dans le code 4.2, alors les ports utilisés par les pilotes de volume démarre par le numéro 5670. La borne maximale n'est quant à elle pas directement définie, et va dépendre des numéros de port utilisés par les autres modules.

```
1 <zeromq intr="preproc" prot="tcp" addr="*" port="5670" />
```

Code 4.2 – Paramétrage ZeroMQ pour un pilote de volume

Dans une première version d'OGSSim, le paramétrage des ports de communication dans le fichier de configuration n'était pas optimisé et leur chevauchement était possible pour des systèmes composés d'un nombre modéré de volumes ou de disques. De plus, à l'initialisation des canaux dans les modules concernés, le calcul du port en fonction de la borne minimale se limitait au millier supérieur. La nouvelle version du logiciel lève cette limitation. Le tableau 4.3 détaille les plages de ports valides pour la première version d'OGSSim et la version actuelle. On peut voir que, dans la version actuelle, le nombre de volumes est limité à 300.

| | Première version | | Version actuelle | |
|---|---|---|---|---|
| | Borne minimale | Plage de ports | Borne minimale | Plage de ports |
| Pilote de volume | 5670 | 5670 - 5699 | 6000 | 6000 - 7999 |
| Pilote de disque | 5700 | 5700 - 5999 | 8000 | 8000 - 65535 |

Table 4.3 – Ports disponibles pour les canaux de communication

Une autre limitation peut empêcher le bon fonctionnement d'OGSSim : le nombre de descripteurs de fichiers disponibles (Linux Foundation, 2016). Chaque connexion à un canal de communication génère une ouverture de fichier, ce qui donne pour un système d'un millier de disques, plus de 2'000 descripteurs de fichier ouverts. Dans un système d'exploitation de type Unix, la variable contenant cette valeur peut être modifiée dans le cas où elle serait trop faible. Elle est stockée dans le fichier `/etc/security/limits.conf`, et est modifiée à l'aide du texte donné dans le code 4.3, pour l'utilisateur nommé `username`. La valeur *soft* est la valeur maximale atteignable par l'utilisateur, la valeur *hard* nécessite d'être administrateur pour être atteinte.

```
1 username soft nofile 4096
2 username hard nofile 16384
```

CODE 4.3 – Paramétrage de la limite du nombre de fichiers ouverts

La figure 4.7 montre le temps d'exécution de notre outil de simulation pour un jeu de 500'000 requêtes sur un système comprenant entre 128 et 3'000 disques. Les tests sont conduits sur un processeur octo-core Intel "Haswell" 3.6 GHz, couplé à 16 Go de RAM. Le système d'exploitation est Ubuntu 15.10 64-bit. OGSSim est construit avec le compilateur C++ de GCC 5.2.1. avec le paramètre d'optimisation `-O3`. Le système simulé est constitué à 38% de RAID-5 et à 62% de RAID-01. On peut voir que la croissance du temps de simulation en fonction de la taille du système n'est pas linéaire. Néanmoins, elle reste bien satisfaisante et le temps de simulation est inférieur à 10 secondes pour 3'000 disques. Effectivement, le nombre de disques n'impacte que la phase d'initialisation de la simulation, où les threads sont créés.

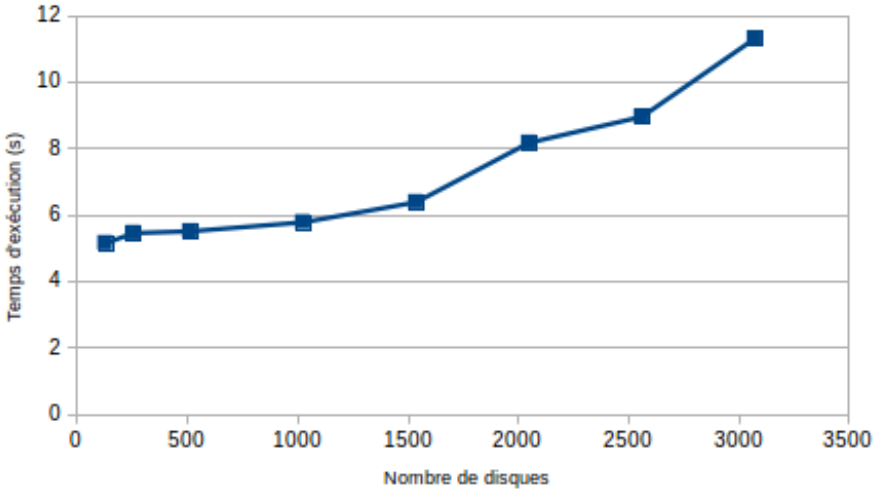



FIGURE 4.7 – Temps d'exécution d'OGSSim en fonction de la taille du système

La limite du nombre de disques dépend d'un autre facteur : la limite du nombre de threads autorisés par le système d'exploitation. Cette limite est liée à la taille de la pile allouée à chaque thread, ainsi qu'à la taille de la RAM (Torvalds, 1992), limite alors imposée par la machine utilisée pour l'expérimentation. Dans notre cas et pour l'exemple de la figure 4.7, cette limite est de 3'000 disques.

# 7 Etude du temps d'exécution d'OGSSim

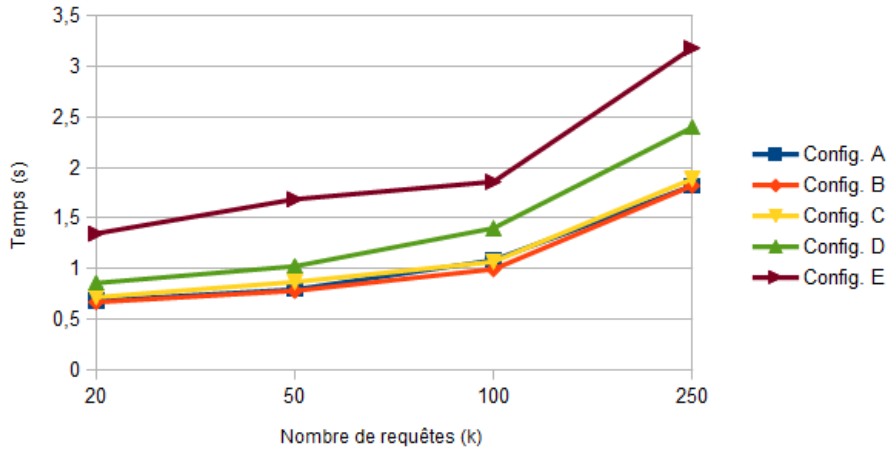
Pour évaluer les performances de notre outil de simulation, nous allons mesurer le temps d'exécution d'OGSSim selon deux paramètres : le nombre de disques composant le système simulé, allant de 32 à 512 disques, et le nombre de requêtes, de 20k à 250k. Les tests sont conduits sur la même machine décrite dans la section précédente. Le système d'exploitation est Ubuntu 15.10 64-bit. OGSSim est construit avec le compilateur C++ de GCC 5.2.1. Le tableau 4.4 détaille les différentes configurations architecturales utilisées.

| | Nb. disques | Nb. tiers | Vol. par tier | Disques par vol. | Nb. bus |
|---|---|---|---|---|---|
| Config. A | 32 | 2 | 2 | 8 | 7 |
| Config. B | 64 | 2 | 2 | 16 | 7 |
| Config. C | 128 | 2 | 4 | 16 | 11 |
| Config. D | 256 | 4 | 4 | 16 | 21 |
| Config. E | 512 | 4 | 8 | 16 | 37 |

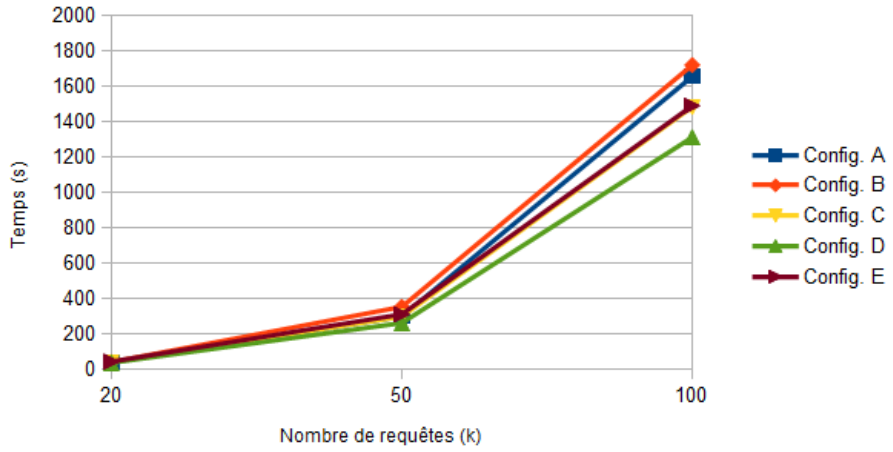
Table 4.4 – Configurations pour l'étude du temps d'exécution d'OGSSim

Les résultats de l'expérimentation sont présentés sur la figure 4.8. Nous avons utilisé deux types de jeux de requêtes, différenciant ainsi les cas où il peut y avoir recouvrement temporel des requêtes (donc attente) ou non. On dit qu'il y a recouvrement si une requête est envoyée au disque avant que la (ou les) précédente(s) ne soi(en)t exécutée(s). La figure 4.8a montre les résultats des expérimentations sans recouvrement temporel. On observe qu'avec l'augmentation de la taille du système simulé, le temps d'exécution augmente également. Ceci est dû au nombre de bus du système, qui est directement lié au nombre de files d'attente utilisées pour le calcul du temps de transfert des requêtes. Pour chaque requête, le modèle de calcul recherche dans chaque file d'attente la requête qui doit être exécutée, et la configuration E possède à peu près 5 fois plus de files d'attente que les configurations A et B. Le temps d'exécution reste néanmoins inférieur à 3,5 secondes pour 250k requêtes.

La figure 4.8b traite du recouvrement temporel. On peut alors voir que les temps d'exécution sont assez semblables pour l'ensemble des configurations, et beaucoup plus importants que dans le cas où il n'y a pas de recouvrement. Le temps d'exécution est inférieur à 30 minutes pour les configurations testées, et des jeux de 100k requêtes. Ceci s'explique également par le modèle de calcul du temps de transfert. La présence de recouvrement implique que plusieurs requêtes sont présentes au même instant dans les files d'attente, et que plusieurs comparaisons sont effectuées pour déterminer la prochaine requête à traiter. De plus, à cause du mécanisme de synchronisation, certaines requêtes ne peuvent pas être traitées car les requêtes antérieures à ces premières ne sont pas encore arrivées au module d'exécution, ce qui retarde le traitement. On observe également que les configurations possédant le plus grand nombre de bus s'exécutent plus rapidement que les autres. La raison réside en l'équilibrage de charge sur les files d'attente des bus : plus il y a de files d'attente, moins il y a de requêtes par file et donc moins de comparaisons à effectuer pour le tri de ces files.




(a) Sans recouvrement temporel de requêtes




(b) Avec recouvrement temporel de requêtes

FIGURE 4.8 – Temps d'exécution d'OGSSim selon la configuration matérielle

# 8 Conclusion et perspectives

Les mécanismes de synchronisation décrits, avec ou sans l'utilisation de canaux de communication, répondent à un besoin primordial de l'outil de simulation ou de la nature du phénomène simulé. Le premier mécanisme évite l'inter-blocage des threads au cas où la structure de données stockant les requêtes temporaires ne possède plus d'emplacements disponibles. Le second mécanisme assure l'intégrité de la sémantique d'exécution des requêtes d'écritures générant des pré-lectures ainsi que le calcul des métriques de performance associées.

OGSSim permet la simulation de systèmes à large échelle. L'environnement de simulation utilisé supporte des systèmes composés de 3'000 disques. En ce qui concerne le temps d'exécution d'OGSSim, on a pu observer pour des systèmes composés de plus de 500 disques, donc dits à large échelle, un temps de simulation de l'ordre de la seconde avec des jeux de 250k requêtes ne présentant pas de recouvrement temporel. Dans le cas inverse, le temps d'exécution ne dépasse pas la demi-heure sur l'ensemble des configurations architecturales testées, pour une simulation de 100k requêtes.

Une amélioration envisagée pour le problème d'inter-blocage est le changement de politique quant à la structure des données utilisée pour stocker les sous-requêtes. Notre contrainte originelle pour cette structure était qu'elle soit de taille fixe et paramétrable par l'utilisateur. Ainsi, la structure peut être facilement adaptée aux caractéristiques de la machine utilisée pour l'expérimentation, et les perpétuels ajouts et suppressions de requêtes ne génèrent pas d'allocation/libération d'espace à outrance. La structure de données envisagée est un vecteur, donc de taille variable, mais allouant un ensemble important de sous-requêtes (de l'ordre de la dizaine de milliers par exemple). De plus, aucune désallocation ne sera effectuée : les emplacements non utilisés restant disponibles pour les prochaines sous-requêtes. Cette structure évite d'appeler un nombre important de fois les primitives systèmes liées à l'allocation et la gestion de la mémoire, et supprime le besoin de la synchronisation entre les pilotes de volume et le module d'exécution.

Une autre perspective à court terme est prévue avec la généricité du modèle de communication d'OGSSim. Dans sa version actuelle, notre outil de simulation ne fonctionne qu'en mémoire distribuée, utilisant les ZeroMQ comme moyen de communication entre les threads. Dans notre projet d'ouverture et de généricité d'OGSSim, nous souhaitons offrir plusieurs alternatives à l'utilisateur, en fonction de son contexte d'expérimentation, ou au pire une interface d'implémentation. Ceci permettra également de séparer l'aspect extraction, décomposition ou calcul des modules existants de l'aspect communication. Cette nouvelle suite de modèles de communication ne demandera plus le paramétrage de la communication par l'utilisateur, afin de ne pas rencontrer de problèmes de scalabilité similaires à celui présenté.

# Chapitre 5

# Le declustering orienté robustesse



*Les systèmes de stockage à large échelle actuels sont de plus en plus sujets aux défaillances, ie. la panne d'un disque. La robustesse d'un système est déterminée par sa capacité à réduire l'impact de ces défaillances. Soit par l'utilisation d'un mécanisme de tolérance aux pannes, ce qui permet la récupération des données perdues à chaque accès demandé par l'utilisateur. Soit par la reconstruction directe des données sur des disques sains ou des blocs de données vierges.*

*La simulation peut permettre d'évaluer et d'analyser le comportement des systèmes de stockage face à ces défaillances. Notre outil OGSSim intègre alors la possibilité d'introduire des défaillances durant une simulation, et obtenir ainsi les différentes métriques de performances.*

*Nous allons dans un premier temps décrire l'intégration du mode défaillant à OGSSim, notre outil de simulation. Puis nous dresserons un état de l'art sur les configurations de stockage orientées robustesse, et plus précisément le declustered RAID. Enfin, nous détaillerons notre premier algorithme de création de schéma de placement des données, à l'aide de quatre contraintes définies.*

# 1 Implémentation du mode défaillant dans OGSSim

La première version d'OGSSim ne possédait qu'un seul mode de simulation : le mode normal. Dans cette version, le module d'événements détaillé dans le chapitre 3 n'était pas encore implémenté et il n'était donc pas envisageable de supporter un autre mode de fonctionnement.

## 1.1 Comportement du module de pré-traitement

L'ajout des événements et leur traitement était nécessaire à la gestion du mode défaillant. Un événement est décrit à l'aide de deux champs obligatoires : son type et la date de son arrivée. A ceux-ci peuvent se rajouter des paramètres optionnels. Par exemple, pour un événement de type défaillance, le paramètre cible permet d'indiquer le disque qui va être victime de la défaillance.

L'ensemble des événements sont reçus par le module de pré-traitement à l'initialisation d'OGSSim. Ils sont ensuite triés par date, avant de démarrer la simulation. L'algorithme 10 décrit le comportement du module de pré-traitement durant la simulation. Le pré-traitement consiste en un parcours des requêtes du tableau, pour effectuer la première étape de la décomposition. Les événements, étant des objets datés, oblige à les parcourir en même temps que les requêtes. Ils sont donc gérés par la condition de la ligne 3 : si le prochain événement à traiter possède une date inférieure à celle de la prochaine requête, alors on traite l'événement.

**Algorithme 10 :** Processus de simulation du module de pré-traitement

**Entrées :** $T_R$ – table des requêtes
$T_E$ – table des événements
$n_R$ – nombre de requêtes
$n_E$ – nombre d'événements

1 $i_E \leftarrow 1$
2 **pour** $i_R$ *allant de* 1 *à* $n_R$ **faire**
3 | **si** $i_E \leq n_E$ **alors**
4 | | **tant que** $T_E[i_E].date \leq T_R[i_R].date$ **faire**
5 | | | traitement de $T_E[i_E]$
6 | | | $i_E \leftarrow i_E + 1$
7 | | **fin**
8 | **fin**
9 | traitement de $i_R$
10 **fin**

Le module de pré-traitement n'est pas le seul module à être impacté par les événements, puisque chaque module des chaînes de décomposition et de calcul a un comportement spécifique à adopter à la réception d'un événement. Par exemple, le pilote de volume doit à la réception d'un événement de type défaillance, si la configuration ciblée possède un mécanisme de tolérance à la panne, décomposer chaque requête à destination du disque défaillant pour récupérer les données issues du même stripe, et reconstruire la donnée perdue à l'aide d'un calcul de parité ou de code d'erreur.

Les seuls événements supportés dans la version actuelle d'OGSSim sont les défaillances qui simulent, à une date donnée, la panne d'un disque. On dit qu'à l'arrivée d'une défaillance dans la simulation, le fonctionnement du système passe en mode défaillant. D'autres événements peuvent être introduits dans le futur, ainsi que les fonctions qui les traitent.

## 1.2 Comportement du pilote de volume

Selon la configuration architecturale, le pilote de volume réagit différemment à l'arrivée d'une défaillance dans le système. On distingue alors trois types de configurations parmi celles supportées par OGSSim :

1. Celles possédant un mécanisme de tolérance à la panne, assurant une récupération ponctuelle des données défaillantes.
2. Celles possédant un mécanisme de reconstruction pérenne des données défaillantes.
3. Celles ne possédant aucun mécanisme lié à la robustesse.

On précise que le mécanisme de tolérance à la panne permet de calculer les données perdues à l'aide de pré-lectures sur les données saines, tandis que le mécanisme de reconstruction écrit sur des blocs vierges, les données perdues à la suite de la défaillance.

Les différences de comportement entre ces trois mécanismes sont référencées dans le tableau 5.1. Quelque soit le mécanisme, à chaque réception d'une défaillance le pilote de volume va en premier lieu informer le pilote de disque ciblé qu'il doit maintenant passer en mode défaillant. Le comportement du pilote de disque en mode défaillant est expliqué dans la section 1.3.

| Comportement | Type | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| Envoi d'un message au disque ciblé | X | X | X |
| Récupération ponctuelle des données à la réception d'une requête | X | – | – |
| Reconstruction pérenne des données à la réception de la défaillance | – | X | – |

Table 5.1 – Comportements adoptés par chaque type de configuration

### Récupération ponctuelle

Le mécanisme de récupération ponctuelle intervient, au niveau du pilote de volume, à chaque réception de requête ciblant le disque défaillant. La requête reçue est décomposée en une suite de sous-requêtes, consistant en une lecture des données issues du même *stripe* logique ou des données miroir, dépendant du type de la configuration ciblée. Les données récupérées ne sont pas gardées en mémoire, ce qui signifie qu'elles auront besoin d'être recalculées si une nouvelle requête souhaite les récupérer. En cas d'écriture, une sous-requête d'écriture de code d'erreur est ajoutée à la suite de sous-requêtes, pour prendre en compte la mise à jour des nouvelles données.

#### Reconstruction pérenne

Dans le cas d'une configuration possédant des emplacements disponibles à la reconstruction pérenne des données, la gestion de la défaillance se fait à l'interception de l'événement de défaillance. A la suite de cette réception, l'ensemble des blocs de données issus du disque défaillant vont être reconstruits sur des blocs libres. Pour chaque bloc, un ensemble de sous-requêtes va être créé, consistant en la lecture des données issues du même *stripe* logique ou des données miroir, puis en l'écriture des données reconstruites sur un bloc libre.

#### Configuration sans mécanisme de redondance

Dans le cas spécifique où les configurations ne possèdent aucune tolérance à la panne, ie. les JBODs et les RAID-0, les données sont définitivement perdues si le disque les contenant devient défaillant. Ce qui signifie que la récupération des données n'est pas possible, et qu'une requête ciblant le disque défaillant ne sera pas transformée : elle sera directement envoyée au pilote de disque défaillant. Si le mécanisme de reconstruction a lieu, les données perdues ne peuvent pas être reconstruites sur des blocs vierges. Ceci implique que toute lecture sur des données 'reconstruites' échouera. Par contre, toute écriture est possible, les données seront mises à jour correctement sur les blocs vierges, et alors les prochaines lectures à destination de ces blocs réussiront.

Le tableau 5.2 recense pour chaque configuration supportée par OGSSim, les requêtes et opérations générées.

| Configuration | Comportement après panne | Nombre d'opérations (pour reconstruire 1 bloc) |
|---|---|---|
| JBOD / RAID-0 | Sans effet | 0 |
| RAID-1 / RAID-01 | Lecture des blocs natifs/miroirs correspondants aux blocs perdus | 1 lecture + 1 écriture |
| RAID-NP | Lecture des blocs du même *stripe* et opération de parité/code d'erreur | N lectures + 1 écriture |

TABLE 5.2 – Requêtes et opérations générées lors de la reconstruction des données

## 1.3 Comportement du pilote de disque et du module d'exécution

Chaque pilote de disque possède un état booléen, indiquant si le disque ciblé par le pilote est sain ou non. Au début de la simulation, cette variable booléenne indique que le disque est sain et peut donc exécuter les requêtes de lecture ou d'écriture normalement. A la réception d'un événement de type défaillance, le pilote va donc mettre à jour l'état, pour indiquer que le disque est en panne, et ne peut plus traiter les requêtes. Si le pilote reçoit une requête et que son disque est renseigné comme défaillant, un booléen de la requête est alors mis à vrai, indiquant qu'elle ne peut être traitée par le disque.

La gestion du mode défaillant par le module d'exécution passe par la réception de requêtes provenant de disques défaillants. Comme vu précédemment, chaque requête possède un champ indiquant si elle est traitée par le disque ou non. Si le champ est à vrai, alors le disque est sain, et la requête peut être transformée en unité de transfert pour intégrer une file d'attente de bus. Si le champ est à faux, la requête n'est pas traitée et aucun calcul n'est effectué. Ce résultat est reporté dans le fichier de sortie d'OGSSim, et il est possible de construire un graphe de visualisation pour ce métrique, comme par exemple la part de requêtes non complétées durant la simulation, représenté par la figure 5.1.

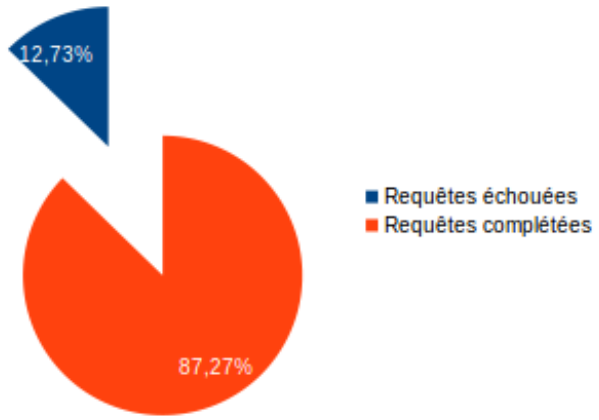



FIGURE 5.1 – Exemple de graphe de visualisation (RAID 4+1)

### 1.4 Perspectives

Des perspectives d'amélioration du mode défaillant d'OGSSim sont possibles. Tout d'abord l'implémentation de nouvelles métriques et graphes de visualisation, quantifiant l'impact du mécanisme de reconstruction sur le traitement des requêtes utilisateur. L'ajout d'un événement de type remplacement de disque pourrait également permettre la simulation de scénarii plus variés axés sur la robustesse et la maintenance des systèmes de stockage.

## 2 Declustered RAID

Pour des systèmes de stockage à large échelle, composés de plus d'une centaine de disques, la probabilité qu'une défaillance intervienne dans le système est d'autant plus grande que le nombre de disques augmente. De plus, une défaillance mène à un processus de reconstruction pérenne des données, qui récupère les données perdues en fonction du mécanisme de redondance utilisé par la configuration, soit redondance intégrale (*mirroring*), parité ou autre code d'erreur, et les réécrit sur un emplacement vierge.

Supposons le cas où le système de stockage nécessite, pour la reconstruction des données, le remplacement du disque défaillant par un disque vierge. Lors du mécanisme de reconstruction, les données lues sont partagées sur chacun des disques sains de

la configuration. Par contre, l'ensemble des écritures vont cibler le nouveau disque, tout juste remplacé, ce qui engendre un surcoût temporel important. Par exemple, un disque de 6 To (Toshiba, 2015), proposant un débit de transfert des données à 170 Mo/s, mettra un minimum de 10 heures à se reconstruire, pour peu que le système ne soit pas sollicité par un utilisateur.

Le *declustered RAID* (Muntz and Lui, 1990) est une manière d'organiser les disques, visant à réduire le temps de reconstruction d'un disque en cas de défaillance. La construction du *declustered RAID* se fait en deux temps :

1. Découpage des disques logiques en blocs.
2. Répartition des blocs logiques sur les disques physiques.

La configuration *declustered RAID* est composée de plusieurs configurations de disques logiques (JBOD, RAID, etc.) qu'elle englobe au sein d'une configuration physique permettant de stocker les blocs de données et leurs redondances respectives de manière distribuée selon un schéma de placement particulier. Ce placement vise, comme précisé auparavant, à privilégier la robustesse du système en accélérant le processus de reconstruction des données en cas de défaillance d'un disque.

La figure 5.2 montre cet exemple de *declustered RAID* où le système logique observé est constitué d'un disque contenant des blocs vierges, que l'on nommera par la suite disque *spare* ($d_{L8}$), et des deux sous-configurations suivantes :

— sous-configuration de 5 disques ($d_{L1}$ à $d_{L5}$), pouvant représenter un RAID-0 ou un RAID-5 ;

— sous-configuration de 2 disques ($d_{L6}$ et $d_{L7}$), pouvant représenter un JBOD ou un RAID-1.

Pour former le *declustered RAID*, chaque disque du système logique est découpé en 5 blocs, et chacun des blocs est réparti sur les disques physiques, notés $d_{P1}$ à $d_{P8}$. Par exemple, les blocs 1.1, 1.2, 1.3, 1.4 et 1.5 du disque $d_{L1}$ sont respectivement placés sur la ligne 1 du disque physique $dL1$, la ligne 3 de $d_{P8}$, la ligne 2 de $d_{P8}$, la ligne 5 de $d_{P4}$ et la ligne 1 de $d_{P2}$.

Si une défaillance intervient sur l'un des disques du système présenté dans la figure 5.2, alors les données reconstruites seront écrites sur les blocs vierges distribués sur le système, plus précisément sur les disques $d_{P1}$, $d_{P3}$, $d_{P5}$, $d_{p6}$ et $d_{p8}$.

Par exemple, dans le cas d'une défaillance touchant le disque $d_{P4}$, comme représenté dans la figure 5.3, les blocs 6.2, 4.1, 3.2, 7.5 et 1.4 sont respectivement reconstruites sur la ligne 1 du disques $d_{P6}$, la ligne 2 de $d_{P3}$, la ligne 3 de $d_{P5}$, la ligne 4 de $d_{P8}$ et la ligne 5 de $d_{P1}$. L'écriture des données reconstruites est alors parallélisée, et durera 5 fois moins de temps qu'avec le système sans *declustered RAID*.

On appelle organisation en mini-RAID, la décomposition d'un système logique en groupe de *stripes* de données auxquels on ajoute des *stripes spare*. Un exemple est donné dans la figure 5.4. On peut voir que les deux mini-RAID sont constitués de trois *stripes* de données et d'un *stripe spare*. Dans l'exemple, les deux mini-RAID

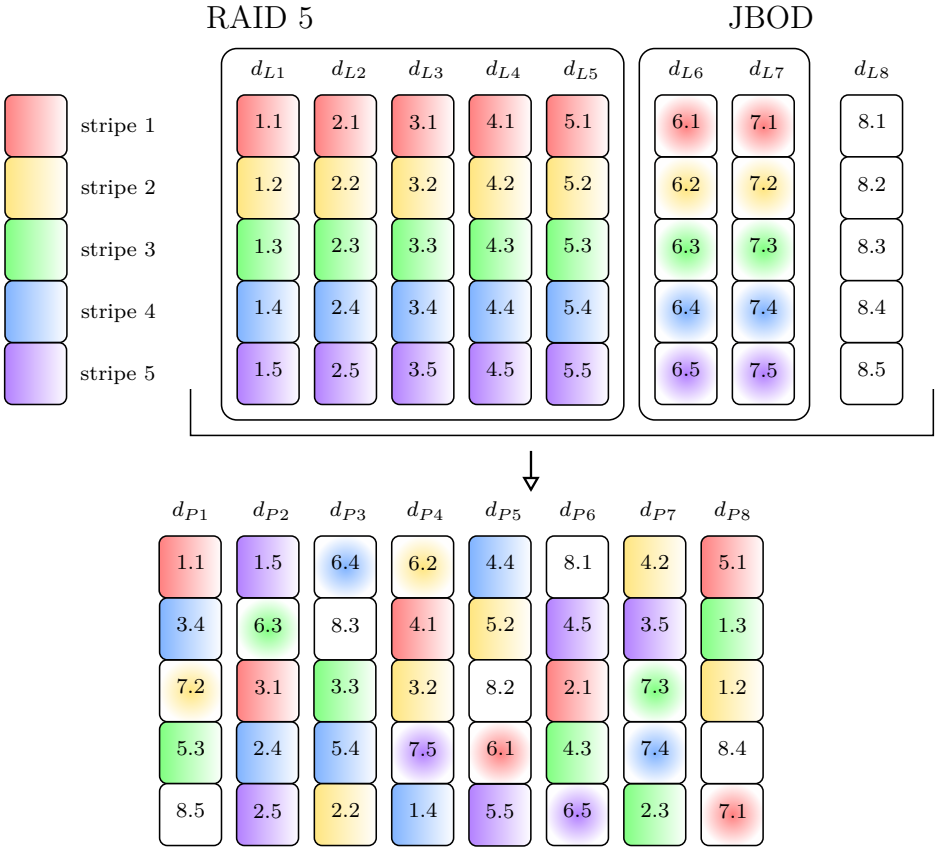



FIGURE 5.2 – Configuration declustered RAID

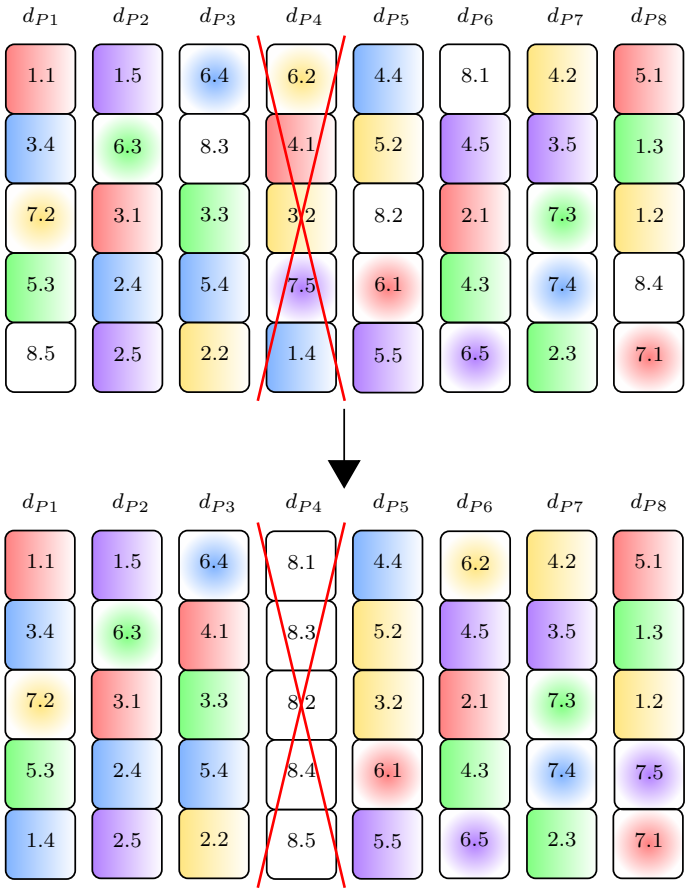



FIGURE 5.3 – Défaillance de $d_{P4}$ sur le système de la figure 5.2

possèdent la même configuration, mais il est possible que le schéma de placement des données diffère d'un mini-RAID à l'autre.

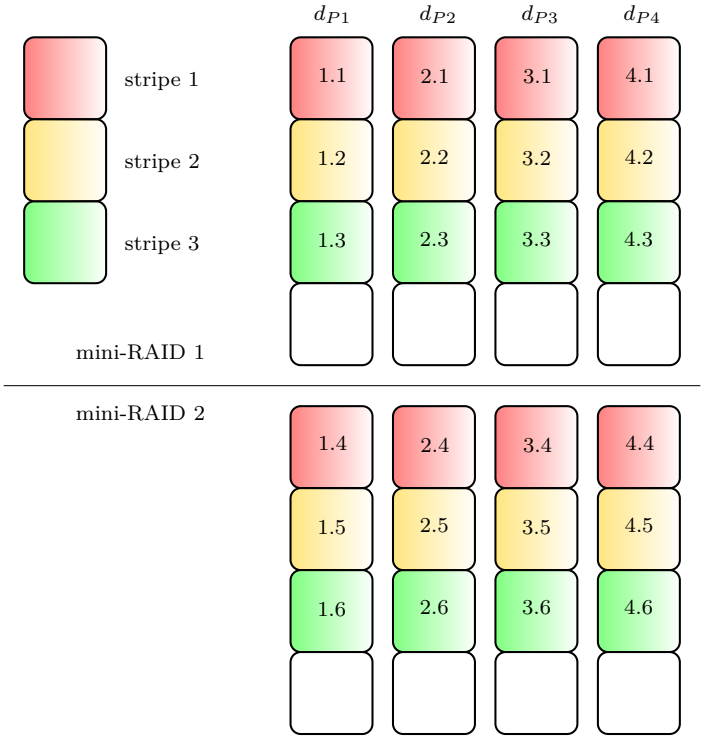



FIGURE 5.4 – Organisation en mini-RAID

## 2.1 Travaux académiques sur le declustered RAID

### a) Naissance du concept

Les fondements du *declustered RAID* ont été développés dans les années 90. Muntz et Lui (Muntz and Lui, 1990) propose, par le biais d'une étude de performance des systèmes de stockage de données en mode défaillant, une nouvelle organisation de type RAID. Cette organisation, alors appelée *clustered RAID*, réduit la dégradation de performances suite à une défaillance et augmente le temps moyen avant la défaillance complète du système.

Le *clustered RAID* est construit en ajoutant à une configuration RAID-5, un disque *spare* logique. Le schéma de placement physique consiste dans un premier temps en un découpage des disques en groupe de $m$ *stripes*, avec $m$ le nombre de disques constituant le RAID-5. Ensuite, les blocs sont répartis sur les disques physiques de telle sorte que, pour les *stripes* d'un même groupe, tous les blocs *spare* sont placés sur le même disque physique. Le disque physique contenant les blocs *spare* change pour chaque groupe de manière cyclique.

Ce schéma de placement permet de réduire le temps nécessaire à la reconstruction des données, par la répartition des requêtes d'écriture sur l'ensemble des disques physiques, de manière équilibrée.

### b) Formalisations et critères de performances

Holand et Gibson (Holand and Gibson, 1992) propose de stocker un RAID-5 logique, où les données sont issues de disques logiques, sur un ensemble de disques physiques de taille plus petite que celle des disques logiques. Ce qui signifie qu'il y a moins de disques logiques que de disques physiques car les systèmes logique et physique possèdent la même capacité de stockage.

Il en est déduit qu'un bon schéma de placement pour un *declustered RAID* devait répondre à un ensemble de six critères :

1. Correction de la simple défaillance
   Un disque physique ne doit pas contenir plus d'un *stripe unit* provenant d'un même *stripe*. Si un disque défaillant en possédait deux, alors les données ne pourraient pas être reconstruites, le RAID-5 étant tolérant à une seule panne.
2. Distribution de la reconstruction
   En cas de défaillance, les requêtes liées à la reconstruction sont équitablement réparties sur les disques.
3. Distribution de la parité
   Les blocs de parité sont équitablement répartis sur les disques.
4. Schéma de placement efficace
   La traduction logique/physique des adresses des blocs ne doit pas consommer de temps d'exécution ou d'espace mémoire excessif.
5. Optimisation des larges écritures
   L'écriture de données contigües correspondant à la taille d'un *stripe* doit correspondre à l'allocation d'un *stripe* entier. Ceci afin d'éviter la phase de pré-lecture pour la mise à jour de la parité.
6. Optimisation du parallélisme
   La lecture de données contigües d'une taille équivalente à la taille d'un *stripe unit* autant de fois que de disques physiques implique un parallélisme des requêtes de lecture sur tous les disques.

Le schéma de placement alors proposé par Holand et Gibson consiste en une répétition de tuples. Le tuple est un ensemble d'indices de disques physiques, et renseigne sur la position physique des *stripe units* d'un *stripe*. La construction des tuples répond à un ensemble de règles correspondant aux critères décrits ci-dessus.

Ce premier mécanisme avait pour but de moins solliciter les disques en cas de reconstruction suite à une défaillance. Effectivement, les *stripe units* issus du même *stripe* logique sont toujours placés sur des disques différents. Et comme le nombre de disques physiques est supérieur au nombre de disques logiques, la reconstruction d'un *stripe unit* ne nécessitera pas la lecture de données provenant de tous les disques physiques.

### c) Exploitation de la localité

Les travaux de Gibson et Patterson (Gibson and Patterson, 1993) sont axés sur la localité des disques au sein d'un système. En supposant un système composé de $m$ RAID-5 de $n$ disques chacun, le but est d'installer sur chaque baie, un disque de chaque configuration RAID-5. Ainsi, le système physique serait composé de $n$ baies,

où chaque baie dépend d'un contrôleur, d'une source électrique et d'un bus pour les transferts de données qui lui est propre.

Cette répartition apporte plusieurs intérêts. Le premier est qu'en cas de défaillance d'un disque, la reconstruction impliquera les disques qui faisaient partie du même RAID-5 que le disque défaillant, et donc que les transferts utiliseront tous les bus disponibles, amoindrissant la congestion des bus. Le second est dans le cas où la totalité d'une baie subit une défaillance, alors les données pourront tout de même être reconstruites, car un seul disque de chaque configuration aura été sujet à la panne.

Gibson et Patterson présentent également l'utilisation de disques *spare*, au sein du système pour la reconstruction des données. Les disques *spare* sont ajoutés soit en tant qu'équivalent d'un RAID-5, et donc un disque sur chaque baie, soit en tant que baie complète. Dans le premier cas, la défaillance d'une baie reconstruit les disques de données dans une baie où un disque de la même configuration est déjà présent. Ce qui signifie qu'une nouvelle défaillance provoque une perte de données. Dans le second cas, une baie défaillante est reconstruite sur la baie vierge, ce qui ne change pas la tolérance du système. Mais si des défaillances ciblent des disques isolés appartenant à la même configuration, alors ces disques sont reconstruits sur la même baie. En cas de défaillance de cette baie, il y a perte de données.

#### d) Optimisations

Schwabe et al. (Schwabe et al., 1996) compare plusieurs types de schémas de placement approximativement optimisés, afin de les comparer au schéma introduit par Holand et Gibson (Holand and Gibson, 1992) qui est lui défini comme parfaitement optimisé car répondant aux critères d'optimisation vus ci-dessus. Les trois types de génération de schéma sont les suivants :

1. construction aléatoire
   Chaque *stripe unit* logique est réorganisé aléatoirement sur un *stripe unit* physique, et l'emplacement de la parité est choisie aléatoirement.
2. non respect d'une contrainte
   Au long de la construction d'un schéma optimal, la distribution de la parité n'est pas optimisée.
3. perturbation d'un schéma optimal
   Si un schéma optimal est trouvé pour une configuration semblable ie. nombre de disques et/ou de *stripe* proches, alors l'adaptation du schéma pour la configuration est souhaitée.

La simulation a montré que les trois types de construction de schéma ont des efficacités très proches de celle proposée par Holand et Gibson, avec un léger avantage pour les constructions avec non respect de la distribution de la parité. De plus, construire un schéma approximativement optimisé prend moins de temps d'exécution. Enfin, la solution de Holand et Gibson a l'inconvénient de ne pas pouvoir être appliquée sur n'importe quel type de configuration (dépendant du nombre de disques et du nombre de *stripes*), alors que la construction aléatoire, par exemple, l'est.

## 2.2 Le declustered RAID au sein des systèmes de fichiers

### a) Le système GPFS

*General Parallel File System* (GPFS) (Schmuck and Haskin, 2002) est un système de fichiers développé par IBM, pour des systèmes de stockage partagés entre plusieurs serveurs. Le but premier de GPFS est de fournir à un noeud de calcul, un accès au système de stockage partagé comme s'il était le seul à l'utiliser. Ainsi, le système de fichiers opère un parallélisme des lectures et écritures de fichier pour chaque utilisateur, tout en assurant la cohérence du système à l'aide de mécanismes de synchronisation. Deux mécanismes sont utilisés : des verrous distribués à chaque serveur pour assurer la cohérence des données lorsque plusieurs utilisateurs souhaitent accéder aux même données et une gestion centralisée des situations de conflit générés par un accès multiple.

Une table d'allocation gère la disponibilité de l'espace de stockage. Chaque disque est divisé en blocs, eux même divisé en 32 sous-blocs. La table est alors séparée en deux parties : une première constituée de 32 bits par bloc, indiquant la disponibilité de chaque sous-bloc, et une seconde étant une liste chaînée des blocs ou sous-blocs disponibles, pour directement y accéder.

La traduction logique/physique est faite à l'aide de l'*extendible hashing* (Fagin et al., 1979), un mécanisme de hachage dont la table a sa taille qui s'agrandit ou diminue en même temps que celle du système de fichiers. Dans GPFS, les répertoires sont utilisés comme objet logique de stockage. Chaque répertoire va donc être associé à un numéro de bloc logique à l'aide de la fonction de hachage. Si la taille d'un répertoire dépasse celle d'un bloc logique, alors un bloc supplémentaire est associé au répertoire. L'indice du bloc logique est ensuite convertit pour correspondre à l'indice d'un bloc physique, l'indice du disque étant déterminé par la division de l'indice de bloc par le nombre de blocs par disque.

Généralement, les systèmes de stockage sur lesquels est porté GPFS sont des configurations RAID tolérants à la défaillance, auxquels sont attachés des contrôleurs RAID chargés de reconstruire les données perdues si demandées par l'utilisateur. GPFS supporte également un mécanisme de réplication, qui lors du remplacement d'un disque défaillant, réécrit les données répliquées sur le disque vierge, et s'il n'y a pas de remplacement, peut écrire ces données sur une partie vierge d'un autre disque.

### b) L'algorithme Crush et le système Ceph

*Crush* (Weil et al., 2006a) est un algorithme de placement logique/physique de données, utilisé dans le système de fichiers Ceph (Weil et al., 2006b). Tout comme pour l'algorithme présent dans GPFS, *Crush* consiste en une distribution pseudo-aléatoire des données, par le biais d'objets.

Le système de stockage sur lequel Ceph repose est hiérarchique. Cette hiérarchie est composée de deux types d'éléments : les disques et les *buckets*. Un *bucket* est un ensemble de *bucket* et/ou de disques, tandis qu'un disque est forcément une feuille de la hiérarchie. Par exemple, étant donné que le système de stockage peut être distribué sur plusieurs sites géographiques, et en supposant qu'un site géographique possède

plusieurs baies de stockage ; on pourrait imaginer une hiérarchie avec un premier niveau constitué de *buckets* représentant les sites géographiques où repose le système, puis pour chacun des *buckets*-site, un ensemble de *buckets* représentant les baies. Enfin, chaque *bucket*-baie possède son ensemble de disques.

L'algorithme *Crush* utilise aussi des répliques des objets qu'il va stocker dans plusieurs disques, ceci afin de pallier une défaillance. Basé sur le principe de Gibson et Patterson (Gibson and Patterson, 1993) pour les disques issus d'une même configuration RAID, chaque réplique sera placée dans des *buckets* différents, et si possible, parmi les *buckets* de plus haut niveau dans la hiérarchie. Une fois les *buckets* éligibles au placement d'une réplique trouvés, la sélection du disque se fait à l'aide d'une fonction de hachage.

Plusieurs méthodes de sélection pour les objets d'un *bucket* sont possibles :

— uniforme – chaque composant a la même probabilité qu'un autre d'être sélectionné, le choix est déterminé par la fonction de hachage ;

— liste – les composants sont listés en fonction de la date de leurs ajouts, puis, dépendant du poids que possède le composant le plus jeune, il a une certaine probabilité d'être choisi ;

— arbre – le comportement est le même que pour la liste, sauf que les composants sont stockés dans un arbre binaire. Pour des soucis d'efficacité de parcours, ce choix est adopté si le nombre de composants est important ;

— courte paille – chaque composant est associé à une valeur résultant d'une fonction de hachage, et celui possédant la plus petite valeur est sélectionné.

Chaque méthode a ses avantages et inconvénients. La méthode uniforme est très efficace si le système ne subit aucune modification. Si uniquement des extensions sont à prévoir, alors la méthode par liste permet une répartition temporairement prioritaire sur les nouveaux disques. Par contre, elle perd en rapidité d'exécution si des suppressions de composants surviennent. Enfin, dans le cas où le système est amené à évoluer, par ajout ou suppression de disque, la méthode de la courte paille est à privilégier car elle octroie les migrations de données les plus optimales. La méthode de l'arbre est celle offrant les meilleurs compromis.

## 2.3 Les brevets consacrés au *declustered RAID*

**a)** (Li and Goel, 2012) est le premier brevet faisant état d'un algorithme de placement pour *declustered RAID* consistant à diviser la configuration RAID en mini RAID.

Les mini RAID d'un même *declustered RAID* peuvent avoir des schémas de placement différents, ce qui mène à l'utilisation d'une table de traduction logique/physique pour les blocs de données. Les schémas des mini RAID répondent à la règle suivante : chaque disque physique ne doit pas contenir plus d'un *stripe unit* issu d'un même *stripe* logique. Ainsi, en cas de défaillance d'un disque, les blocs perdus sont reconstruits sur le premier *stripe spare* du mini RAID, tout en respectant la règle énoncée ci-dessus.

Le nombre de défaillances tolérées par ce système dépend du nombre de *stripes spare* insérés dans chaque mini RAID. De plus, la reconstruction des données suite à une défaillance mène à un parallélisme des requêtes d'écriture, étant donné que les *stripe units* sont distribués sur les disques. Par contre, en cas de double défaillance ie. la seconde défaillance intervient avant la fin de la reconstruction des données perdues à cause de la première défaillance, provoquant ainsi une perte de données partielle.

**b)** (Shanbhag et al., 2014) alloue des objets de stockage au sein de deux baies de disques. La seconde baie de disques est utilisée pour stocker la copie des données de la première baie.

Le répartitionnement des données est représenté sur la figure 5.5. Les disques de la première baie physique consistent en un agglomérat de plusieurs configurations RAID. Dans l'exemple donné, trois configurations logiques, RAID-6, RAID-1 et RAID-5, sont répartis sur 5 disques. Les premiers *stripes* des trois premiers disques sont utilisés pour stocker le RAID-5, les mêmes premiers *stripes* des deux autres disques servent à stocker les données du RAID-1. Enfin, les derniers *stripes* sont utilisés pour le RAID-6. La seconde baie physique de stockage est un *declustered RAID*, où les données sont alloués en utilisant un algorithme de placement, comme *Crush* (Weil et al., 2006a) par exemple.

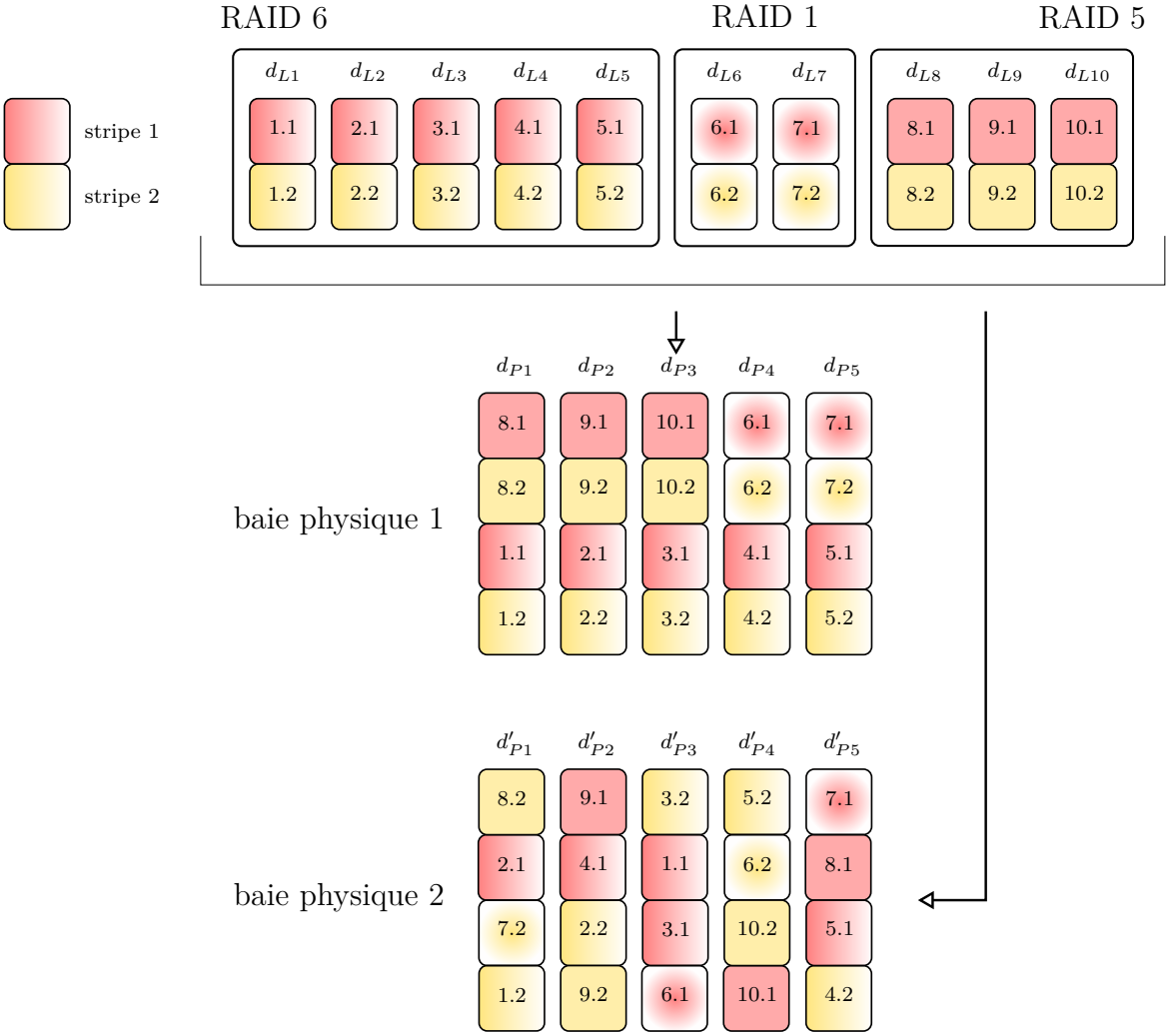



FIGURE 5.5 – Exemple de declustered RAID (Shanbhag et al., 2014)

En mode normal, les requêtes en lecture sont dirigées vers la première baie de stockage. Chaque écriture de données implique une mise à jour sur les deux baies de stockage. En mode défaillant, avec un disque défaillant parmi ceux de la première baie de stockage, les requêtes sont automatiquement redirigées vers la seconde baie.

Le système décrit par ce brevet utilise un mécanisme de réplication ciblant la totalité d'une baie de stockage. Tel que présenté, les requêtes de lecture ne peuvent pas être dirigées à la fois sur la première baie, et sur la seconde, pour mieux exploiter le parallélisme d'accès. Par contre, si une seconde défaillance intervient dans le système, aucune perte de données n'est constatée car :

— si elle touche la même baie de stockage que la première, les disques de l'autre baie sont tous sains ;

— si elle touche l'autre baie de stockage, la redondance des configurations utilisées permet la reconstruction des données.

### 2.4 Bilan

Il existe dans la littérature aussi bien académique qu'industrielle, tout un ensemble de règles à l'établissement d'un schéma de placement optimal pour le *declustered RAID*. Plusieurs méthodes ont été developpées, allant de la construction d'un schéma optimal (Holand and Gibson, 1992) à la répartition aléatoire des données (Schwabe et al., 1996), de la perturbation d'un schéma optimal (Schwabe et al., 1996) à l'utilisation de fonctions de hachage (Schmuck and Haskin, 2002; Weil et al., 2006a).

Chaque méthode a ses avantages et inconvénients, au niveau du nombre de défaillances tolérées ou de l'efficacité du parallélisme des accès apporté. La sauvegarde des données en cas de multiples défaillances est assurée, dans les méthodes récentes, à l'aide de réplicas. L'utilisation de ce mécanisme implique une redondance intégrale et multiple des données. De plus, le schéma de placement, basé sur les fonctions de hachage, nécessite de stocker en mémoire la table de traduction logique/physique qui devient non négligeable sur des systèmes de stockage à large échelle.

## 3 Notation

La table 5.3 présente les différentes notations utilisées dans la suite de ce chapitre et dans le chapitre suivant.

## 4 Contraintes de placement

Pour obtenir un schéma de placement optimisé des données, nous avons défini quatre contraintes. Chacune de ces contraintes sert à garantir une propriété caractérisant le système de stockage ciblé, et peut être axée performance et/ou robustesse.

### 4.1 Contrainte 1 : parallélisme entre les disques

Le parallélisme entre les disques est une contrainte d'optimisation de performance. Elle assure que les blocs d'un disque logique se retrouvent sur des disques physiques distincts. En cas d'accès contigu d'un grand nombre de blocs de données, les accès

| Notation | Commentaire |
|---|---|
| $\lambda$ | Facteur de décalage |
| adr | Adresse logique du bloc |
| unit_size | Taille de bloc |
| $n_d$ | Nombre de disques |
| $n_f$ | Nombre de disques libres |
| $n_v$ | Nombre de volumes |
| $X$ | Indice d'entité logique ($L$) ou physique ($P$) |
| $d_X(i)$ | Disque d'indice $i$ du système $X$ |
| $v(k)$ | Volume logique d'indice $k$ du système |
| $s_X(j)$ | *Stripe* d'indice $j$ du système $X$ |
| $b_X(i,j)$ | Bloc de données du disque $i$ et de *stripe* $j$ du système $X$ |
| $E_p(i)$ | Ensemble de provenance du disque physique $i$ |

TABLE 5.3 – Table de notation pour l'algorithme SD2S

seront alors distribués sur les disques et seront effectués en parallèle. La figure 5.6 représente un exemple de disposition respectant la première contrainte. Le disque logique d'origine du bloc est indiqué par le premier indice du bloc. Ainsi, pour que la contrainte soit respectée, les premiers indices des blocs d'un même disque physique doivent tous être différents. La formalisation de cette contrainte est la suivante :

$$\forall(i,\, j) \in [1, n_d]^2\,;\, \exists(s,\, t) \in [1, n_d - n_f]^2 \text{ tel que}$$
$$b_L(i,s) = b_P(j,t) \Rightarrow b_L(i,s') \neq b_P(j,t')$$
$$\forall(s',\, t') \in [1, n_d - n_f] \setminus \{s\} \times [1, n_d - n_f] \setminus \{t\}$$

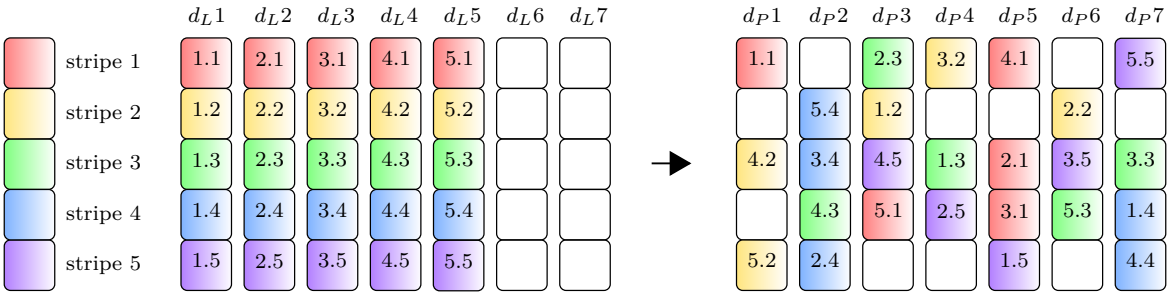



FIGURE 5.6 – Respect de la première contrainte

## 4.2 Contrainte 2 : parallélisme entre les stripes

La deuxième contrainte, en plus d'être une contrainte d'optimisation de performance par l'exploitation du parallélisme entre les *stripes*, est également une contrainte de robustesse. Si deux blocs issus du même *stripe* sont placés sur un même disque physique, et que ce disque devient défaillant, alors deux cas peuvent se présenter :

— soit le système n'est tolérant au maximum qu'à une seule panne et les données sont définitivement perdues ;

— soit le système est tolérant à plus d'une panne et la reconstruction des données est possible et engendrera deux calculs de code d'erreur au lieu d'un.

Sur la figure 5.7, les blocs d'une même ligne de données sont de la même couleur. Les disques physiques ne doivent donc pas posséder de blocs de couleur identique. Cette contrainte est formalisée de la manière suivante :

$$\forall (i,\, j) \in [1, n_d]^2\,;\, \exists (s,\, t) \in [1, n_d - n_f]^2 \text{ tel que}$$
$$b_L(i, s) = b_P(j, t) \Rightarrow b_L(i', s) \neq b_P(j, t')$$
$$\forall (i',\, t') \in [1, n_d] \setminus \{i\} \times [1, n_d - n_f] \setminus \{t\}$$

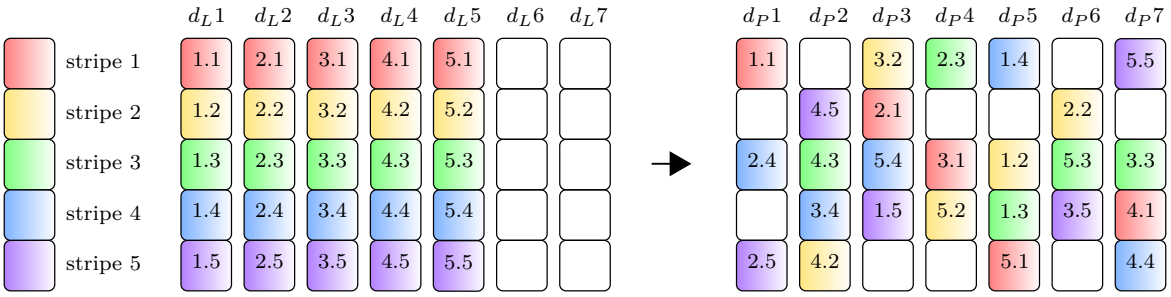



FIGURE 5.7 – Respect de la seconde contrainte

## 4.3 Contrainte 3 : indépendance des ensembles de provenance

La troisième contrainte est imposée pour garantir la fiabilité du système en cas de défaillance multiple. La provenance d'un bloc de donnée résulte de la combinaison de l'indice du volume et du *stripe* logique d'où il provient. Pour un disque physique, on définit son ensemble de provenance comme l'ensemble des combinaisons de volumes et de *stripes* logiques des blocs de données qu'il contient. L'ensemble de provenance d'un disque physique est formalisé de la manière suivante :

$$\forall i \in [1, n_d],\; E_p(i) = \{(k,\, j') \in [1, n_v] \times [1, n_d - n_f]\}$$
$$\text{tel que } \forall j \in [1, n_d - n_f],\; b_P(i,\, j) = b_L(i',\, j') \text{ avec } d_L(i') \in v(k)$$

La figure 5.8 présente pour un système tolérant à une seule défaillance, deux scénarii de double défaillance. Dans le scénario A, le système accuse une défaillance des disques 2 et 3. Ces deux disques possèdent des ensembles de provenance très proches : {bleu, jaune, vert, violet} pour le premier et {bleu, jaune, rouge, violet} pour le second. Ils ne diffèrent donc que d'un seul *stripe*. Cela implique que les *stripes* bleu, jaune et violet ne pourront pas être reconstruits, et on ne reconstruirait donc que 25% des données. Dans le scénario B, la double défaillance affecte les disques 1 et 4, ayant comme ensemble de provenance respectif {bleu, rouge, violet} et {jaune, vert}. Les deux ensembles de provenance sont disjoints, ce qui signifie que chaque bloc de données perdu peut être reconstruit grâce au code d'erreur adéquat. Cela implique une récupération de la totalité des données, ce qu'on cherche à obtenir pour assurer la robustesse du système.

Cette contrainte a pour but d'éviter la présence d'ensembles de provenance égaux entre les différents disques pour prévenir la perte totale des données. En sachant que plus les ensembles sont différents deux à deux, plus le taux de reconstruction de données en cas de double défaillance est grand. La formalisation de cette contrainte est la suivante :

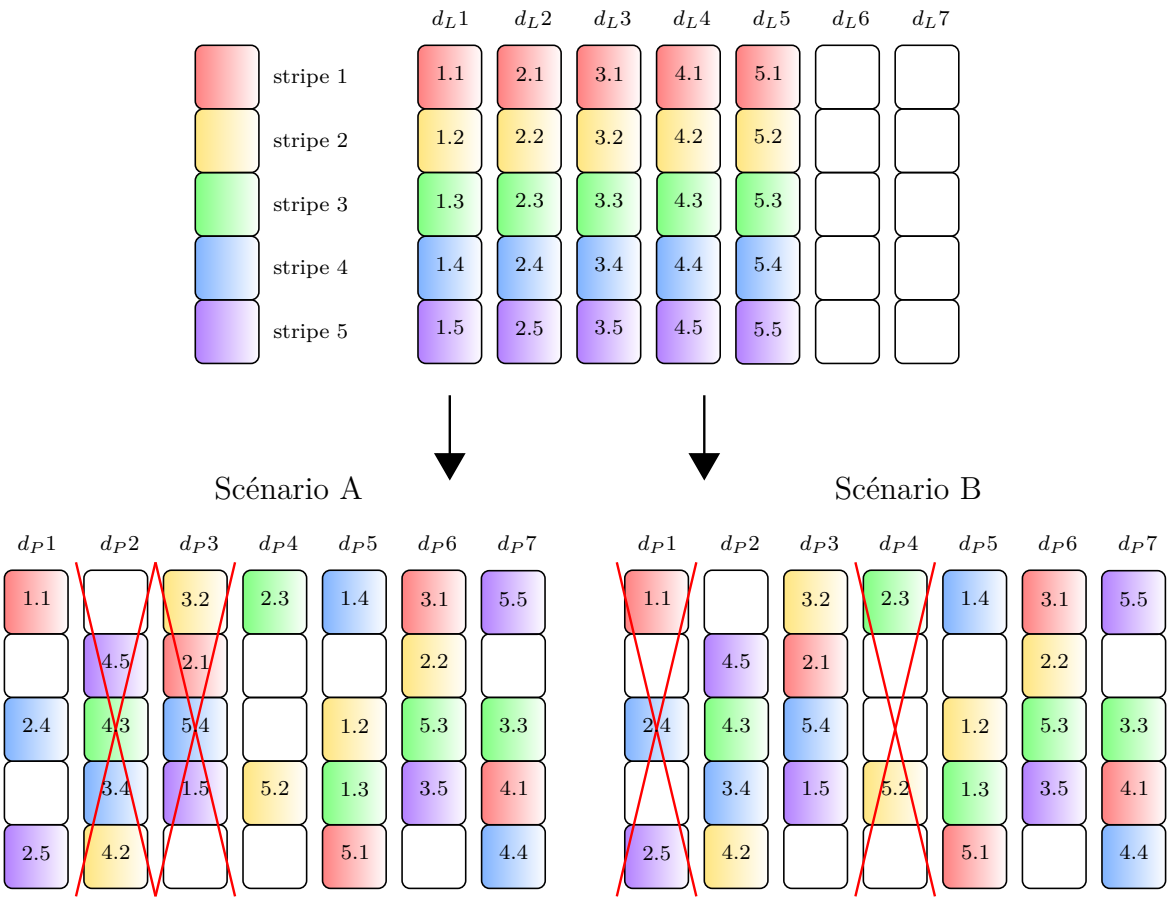



FIGURE 5.8 – Respect de la troisième contrainte

$$\forall(i,\, j) \in [1, n_d]^2,$$
$$(E_P(i) \cap E_P(j) \neq E_P(i)) \;\wedge\; (E_P(i) \cap E_P(j) \neq E_P(j))$$

### 4.4 Contrainte 4 : équilibrage de charge

L'équilibrage de charge consiste en la recherche d'un schéma de placement où les disques physiques possèdent le même nombre de blocs de données (réciproquement le même nombre de blocs libres). Cette contrainte permet de répartir au mieux les accès de manière uniforme sur les disques. La figure 5.9 montre un exemple de schéma pour lequel les disques ont des charges équilibrées. Notre algorithme SD2S vérifie cette contrainte mais peut la relâcher si besoin afin de privilégier l'optimisation de la robustesse du système. Cette contrainte est formalisée comme suit :

$$\forall(i,\, j) \in [1, n_d]^2,\ |\text{card}\ (E_p(i)) - \text{card}\ (E_p(j))| \leq 1$$

## 5 Algorithme naïf

Suite à l'établissement des contraintes de placement, nous avons élaboré une première version de notre algorithme SD2S, appelée algorithme naïf que nous décrivons dans l'algorithme 11. Cet algorithme va optimiser chaque placement de bloc en choisissant aléatoirement une position parmi toutes celles respectant l'ensemble des quatre

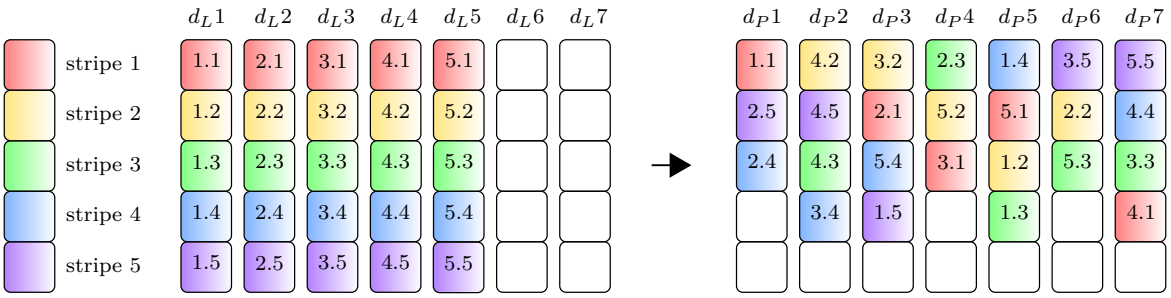



FIGURE 5.9 – Respect de la quatrième contrainte

contraintes précédentes. Ainsi, pour chaque bloc de données à placer, une sélection des disques est faite en fonction des contraintes 1, 2 et 4. Pour optimiser l'équilibrage de charge des blocs sur les disques, sont sélectionnés les blocs libres des disques possédant le moins de blocs utiles. Une fois la sélection faite, la position du bloc de données est choisie aléatoirement parmis les blocs libres retenus.

La création du schéma ne s'opère pas sur le système de stockage en entier mais sur quelques *stripes*. Cet ensemble de *stripes* est appelé période car le schéma pour cet ensemble se répète de manière cyclique, couvrant tout le système. L'utilisation des périodes permet de minimiser le temps de création du schéma de placement, car cela engendre moins de *stripes* à traiter, tout en assurant le respect des contraintes, dans la mesure du possible. La répétition de la période n'a aucun impact sur les contraintes 2 et 3 qui se focalisent sur la provenance logique en terme de *stripe* des données. La contrainte 4 sur l'équilibrage de charge est toujours respectée : si le nombre de blocs de données est identique sur chaque disque pour une période donnée, l'ajout d'autres périodes apporte un nombre de blocs de données identiques également. La contrainte 1 assure que chaque disque physique ne possède pas plus d'un bloc de données issu d'un même disque logique. Cette contrainte a un sens si le nombre de *stripes* est semblable au nombre de disques. Dans le cas, représentatif des systèmes réels, où le nombre de *stripes* est largement supérieur, on va chercher à minimiser le nombre de blocs de données d'un disque physique provenant d'un même disque logique, et donc ici, ce nombre est égal au nombre de périodes.

A la suite de cette sélection, le calcul des ensembles de provenance des disques physiques est fait pour déterminer si la troisième contrainte est respectée. Si elle ne l'est pas, alors une nouvelle sélection des positions est effectuée. Dans le cas où aucun schéma respectant la troisième contrainte n'est trouvé après un nombre défini de tentatives (par exemple un millier), un schéma de placement est adopté, selon lequel les positions physiques des blocs sont identiques aux positions logiques.

Le côté négatif de cette méthode naïve est la possibilité de ne pas aboutir à un schéma respectant les quatre contraintes. Effectivement, le côté aléatoire de la sélection parmi les positions éligibles peut amener à ne plus obtenir de positions éligibles pour les derniers blocs. La probabilité de ne pas obtenir de schéma optimal, même après un millier d'itérations n'est donc pas nulle.

Le tableau 5.4 montre une série d'expérimentations sur la méthode naïve. Le sous-tableau 5.4a montre les systèmes testés, allant de 8 à 20 disques avec un nombre de volumes et de disques *spare* variant. Le tableau 5.4b montre les résultats obtenus,

**Algorithme 11 :** Algorithme naïf – sélection des positions de blocs logiques

```
1  pour chaque bloc logique de données b_L de la période faire
2      E ← ∅ : ensemble des blocs physiques libres vérifiant les contraintes
3      min ← 0
4      pour chaque disque physique faire
5          si le disque possède déjà un bloc du même disque logique alors
6              continuer
7          fin
8          si le disque possède déjà un bloc du même stripe logique alors
9              continuer
10         fin
11         si le nombre de blocs libres du disque < min alors
12             continuer
13         fin
14         si le nombre de blocs libres m du disque > min alors
15             min ← m
16             E ← ∅
17         fin
18         pour chaque bloc libre b du disque physique faire
19             E ← E ∪ b
20         fin
21     fin
22     b_P ← alea (E)
23     donnees (b_P) ← donnees (b_L)
24 fin
```

pour des séries de 20 exécutions, en terme de nombre moyen de schémas générés avant l'obtention d'un schéma optimal. Il est également renseigné le pourcentage de réussite d'obtention du schéma optimal avant que le nombre de schémas générés ne dépasse le seuil fixé à 50'000 itérations. On peut observer que pour des systèmes très simples, un schéma optimal est toujours trouvé en peu de tentatives. Avec la complexité du système, en nombre de disques et de configurations, le nombre d'essais effectués avant d'obtenir le schéma optimal augmente radicalement. Ceci est dû à la sélection aléatoire dans l'algorithme.

De plus, le temps de création du schéma est long, car on utilise $3 \times n_d^2$ comparaisons par bloc à placer. Comme il y a environ $n_d^2$ blocs à placer, l'algorithme détermine un schéma de placement au terme de $3 \times n_d^4$ comparaisons. Pour les processeurs actuels possédant une fréquence de l'ordre du gigahertz, et en supposant un système de stockage composé d'une centaine de disques, cela donne un temps d'exécution de cet algorithme de l'ordre de la seconde par itération. Or, nous sommes généralement amené à itérer des dizaines de milliers de fois, sans compter le calcul du temps de vérification de la troisième contrainte.

L'ensemble de ces remarques nous ont donc poussé à proposer un algorithme plus rapide pour le calcul du schéma de placement optimal.

| Nombre de disques | 1 volume | | 2 volumes | | 3 volumes | |
|---|---|---|---|---|---|---|
| | Disques par volume | Disques spare | Disques par volume | Disques spare | Disques par volume | Disques spare |
| 8 | 6 | 2 | 3 | 2 | - | - |
| 10 | 7 | 3 | 4 | 2 | 3 | 1 |
| 12 | - | - | 5 | 2 | 3 | 3 |
| 14 | - | - | 5 | 4 | 4 | 2 |
| 16 | - | - | 6 | 4 | 5 | 1 |

(a) Systèmes testés

| Nombre de disques | 1 volume | | 2 volumes | | 3 volumes | |
|---|---|---|---|---|---|---|
| | Schémas générés | % de réussite | Schémas générés | % de réussite | Schémas générés | % de réussite |
| 8 | 39 | 100 | 741 | 100 | - | - |
| 10 | 136 | 100 | 14'004 | 60 | 49'335 | 5 |
| 12 | - | - | 45'282 | 10 | > 50'000 | 0 |
| 14 | - | - | > 50'000 | 0 | > 50'000 | 0 |
| 16 | - | - | > 50'000 | 0 | > 50'000 | 0 |

(b) Résultats – nombre moyen de schémas générés et pourcentage de réussite de création sur 20 exécutions

TABLE 5.4 – Résultats de la création de schéma de l'algorithme naïf

# 6 Conclusion

Le mode défaillant d'OGSSim permet la simulation de systèmes de stockage et leur confrontation face à la défaillance d'un ou plusieurs disques. L'intégration de ce mode se fait par l'ajout d'un type d'événement, appelé défaillance, générant des modifications de comportement des pilotes de volume et de disque principalement. En fonction de la configuration contenant le disque défaillant, les données sont soit reconstruites directement sur des emplacements libres, soit récupérées à l'aide des données stockées dans les disques sains à chaque demande d'accès de la part de l'utilisateur.

L'étude des différents algorithmes de placement des données pour des systèmes configurés en *declustered RAID* montre que les mécanismes de tolérance à la panne basés sur les réplicas sont largement utilisés dans les systèmes de stockage de données actuels. L'utilisation de multiples redondances intégrales à comme avantage de ne pas faire subir de perte de performances en cas de défaillances.

Les contraintes de placement que nous avons définies optimisent le parallélisme d'accès et la robustesse des systèmes de stockage. L'algorithme naïf que nous avons proposé, effectue la construction du schéma de placement en un temps important, générant plusieurs centaines voire milliers d'essais avant de trouver un schéma dit optimal.

# Chapitre 6

# Robustesse des systèmes à large échelle



*La robustesse est le paramètre déterminant la possibilité et l'efficacité de la reconstruction des données, suite à la défaillance d'un disque. Ce paramètre est d'autant plus important avec l'augmentation en nombre des disques composant les larges systèmes de stockage ainsi que la taille des dits composants. En effet, le nombre élevé de disques augmente la probabilité qu'une défaillance arrive et une taille importante du disque intervient directement sur l'allongement du temps de reconstruction des données stockées sur ce disque après une défaillance.*

*Comme décrit dans la chapitre 5, le declustered RAID est une organisation de disques ayant pour but la réduction du temps de reconstruction d'un disque en cas de défaillance. Il consiste en l'éclatement en blocs des données provenant de plusieurs volumes (JBOD ou RAID), et en leur répartition sur un système de stockage unique. C'est cette répartition qui est à l'origine du gain de temps, car au lieu d'effectuer la reconstruction de données à partir de blocs issus d'un seul volume (une quinzaine de disques en moyenne), les requêtes sont réparties sur autant de fois plus de disques, que le declustered RAID n'englobe de volumes.*

*Dans ce chapitre, nous proposons un nouvel algorithme de placement pour declustered RAID, appelé SD2S. Il se base sur l'ensemble de contraintes définies dans le chapitre 5, assurant la robustesse du système. Nous allons dans un premier temps présenter la partie majeure de notre algorithme, à savoir l'utilisation de matrices de degrés pour déterminer l'efficacité d'un schéma de placement. Puis nous détaillerons*

*les algorithmes de création de schémas de placement en mode normal et en mode défaillant. Nous terminerons par une présentation de différents résultats, montrant la performance et la robustesse de notre méthode SD2S.*

# 1 Proposition de l'algorithme SD2S

La méthode de placement que nous proposons a pour but d'offrir à un système de stockage, un schéma de placement alliant robustesse et performance. Elle est appelée Symmetric Difference of Source Sets algorithm (SD2S) et son principe de base repose sur l'indépendance entre les ensembles de provenance des blocs de données.

# 2 Matrice des degrés

La contrainte la plus critique pour la robustesse d'un système de stockage est celle concernant les ensembles de provenance des données. Il s'agit également de la plus difficile à respecter car elle ne peut être vérifiée qu'une fois le schéma de redirection établi. Nous allons définir un lexique propre au respect de cette contrainte, débouchant sur une méthode de détermination d'optimalité d'un schéma de placement orienté robustesse.

## 2.1 Définition d'un degré

Un **degré**, noté $deg$, est un couple de valeurs résultant de la comparaison entre deux ensembles de provenance. Elle exprime la disjonction des ensembles de provenance et leur différence de cardinal. Soient deux disques physiques $i$ et $j$ distincts, on a :

$$deg\ (i,j) = (\alpha, \beta) \tag{6.1}$$

$$\alpha = \min\ (\text{card}\ (E_i), \text{card}\ (E_j)) - \text{card}\ (E_i \cap E_j) \tag{6.2}$$

$$\beta = |(\text{card}\ (E_i) - \text{card}\ (E_j)| \tag{6.3}$$

Le paramètre $\alpha$ du degré donne le nombre minimal de blocs de données qui pourront être récupérés si une double défaillance des disques $i$ et $j$ intervient. Le paramètre $\beta$ indique si les ensembles de provenance possèdent un cardinal différent. Ce qui nous permet de mesurer la reconstruction des données sur les disques $i$ et $j$, notée $r_s$ en cas de double défaillance de ces deux disques qui est :

$$r_s\ (i,j) = \frac{2 \times \alpha + \beta}{\text{card}\ (E_i) + \text{card}\ (E_j)} \tag{6.4}$$

On peut vérifier avec cette équation les cas où les ensembles de provenance des deux disques $i$ et $j$ sont égaux, et où ils sont disjoints.

#### Cas des ensembles de provenance égaux

Si les deux ensembles sont égaux, alors on a :

$$\begin{aligned} E_i = E_j &\Rightarrow \text{card}\ (E_i) = \text{card}\ (E_j) \text{ et card } (E_i \cap E_j) = \text{card}\ (E_i) \\ &\Rightarrow \alpha = \text{card}\ (E_i) - \text{card}\ (E_i) = 0 \text{ et } \beta = 0 \\ &\Rightarrow r_s\ (i,j) = \frac{2 \times 0 + 0}{\text{card}\ (E_i) + \text{card}\ (E_j)} \\ &\Rightarrow r_s\ (i,j) = 0 \end{aligned}$$

Par conséquent, la fraction des données reconstruites est nulle, et effectivement, aucune donnée ne peut être reconstruite.

#### Cas des ensembles de provenance disjoints

Si les deux ensembles sont disjoints, alors on a, pour card $(E_i) \geq$ card $(E_j)$ :

$$\begin{aligned} E_i \cap E_j = \{\} &\Rightarrow \alpha = \text{card } (E_j) - 0 = \text{card } (E_j) \text{ et } \beta = \text{card } (E_i) - \text{card } (E_j) \\ &\Rightarrow r_s\ (i,j) = \frac{2 \times \text{card } (E_j) + \text{card } (E_i) - \text{card } (E_j)}{\text{card } (E_i) + \text{card } (E_j)} \\ &\Rightarrow r_s\ (i,j) = \frac{\text{card } (E_i) + \text{card } (E_j)}{\text{card } (E_i) + \text{card } (E_j)} = 1 \end{aligned}$$

Par conséquent, la fraction des données reconstruites obtenue est maximale. Toutes les données peuvent donc être reconstruites.

Nous introduisons alors le terme de **classe** de degré, catégorisant les valeurs en fonction de la sauvegarde possible de données en cas de multiple défaillance. La classe d'un degré est notée $C_d$ et vaut pour chaque $deg\ (i,j)$ :

$$C_d\ (deg\ (i,j)) = \begin{cases} 0 \text{ si } \alpha = 0 \text{ et } \beta = 0 \\ 1 \text{ si } \alpha = 0 \text{ et } \beta > 0 \\ 2 \text{ si } \alpha > 0 \end{cases}$$

## 2.2 Construction de la matrice des degrés

A partir des degrés définis précédemment, nous allons établir une matrice des degrés $M_{deg}$ avec pour indice de ligne et de colonne les disques du systèmes. La taille de la matrice est donc de $n_p^2$. La matrice est construite comme suit :

$$M_{deg}\ (i,j) = deg\ (i,j) \tag{6.5}$$

La diagonale de cette matrice ne contient pas de valeur, car n'aurait pas de sens sinon. En effet, une défaillance ciblant deux fois le même disque n'a pas plus d'impact qu'une seule, et n'a donc pas d'intérêt ici. La table 6.1 montre un exemple de matrice des degrés obtenue à partir du schéma de placement représenté sur la figure 6.1. On peut constater dans la matrice que l'un des degrés est de classe 0, ce qui indique qu'en cas de double défaillance ciblant ces deux disques, la totalité des données stockées dans ces disques sera perdue.

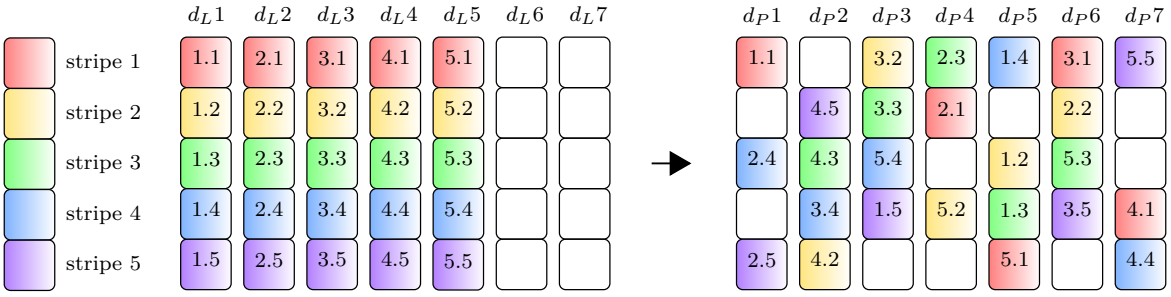



FIGURE 6.1 – Exemple de schéma de placement

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | - | (1, 1) | (1, 1) | (2, 0) | (1, 1) | (1, 1) | (0, 0) |
| 2 | (1, 1) | - | (0, 0) | (1, 1) | (1, 0) | (1, 0) | (1, 1) |
| 3 | (1, 1) | (0, 0) | - | (1, 1) | (1, 0) | (1, 0) | (1, 1) |
| 4 | (2, 0) | (1, 1) | (1, 1) | - | (0, 1) | (0, 1) | (2, 0) |
| 5 | (1, 1) | (1, 0) | (1, 0) | (0, 1) | - | (1, 0) | (1, 1) |
| 6 | (1, 1) | (1, 0) | (1, 0) | (0, 1) | (1, 0) | - | (1, 1) |
| 7 | (0, 0) | (1, 1) | (1, 1) | (2, 0) | (1, 1) | (1, 1) | - |

TABLE 6.1 – Matrice des degrés du système de la figure 6.1

On définit alors, de manière équivalente aux degrés, la classe d'une matrice, notée $C_M$ et qui vaut :

$$C_M = \min_{i,j} C_d \left(M_{deg} \left(i, j\right)\right) \tag{6.6}$$

En fonction de la classe de la matrice, on a donc les implications suivantes :

$$C_M = 0 \Rightarrow \exists (i, j) \in [0, n_d[^2 \text{ tel que } r_s \left(i, j\right) = 0$$

$$C_M \geq 1 \Rightarrow \forall (i, j) \in [0, n_d[^2, r_s \left(i, j\right) > 0$$

La différence entre les matrices de classe 1 et 2 est que les schémas de placement impliquant une matrice des degrés de classe 2 assure une reconstitution des données des deux disques défaillants quelque soit la double défaillance, c'est à dire qu'importe les indices des deux disques défaillants. Si la matrice est de classe 1, pour au moins une double défaillance, la reconstruction des données n'est possible que pour un seul des deux disques.

Avec l'utilisation de cette matrice des degrés, nous pouvons sélectionner les schémas de placement calculés pour assurer la reconstruction d'une fraction des données en cas de double défaillance, et donc respecter la contrainte sur les ensembles de provenance des données (contrainte 3).

# 3 Algorithme SD2S

Outre le besoin de robustesse du système qui doit être garanti par le schéma de placement, notre autre objectif est de proposer un algorithme de création de schéma de manière rapide, afin de ne pas pénaliser la performance pour autant. Après avoir fixé les différentes contraintes à respecter et proposé un premier algorithme satisfaisant l'ensemble des contraintes, nous avons observé que sélectionner le placement de chaque bloc de données selon ces contraintes était consommateur de temps.

Pour remédier à cela, on s'est orienté pour la sélection du placement vers un schéma satisfaisant les contraintes 1, 2 et 4, et de vérifier s'il respecte la troisième contrainte grâce à l'utilisation de la matrice des degrés. Pour une plus grande rapidité de calcul, nous avons également fait le choix de diviser le système de stockage en périodes, et de

trouver un schéma optimal pour une période, qui sera ensuite répété pour l'ensemble du système. La taille de cette période est de $(n_p - n_s)$, qui n'est que le nombre de disques logiques de donnée.

## 3.1 Le fonctionnement en mode normal

Nous avons fait le choix de créer nos schémas de placement à partir d'opérations modulo. Le schéma de placement effectue un décalage horizontal d'une valeur notée $\lambda$ aux blocs de données d'un *stripe*. Un exemple est donné figure 6.2 pour $\lambda = 2$.

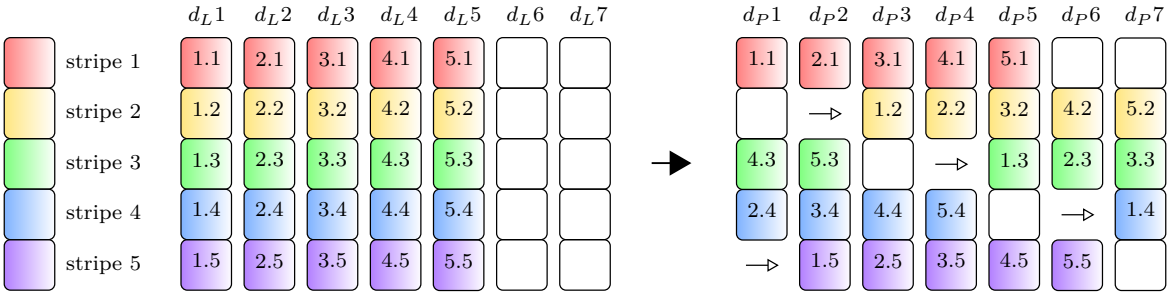



FIGURE 6.2 – Exemple de placement avec décalage à gauche ($\lambda = 2$)

Cette construction de schéma par décalage nous assure automatiquement la satisfaction de la seconde contrainte, basée sur le parallélisme des *stripes*, car le décalage est horizontal : les blocs appartenant à un même *stripe* logique feront partie du même *stripe* physique. La première contrainte, sur le parallélisme des disques, peut être satisfaite également dans le cas où le $\lambda$ choisi est premier avec le nombre de disques physiques du système. Dans ce cas, les blocs prendront toutes les positions disponibles dans $[1, n_p]$ pour un cycle de $n_p$ opérations de décalage.

Le respect de la dernière contrainte, sur l'équilibrage de charge, n'est par contre pas observé dans tous les cas lors de la construction, et dépend directement du nombre $n_s$ de disques libres.

La méthode SD2S (Symmetric Difference of Source Sets) est la combinaison de l'utilisation d'un schéma de placement avec décalage et du calcul de la matrice des degrés. Elle est décrite dans l'algorithme 12 qui est découpé en deux parties. Tout d'abord, pour satisfaire la première contrainte, nous recherchons la première valeur de $\lambda$ première avec $n_p$ donnant une classe de matrice $C_M$ optimale. Si aucun $\lambda$ n'est trouvé, nous ignorons alors le respect de la première contrainte, qui n'a pas d'impact sur la robustesse du système, pour se focaliser sur les $\lambda$ non premiers avec $n_p$.

## 3.2 Le fonctionnement en mode défaillant

Lorsqu'une défaillance survient, le système va être amené, s'il en est capable, à reconstruire les données perdues. Dans le cas du declustered RAID, les blocs de données issus du disque défaillant sont réécrits sur des emplacements libres des disques sains.

**Algorithme 12 :** Création du schéma en mode normal

**Entrées :** $n_p$ – nombre de disques physiques
$n_s$ – nombre de disques libres
**Sorties :** $\lambda$ – pas du décalage

1 **pour** $\lambda$ *allant de* 1 *à* $n_p$ **faire**
2 | **si** $i$ *n'est pas premier avec* $n_p$ **alors**
3 | | **continuer**
4 | **fin**
5 | $E \leftarrow$ ensembleProvenance $(n_p, n_s, \lambda)$
6 | $M \leftarrow$ matriceDegres $(E)$
7 | $C_M \leftarrow$ classeMatrice $(M)$
8 | **si** $C_M = 2$ **alors**
9 | | **retourner** $\lambda$
10 | **fin**
11 **fin**
12 **pour** $\lambda$ *allant de* 1 *à* $n_p$ **faire**
13 | **si** $i$ *est premier avec* $n_p$ **alors**
14 | | **continuer**
15 | **fin**
16 | $E \leftarrow$ calculEnsembleProvenance $(n_p, n_s, \lambda)$
17 | $M \leftarrow$ calculMatriceDegres $(E)$
18 | $C_M \leftarrow$ calculClasseMatrice $(M)$
19 | **si** $C_M = 2$ **alors**
20 | | **retourner** $\lambda$
21 | **fin**
22 **fin**

La robustesse étant la priorité de notre méthode SD2S, nous avons opté d'assurer au mieux la fiabilité du système en cas d'une reconstruction. A chaque reconstruction, le nombre de blocs libres dans le declustered RAID diminue, ce qui a pour effet d'ajouter des éléments dans les ensembles de provenance des disques. Cela va également baisser la probabilité d'avoir des ensembles de provenance disjoints deux à deux. De ce fait, et pour optimiser la robustesse du système, nous basons le schéma de reconstruction sur la seconde contrainte. Un bloc de données devant être reconstruit ne changera pas de *stripe* physique, ceci assurant qu'un disque physique possèdera toujours au plus un seul bloc de données provenant d'un même *stripe* logique.

Le choix du bloc cible pour stocker la donnée reconstruite se fait alors comme indiqué dans l'algorithme 13. On recherche un bloc libre placé sur le même *stripe* qui appartient, si possible, à un disque ne possédant pas de blocs faisant partie du même disque logique. Cette condition permet de maximiser le parallélisme d'accès aux disques.

Pour chaque disque défaillant, un vecteur de $(n_p - n_s)$ indices est créé, appelé vecteur de redirection. Pour chaque *stripe* de la période, l'indice correspondant du vecteur donne le disque physique possédant les données reconstruites.

**Algorithme 13 :** Création du schéma en mode défaillant

**Entrées :** $i$ – indice du disque défaillant
$n_d$ – nombre de disques
$n_f$ – nombre de disques libres

1 **pour** $j$ *allant de* 1 *à* $(n_p - n_s)$ **faire**
2  **si** *le bloc de données* $j$ *est un bloc libre* **alors**
3  **continuer**
4  **fin**
5  **pour** $k$ *allant de* 1 *à* $n_p$ **faire**
6  $E \leftarrow$ ensembleProvenance $(n_p, n_s, \lambda)$ **si** $E_k$ *ne contient pas la provenance du bloc* $j$ *et que le bloc* $j$ *du disque* $k$ *est libre* **alors**
7  le bloc de données est reconstruit sur le stripe $j$ du disque $k$
8  **continuer**
9  **fin**
10  **fin**
11  le bloc de données est reconstruit sur le premier disque rencontré dont le bloc du stripe $j$ est libre
12 **fin**

# 4 Résultats

Nous évaluons dans cette section divers résultats issus de la comparaison entre notre algorithme SD2S et la méthode Crush (Weil et al., 2006a) largement utilisée dans les systèmes de fichiers modernes tels que Ceph (Weil et al., 2006b). La version de Crush utilisée ici est implémentée sans réplicas. L'évaluation est découpée en trois parties : le coût en espace mémoire, le temps de calcul nécessaire à son exécution et le comportement des requêtes de reconstruction suite à une défaillance. Un dernier résultat sera présenté, discutant de la fraction des données sauvegardées en cas de multiples défaillances.

## 4.1 Coût spatial

Nous étudions le fonctionnement du système dans les deux modes, le mode dit normal sans défaillance et le mode défaillant avec au moins une défaillance de disque.

a) **Mode normal** : la méthode Crush a besoin de stocker la traduction logique/-physique de chacun des blocs de données dans une table. Le coût lié à l'utilisation de cette table est de $(n_p \times n_b)$ adresses. Quant à SD2S, il utilise des opérations modulo pour traduire les adresses logiques en physiques et n'a pas besoin de stocker les correspondances. De plus, l'opération modulo étant une opération de base, elle est simple et rapide à l'exécution.

b) **Mode défaillant** : notre algorithme SD2S utilise un vecteur de taille $(n_p - n_s)$ adresses pour stocker les correspondances des blocs reconstruits. La taille de la table de traduction pour la méthode Crush ne change pas, mais reste tout de même supérieure à la taille utilisée par SD2S. Dans le pire cas où le système subit $n_s$ défaillances, SD2S a un coût mémoire de $(n_s \times (n_p - n_s))$ adresses.

La table 6.2 résume le coût en espace mémoire des deux méthodes. Le facteur donné dans la table représente le rapport entre le coût en mémoire pour SD2S et celui de

Crush.

| Méthode | SD2S | Crush | F = (SD2S / Crush) |
|---|---|---|---|
| Mode normal | 0 | $n_p \times n_b$ | 0 |
| 1 défaillance | $(n_p - n_s)$ | $n_p \times n_b$ | $\frac{n_p - n_s}{n_p \times n_b}$ |
| $n_s$ défaillances | $n_s \times (n_p - n_s)$ | $n_p \times n_b$ | $\frac{n_s \times (n_p - n_s)}{n_p \times n_b}$ |

TABLE 6.2 – Coût spatial des méthodes SD2S et Crush

L'ordre de grandeur de chaque paramètre est indiqué dans la table 6.3. Le facteur pour une seule défaillance est donc de $10^{-6}$ et de $10^{-5}$ pour $n_s$ défaillances. En conclusion, pour le nombre de défaillances subies par le système, la méthode SD2S génère un coût en espace mémoire plus faible que Crush.

| Paramètre | Ordre de grandeur |
|---|---|
| $n_p$ | $10^2$ |
| $n_s$ | $10^1$ |
| $n_b$ | $10^5$ |

TABLE 6.3 – Ordre de grandeur des paramètres du système

**Exemple :**

Supposons un système de stockage très large, composé de 1000 disques, comme c'est le cas actuellement, dont une vingtaine sont des disques libres destinés à la reconstruction. On suppose également que le nombre de blocs par disques est de $10^5$. Crush utilisera une table de traduction logique/physique composée de $10^8$ adresses. En mode normal, SD2S n'utilisera aucun espace mémoire pour la traduction. Ce qui signifie que nous obtenons un gain de 100 millions d'adresses stockées.

Après une défaillance, SD2S créera une table de 980 entrées, arrondi à $10^3$. SD2S va donc utiliser $10^5$ fois moins d'espace mémoire pour stocker sa table de traduction. Dans le cas où le nombre maximal de défaillances est subit par le système, soit un nombre de fois égal à $n_s$, la table de traduction de SD2S a besoin de stocker $2 \times 10^4$ adresses. Le facteur de différence entre les deux méthodes passe alors à $5 \times 10^{-3}$ au profit de SD2S.

## 4.2 Coût temporel

Pour l'étude du coût temporel associé à notre méthode SD2S, nous allons comparer le temps que prennent à la fois la construction et la reconstruction du schéma de placement suite à une défaillance, toujours relativement à la méthode Crush. La version de Crush utilisée pour la comparaison est sans réplicas, ce qui signifie que les données ne sont présentes dans le système qu'en un seul exemplaire.

Pour cette série d'expérimentations, nous allons supposer un système de type declustered RAID de 20 disques Hitachi A7K1000 (HGST, 2007), composé de 3 configurations et de 3 disques libres, destinés à recevoir des données issues de la reconstruction. Les configurations sont un RAID-1, un RAID-01 et un RAID-NP. Elles sont composées respectivement de 4, 8 et 5 (dont 1 de parité) disques.

La figure 6.3 montre la comparaison pour la construction du schéma de placement. L'ordonnée indique le temps utilisé pour la création du schéma, et l'abscisse représente différents scénarii de défaillance. Etant donné que la défaillance d'un disque intervient après la construction du schéma, on peut observer que le scénario n'a pas d'impact sur le temps de simulation. SD2S présente un temps de construction 1000 fois inférieur à Crush. Ceci est dû en majeure partie à la fenêtre de *stripes* concernés par la création. Là où Crush crée son schéma pour le système entier, SD2S ne le fait que pour une fraction égale à $(n_p - n_s)/n_b$ du système, appelée période, liée à notre stratégie de placement par décalage.

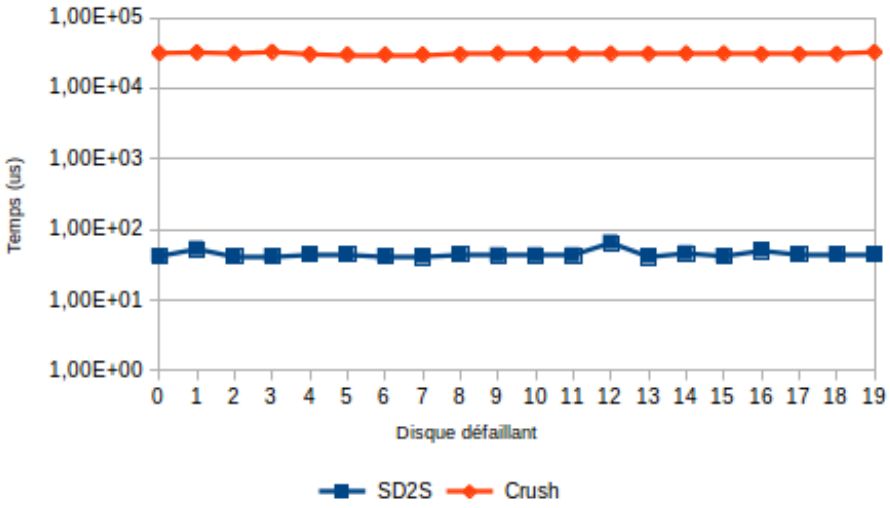



FIGURE 6.3 – Temps de simulation de la création du schéma de **construction**

Dans notre expérimentation, nous utilisons des disques d'une capacité de 500 Go, découpés en blocs de 8 Mo, ce qui donne environ 60 000 blocs de données pour un disque. La fraction du système qui représente la période et sur laquelle SD2S se focalise n'est que de 340 blocs de données, soit 200 fois moins de blocs.

Les temps de création du schéma de reconstruction, suite à une défaillance, sont donnés figure 6.4. Les repères sont les mêmes que pour le graphe précédent. Les résultats sont cette fois-ci moins stables que pour le schéma de construction. Ceci est directement lié à l'identité du disque qui devient défaillant, et qui ne possède pas forcément le même nombre de blocs de données que les autres. L'écart entre les deux méthodes est d'un facteur 3 en faveur de SD2S. L'écart est ici moins important que pour le temps de création du schéma de construction de par notre recherche de placement optimal pour chaque bloc perdu à cause de la défaillance. Lors de l'établissement du schéma de construction, nous ne cherchons pas à optimiser le placement des blocs un par un, puisqu'il est calculé à l'aide d'opérations modulo.

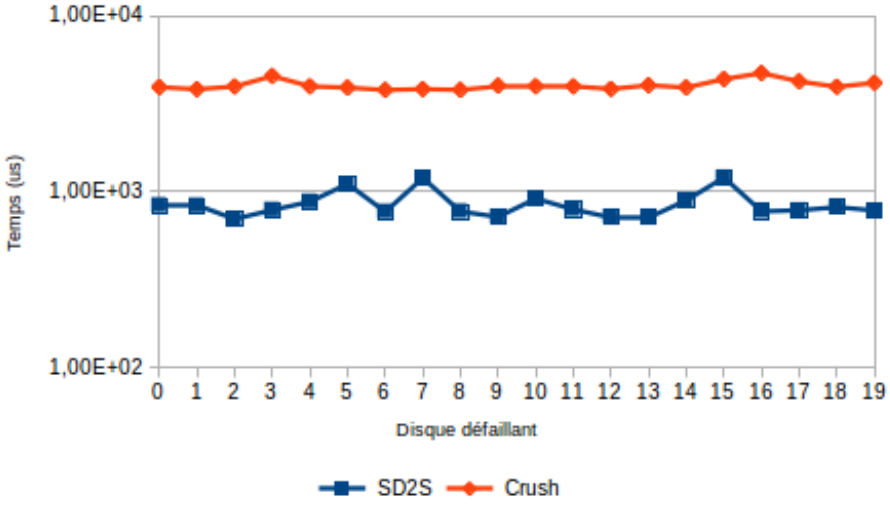



FIGURE 6.4 – Temps de simulation de la création du schéma de **reconstruction**

La dernière expérimentation concernant la création des schémas va porter sur la taille de la configuration architecturale. Nous faisons varier le nombre de disques, et la composition du système, comme indiqué dans le tableau 6.4.

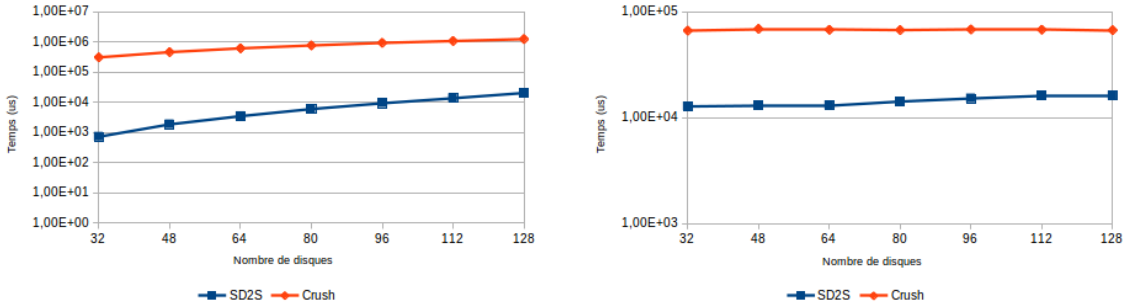
| Nombre de disques | Nombre de volumes | Nombre de disques *spare* |
|---|---|---|
| 32 | 4 | 4 |
| 48 | 6 | 6 |
| 64 | 8 | 8 |
| 80 | 10 | 10 |
| 96 | 12 | 12 |
| 112 | 14 | 14 |
| 128 | 16 | 16 |

TABLE 6.4 – Configurations pour l'expérimentation de la figure 6.5

Les résultats sont présentés sur la figure 6.5. De manière générale, les temps de création de schéma sont proches de ce que nous avions vu avec l'expérimentation précédente, la variation de la taille du système n'ayant que peu d'impact. On observe néanmoins pour le schéma de construction (figure 6.5a) que le rapport entre le temps de création pour SD2S et celui pour Crush est d'environ 100 pour 128 disques, alors qu'il était de 1000 pour 20 disques. Pour ce qui est du schéma de reconstruction (figure 6.5b, on observe que le temps utilisé par Crush reste stable tandis que celui de SD2S a tendance à croître faiblement. Ce résultat est logique : d'un côté, Crush effectue quelque soit le nombre de disques du système, un calcul de placement pour chaque bloc de données se trouvant sur le disque défaillant, de l'autre SD2S, ayant une période de taille équivalente au nombre de disques du système, le nombre de placements à effectuer s'aggrandit. Malgré ce constat, SD2S reste 4 fois plus rapide que Crush pour la création du schéma de reconstruction.

## 4.3 Comportement des requêtes de reconstruction

Les expérimentations sont effectuées sur le même système de stockage présenté dans la section 4.2. Le schéma de placement est optimal pour un pas de décalage




(a) Schéma de construction (b) Schéma de reconstruction

FIGURE 6.5 – Temps de simulation de la création des schémas et la taille du système

$\lambda = 1$. De même, chaque expérimentation est un scénario de défaillance ciblant l'un des disques du système.

La figure 6.6 montre pour chaque scénario le temps de réponse moyen et médian des requêtes générées pour la reconstruction des données du disque défaillant. On observe que dans la majorité des cas, le temps de réponse médian de notre méthode est nettement inférieur à celui de Crush. Pour quatre scénarii, la tendance est inverse, avec un avantage pour Crush, surtout marqué pour le seul scénario où le disque 10 est défaillant. La valeur moyenne suit la même tendance, avec néanmoins des écarts plus importants. Ces résultats montrent que pour une majorité des scénarii testés, le temps de réponse des requêtes est plus court en utilisant SD2S que Crush, et que seules quelques requêtes sont grandement impactées.

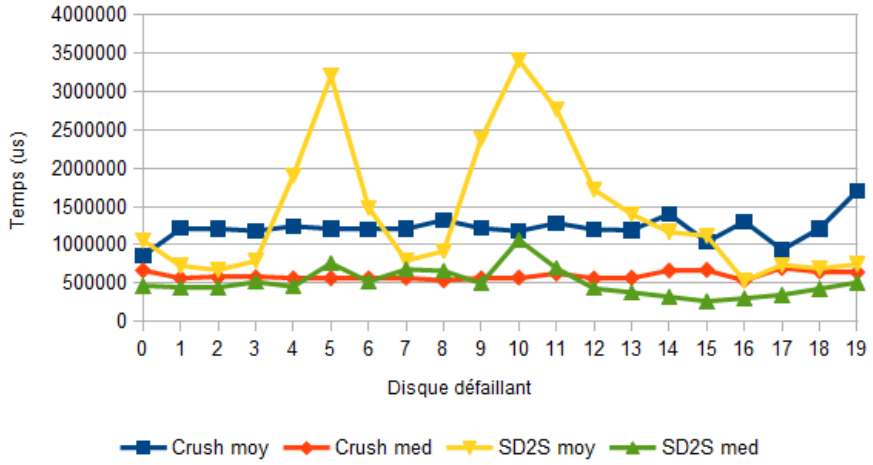
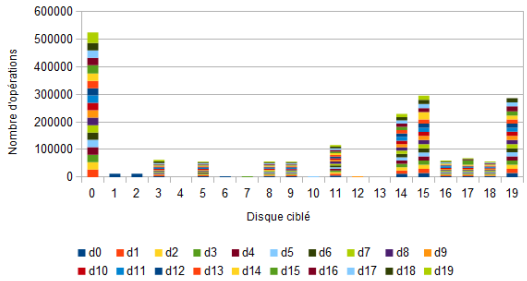



FIGURE 6.6 – Temps de réponse des requêtes de reconstruction

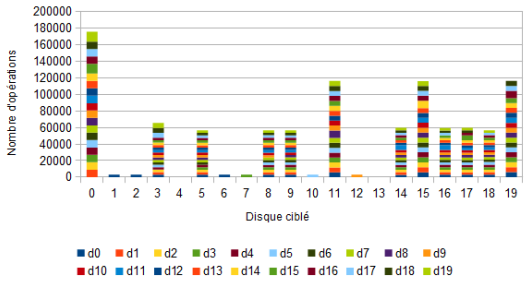
La seconde expérimentation présente l'équilibrage de charge des requêtes de reconstruction sur les disques du système. Les résultats sont montrés sur la figure 6.7. L'abscisse du graphe indique le disque ciblé par les requêtes de reconstruction, et les scénarios sont indiqués dans la légende. La première observation concerne l'équilibrage de charge. Dans la quasi totalité des scénarii, Crush utilise toujours les mêmes disques. Par exemple, le disque 0 est toujours sollicité en lecture et en écriture (sauf s'il est

défaillant) alors que d'autres, comme le disque 6 ne le sont jamais. A l'inverse, avec SD2S, les disques sont ciblés dans des proportions équivalentes.

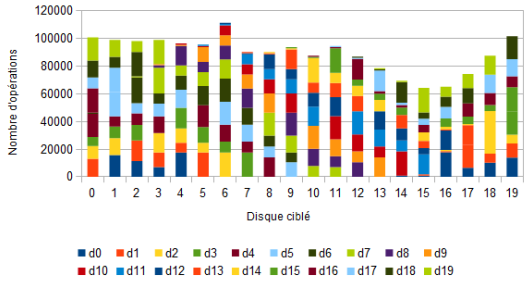
En portant le focus sur SD2S (figures 6.7c et 6.7d), on peut voir que pour un scénario donné, les requêtes en lecture vont cibler les disques proches du disque défaillant. Par exemple, si le disque 5 est défaillant, les requêtes de reconstruction en lecture seront à destination des disques 1 à 4 et 6 à 9. Ceci s'explique par la construction de notre schéma. Le pas de décalage étant égal à 1, les *stripe units* sont situés sur ces mêmes disques. Par contre, les requêtes en écriture sont elles réparties sur l'ensemble des disques, excepté les disques 17 à 19, ne comportant aucun *stripe unit spare*.



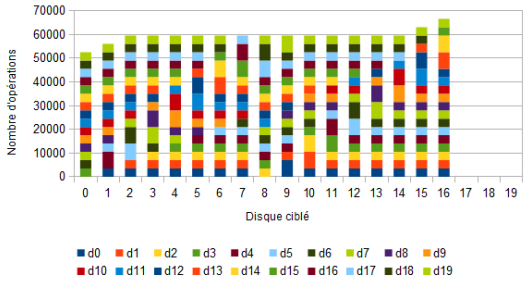

(a) Crush – requêtes en lecture




(b) Crush – requêtes en écriture




(c) SD2S – requêtes en lecture




(d) SD2S – requêtes en écriture

FIGURE 6.7 – Répartition des requêtes de reconstruction

## 4.4 Fraction des données sauvegardées

Dans un système ayant déjà subi une défaillance, l'arrivée d'une seconde défaillance est critique, car générant une perte de données dans la quasi totalité des cas. L'objectif est donc de minimiser cette perte, grâce au schéma de placement.

Supposons un *declustered RAID* utilisant un schéma de placement optimal généré par notre méthode SD2S. En considérant deux disques physiques $d_{Pi}$ et $d_{Pj}$, avec deg $(i, j) = (\alpha, \beta)$, on a forcément $\alpha > 0$, étant donné que le schéma de placement est optimal. On suppose que $d_{Pi}$ devient défaillant à $T = t_1$ et $d_{Pj}$ à $T = t_2$. Nous considérons alors deux cas pour $\Delta T = t_2 - t_1$ :

**Cas où $\Delta T = 0$**

Si les deux défaillances arrivent au même moment, la part $r$ de données sauvées est égale à :

$$r = \frac{(2\alpha + \beta)}{2 \times (n_d - n_f)} \tag{6.7}$$

Le numérateur de l'équation correspond au nombre de blocs de données ne se trouvant que sur un seul des deux disques défaillants et le dénominateur représente le nombre total de blocs de données.

**Cas où $\Delta T > 0$**

Si la seconde défaillance intervient après la première, mais avant que le mécanisme de reconstruction ne soit achevé, alors la part $r'$ de données sauvées devient :

$$r' = r + (1 - r) \times frd \tag{6.8}$$

avec $frd$ la part de données reconstruites avant l'arrivée de la seconde défaillance.

## 4.5 Bilan

Pour résumer, les coûts aussi bien spatial que temporel associés à la méthode SD2S sont plus faibles que ceux de Crush grâce à une construction de schéma de placement réduite à une période, et à une traduction logique/physique basée sur des opérations modulo, évitant l'utilisation d'une table de traduction en mode normal. En mode défaillant, l'espace mémoire utilisé par le vecteur de redirection de SD2S est très inférieur à celui utilisé par Crush, d'un facteur $10^{-6}$ pour une défaillance et d'un facteur $10^{-5}$ pour $n_s$ défaillances.

Le mécanisme de reconstruction de notre méthode effectue un meilleur équilibrage de charge sur les disques sains que la méthode Crush, et le temps de réponse médian est plus faible avec SD2S. Enfin, nous assurons une fraction des données sauvegardées non nulle, quelque soit la double défaillance affectant le système de stockage des données.

# 5 Conclusion et perspectives

La méthode SD2S proposée dans ce chapitre offre un coût spatial et temporel réduit comparé à la méthode Crush, actuellement utilisée dans des systèmes de fichiers distribués (Red Hat Inc, 2016). Notre méthode utilise des opérations modulo, rapides à effectuer, pour gérer la traduction logique/physique des données. Ces opérations sont basées sur le décalage avec un pas $\lambda$, déterminé suite à la construction de matrices des degrés, déterminant l'optimalité d'un schéma de placement. De plus, l'utilisation d'un schéma basé sur le décalage des *stripe units* nous offre un meilleur équilibrage de charges lors du mécanisme de reconstruction des données.

La fraction de données pouvant être reconstruites suite à une double défaillance dépend de la proximité des ensembles de provenance des disques physiques, ie. le nombre d'éléments qu'ils ont en commun. L'une des perspectives de recherche qui font suite à la proposition de cette méthode, est de chercher à maximiser cette fraction de données reconstruites en sélectionnant la matrice des degrés sur des critères d'optimalité plus restreints. Par exemple, si en plus d'éviter les ensembles égaux, nous cherchions à maximiser la médiane des degrés présents dans la matrice, nos couples d'ensembles auront tendance à posséder moins d'éléments en commun.

Cette recherche d'optimalité présenterait un coût temporel de construction plus grand qu'avec la version actuelle. L'objectif sera donc de trouver le bon compromis entre le surcoût qu'aurait le temps d'exécution, et l'augmentation de la fraction de données reconstruites.

L'autre perspective de recherche est d'étendre le travail de comparaison, en mesurant notre algorithme SD2S à une version de Crush avec l'utilisation des réplicas, point fort de la méthode.

# Conclusion

Face à l'augmentation de la taille et la complexité technologique des centres de données, simuler de tels systèmes de stockage afin d'en prédire le comportement et d'en analyser la performance et la robustesse devient une tâche compliquée. Les travaux réalisés dans cette thèse se concentrent sur la simulation de systèmes de stockage hétérogènes à large échelle, et à l'étude de leurs performances et de leur fiabilité.

## Contributions apportées

Nous distinguons dans cette thèse, deux contributions majeures :

1. la conception et le développement d'OGSSim : un outil de simulation ouvert et générique supportant des systèmes de stockage de données hétérogènes à large échelle.
2. la proposition, l'implémentation et l'évaluation de l'algorithme SD2S : un algorithme de placement de données pour configuration de stockage orientée robustesse, sans utilisation de réplicas.

### OGSSim

OGSSim est un logiciel composé actuellement de 17'000 lignes de code et possédant un schéma conceptuel modulaire, où chaque module tient un rôle précis dans le processus de simulation. Les données d'entrée du logiciel sont fournies à l'aide de fichiers XML, représentant le système de stockage simulé (les disques, les configurations et les bus utilisés), le jeu de requêtes associé au scénario de simulation et la configuration du processus de la simulation. Chaque module est instancié par au moins un *thread*, et les communications *inter-thread* sont assurés par ZeroMQ, une bibliothèque de communication par socket.

Les modules d'OGSSim sont répartis en trois chaînes distinctes. La chaîne d'extraction récupère des fichiers d'entrée l'ensemble des données utilisées pour la simulation, à savoir la configuration architecturale, le jeu de données et les événements externes comme par exemple les défaillances de disque. L'ensemble de ces données sont placés en mémoire partagée, et rendus disponibles pour tous les modules d'OGSSim. La chaîne de décomposition transforme toute requête logique en ensembles de sous-requêtes physiques, où chacune a pour cible les données d'un seul disque. La traduction s'effectue en trois étapes :

1. Redirection des requêtes logiques vers la configuration ciblée.
2. Décomposition des requêtes logiques en sous-requêtes, tel que chaque sous-requête cible des données contigües d'un seul disque.

3. Redirection de la sous-requête vers le disque ciblé et transformation en sous-requête physique.

La dernière chaîne de modules est la chaîne de calcul. Elle a la charge, par le biais de modèles de calcul, de calculer les métriques de performance de la simulation, comme par exemple le temps de réponse des requêtes, le taux d'utilisation des disques ou le nombre de requêtes échouées. Le module de visualisation créé, en fonction des besoins de l'utilisateur, divers graphes représentant ces métriques.

OGSSim implémente deux modes de fonctionnements : un mode normal, où le jeu de requêtes est simulé sans événements extérieurs et un mode défaillant, où une ou plusieurs défaillances peuvent intervenir. OGSSim est actuellement en phase de validation. Les modèles de calcul implémentés dans OGSSim ont été validés pour les SSDs, et en cours de validation pour les HDDs. La validation des configurations complètes est en discussion. OGSSim peut simuler des systèmes composés d'au moins 3'000 disques et supporte donc bien les systèmes à large échelle. Son temps d'exécution reste tout à fait raisonnable de l'ordre de la demi-heure pour des systèmes à 3'000 disques avec des RAIDs.

**SD2S**

SD2S est un nouvel algorithme de placement de données pour *declustered RAID*, basé sur le respect de quatre contraintes :

1. Parallélisme d'accès aux disques.
2. Parallélisme d'accès aux *stripes*.
3. Disjonction des ensembles de provenance des données.
4. Equilibrage de charge.

Ces quatre contraintes permettent dans un premier temps l'établissement d'un algorithme de création de schémas de placement de données et sa reconstruction quand une panne survient. En mode normal, le schéma consiste en un décalage des *stripes* d'un pas $\lambda$. Un $\lambda$ premier avec le nombre de disques physiques du système, assure le respect des deux premières contraintes. Ce $\lambda$ est sélectionné à la suite du calcul du rang de la matrice des degrés.

Un degré représente la distance entre les ensembles de provenance de deux disques physiques. Si le rang de la matrice des degrés est optimal, alors les ensembles de provenance sont tous partiellement disjoints deux à deux. Cette distance assure la reconstruction d'une partie non nulle des données en cas de double défaillance.

Le temps de création du schéma de placement avec SD2S est 1'000 fois inférieur à celui de Crush, sans réplicas. Cette différence s'explique par notre volonté de concentrer la création du schéma sur un *mini-RAID*, plutôt que sur la configuration entière. De plus, utilisant un schéma de placement à décalage, notre coût en espace mémoire est nul, tandis que les méthodes de type Crush utilise une table de redirection assez coûteuse en espace.

En mode défaillant, nous reconstruisons le schéma de placement en assurant le parallélisme d'accès aux *stripes*, évitant ainsi que deux *stripe units* d'un même *stripe* ne soit perdus en cas d'une future défaillance. La construction du schéma est 10 fois plus rapide avec SD2S que Crush, et notre coût en espace mémoire est, selon les ordres de grandeur présentés, $10^5$ fois inférieur.

# Perspectives et travaux futurs

Plusieurs perspectives se présentent à court et moyens termes, faisant suite aux contributions réalisées dans cette thèse.

### Validation d'OGSSim contre les HDDs et contre des systèmes réels

Des expérimentations complémentaires sur les HDDs sont en cours pour affiner notre modèle de calcul et représenter toute la richesse et la complexité de ces supports. L'un des principaux problèmes à surmonter est le manque de documentation sur l'architecture interne des disques. S'ensuivra la validation des systèmes plus complexes. Celle-ci nécessite l'accès à une configuration matérielle existante et pouvoir les paramétrer. Des discussions dans ce sens sont en cours.

### Amélioration de l'implémentation d'OGSSim

Cette amélioration portera sur le choix des structures de données utilisés pour le stockage des requêtes. La structure utilisée dans la version actuelle répond à une volonté de ne pas solliciter à outrance les appels systèmes pour la gestion de la mémoire. En contre-partie, elle nécessite l'emploi d'un mécanisme de synchronisation dans le cas où le nombre de sous-requêtes dépasse la taille de la structure. Une structure de taille dynamique, au lieu de fixe, permettra aux pilotes de volume d'ajouter autant de sous-requêtes que nécessaire, sans utilisation de mécanisme de synchronisation. Pour éviter la sur-utilisation des appels systèmes, les allocations seront faites millier par millier.

### Enrichissement de la gestion permanente

Cet enrichissement se fera par l'implémentation d'algorithmes de défragmentation, d'égalisation d'usure et de ramassage de miettes, comme par exemple les algorithmes Rejuvenator et Sequential Garbage Collection. L'ajout de ces algorithmes est facilité par l'implémentation de l'interface pour les modèles de gestion permanente. Ils serviront alors de référence pour la comparaison d'algorithmes futurs.

### Généricité du modèle de communication

L'utilisation d'un modèle générique permet d'adapter le mécanisme de communication à l'architecture utilisée pour la simulation. L'ensemble des fonctions de communication seront alors réunies au sein d'une même interface, et à chaque architecture supportée correspondra une implémentation de cette interface (ZeroMQ, MPI, etc.). Ce nouveau modèle est la première étape du portage d'OGSSim sur l'environnement parallèle multi-noeud.

### Parallélisation d'OGSSim

L'un des freins à la scalabilité du logiciel, et donc à la taille du système pouvant être simulée est la limite du nombre de threads pouvant être instanciés sur une machine. Effectivement, un thread est attaché à chaque pilote de disque. Le passage à un environnement multi-noeud permettra donc l'instanciation de plus de threads, et donc de pouvoir supporter plus de disques dans le système simulé.

### Optimisation de la matrice des degrés

La version actuelle de SD2S utilise une matrice des degrés calculée avec des ensembles de provenance partiellement disjoints. Optimiser cette matrice revient à chercher à maximiser le nombre de couples d'ensembles de provenance disjoints ou, si non possible, le nombre d'éléments différents permet de maximiser la part de données reconstruites suite à une double défaillance.

### Etude comparative avec des réplicas

Les réplicas sont la caractéristique principale de Crush, ce qui permet d'assurer la robustesse des données, et un accès sans dégradation de performances même en mode défaillant. Confronter cette version à notre méthode SD2S nous permettra d'obtenir un bilan complet et une comparaison plus juste car la force de Crush est l'utilisation des réplicas, même si cela implique un coût spatial significatif que SD2S n'a pas.

# Bibliographie


Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. (2008). Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA. USENIX Association.

AnandTech (2012). Samsung ssd 840 (250gb) review. `http://www.anandtech.com/show/6337/samsung-ssd-840-250gb-review/2`.

Apache Software Foundation (2016). Site web xerces-c. xerces.apache.org/xercesc-c/.

Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. (2015). Hard disk drives. In *Operating Systems : Three Easy Pieces*, chapter 37. Arpaci-Dusseau Books.

Balakin, A. (2016). Site web mathgl. mathgl.sourceforge.net.

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA. USENIX Association.

Bucy, J. S., Schindler, J., Schlosser, S. W., and Ganger, G. R. (2008). The disksim simulation environment version 4.0 reference manual.

Capps, J. (2002). System and method for defragmenting a file system. US Patent 6,397,311.

Chang, L.-P. (2007). On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 1126–1130, New York, NY, USA. ACM.

Chang, L.-P. and Kuo, T.-W. (2005). Efficient management for large-scale flash-memory storage systems with resource conservation. *Trans. Storage*, 1(4) :381–418.

Chen, F., Koufaty, D. A., and Zhang, X. (2009). Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA. ACM.

Chen, P. M. and Lee, E. K. (1995). Striping in a raid level 5 disk array. *SIGMETRICS Perform. Eval. Rev.*, 23(1) :136–145.

Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. (1994). Raid : High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2) :145–185.

Chiang, M.-L. and Chang, R.-C. (1999). Cleaning policies in mobile computers using

flash memory. *Journal of Systems and Software*, 48(3) :213 – 231.

Chung, C. C. and Hsueh, N. M. (2012). A low-complexity high-performance wear-leveling algorithm for flash memory system design. In *2012 IEEE Asia Pacific Conference on Circuits and Systems*, pages 595–598.

Cornwell, M. (2012). Anatomy of a solid-state drive. *Commun. ACM*, 55(12) :59–63.

Dau, S. H., Jia, Y., Xi, W., and Chan, K. S. (2014). Parity declustering for fault-tolerant storage systems via t-designs. In *the IEEE International Conference on Big Data (Big Data)*.

Deenadhayalan, V. (2011). Gpfs native raid for 100,000-disk petascale systems devops. In *the 25th Large Installation System Administration Conference (Usenix LISA)*.

El Maghraoui, K., Kandiraju, G., Jann, J., and Pattnaik, P. (2010). Modeling and simulating flash based solid-state disks for operating systems. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 15–26, New York, NY, USA. ACM.

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. (1979). Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3) :315–344.

Ganger, G. R., Worthington, B. L., and Patt, Y. N. (1998). The disksim simulation environment – version 1.0 reference manual.

Gibson, G. and Ganger, G. (2011). Principles of operation for shingled disk devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory.

Gibson, G. and Patterson, D. (1993). Designing disk arrays for high data reliability. *Journal of Parallel and Distributed Computing*, 17 :4–27.

Gleixner, T., Haverkamp, F., and Bityutskiy, A. (2006). Ubi - unsorted block images.

Goodson, G. and Iyer, R. (2010). Design tradeoffs in a flash translation layer. In *Proceedings of Workshop on the Use of Emerging Storage and Memory Technologies*.

Google (2016). Dépôt github de glog. github.com/google/glog.

Gougeaud, S., Zertal, S., Lafoucriere, J.-C., and Deniel, P. (2015a). Ogssim : Open generic data storage systems simulation tool. In *8th International Conference on Simulation Tools and Techniques (SIMUtools)*.

Gougeaud, S., Zertal, S., Lafoucriere, J.-C., and Deniel, P. (2015b). Ogssim : Open generic data storage systems simulation tool. *EAI Endorsed Transactions on Scalable Information Systems*, 16(9).

Gougeaud, S., Zertal, S., Lafoucriere, J.-C., and Deniel, P. (2016a). Block shifting layout for efficient and robust large declustered storage systems. In *14th International Conference on High Performance Computing and Simulation (HPCS)*.

Gougeaud, S., Zertal, S., Lafoucriere, J.-C., and Deniel, P. (2016b). A generic and open simulation tool for large multi-tiered hierarchical storage systems. In *International Symposium on Performance Evaluation of Computer and Telecommunication*

*Systems (SPECTS).*

Han, L., Ryu, Y., and Yim, K. (2006). Cata : A garbage collection scheme for flash memory file systems. In *Proceedings of the Third International Conference on Ubiquitous Intelligence and Computing*, UIC'06, pages 103–112, Berlin, Heidelberg. Springer-Verlag.

Harrison, P. G., Harrison, S. K., Patel, N. M., and Zertal, S. (2012). Storage workload modelling by hidden markov models : Application to flash memory. *Perform. Eval.*, 69(1) :17–40.

HGST (2007). Spécifications hitachi ultrastar a7k1000. `https://www.hgst.com/sites/default/files/resources/Ultrastar_A7K1000_final_DS.pdf`.

HGST (2015). Spécifications hgst ultrastar archive ha10. `https://www.hgst.com/sites/default/files/resources/Ha10-Ultrastar-HDD-DS.pdf`.

HGST (2016). Spécifications hgst travelstar z7k500. `http://www.hgst.com/sites/default/files/resources/TSZ7K500_ds_v3.pdf`.

Hill, K. (2013). Blueprints of nsa's ridiculously expensive data center in utah suggest it holds less info than thought. Forbes website.

Holand, M. and Gibson, G. (1992). Parity declustering for continuous operation in redundant disk arrays. In *the 5th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

Hu, Y., Jiang, H., Feng, D., Tian, L., Luo, H., and Zhang, S. (2011). Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 96–107, New York, NY, USA. ACM.

iMatix Corporation (2016). Site web zeromq. zeromq.org.

Jin, P., Su, X., Li, Z., and Yue, L. (2009). A flexible simulation environment for flash-aware algorithms. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 2093–2094, New York, NY, USA. ACM.

Jung, M., Choi, W., Gao, S., Wilson III, E. H., Donofrio, D., Shalf, J., and Kandemir, M. T. (2016). Nandflashsim : High-fidelity, microarchitecture-aware nand flash memory simulation. *Trans. Storage*, 12(2) :6 :1–6 :32.

Jung, M., Wilson, E., Donofrio, D., Shalf, J., and Kandemir, M. (2012). Nandflashsim : Intrinsic latency variation aware nand flash memory system modeling and simulation at microarchitecture level. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12.

Kawaguchi, A., Nishioka, S., and Motoda, H. (1995). A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 13–13, Berkeley, CA, USA. USENIX Association.

Kim, H.-J. and Lee, S.-G. (2002). An effective flash memory manager for reliable flash memory space management. In *IEICE Transactions on Information and System.*

Kim, Y., Oral, S., Shipman, G., Lee, J., Dillow, D., and Wang, F. (2011). Harmonia : A globally coordinated garbage collector for arrays of solid-state drives. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12.

Kim, Y., Tauras, B., Gupta, A., and Urgaonkar, B. (2009). Flashsim : A simulator for nand flash-based solid-state drives. In *Advances in System Simulation, 2009. SIMUL '09. First International Conference on*, pages 125–131.

Kitware Inc (2011). Site web cmake. cmake.org.

Lee, E. K. (1993). *Performance Modeling and Analysis of Disk Arrays*. PhD thesis, EECS Department, University of California, Berkeley.

Lee, J., Kim, Y., Shipman, G., Oral, S., Wang, F., and Kim, J. (2011). A semi-preemptive garbage collector for solid state drives. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 12–21.

Lee, S., Fleming, K., Park, J., K, H., Caulfield, A. M., Swanson, S., Arvind, and Kim, J. (2010). Bluessd : An open platform for cross-layer experiments for nand flash-based ssds. In *Proceedings of the 2010 Workshop on Architectural Research Prototyping*.

Lehrer, N. (2014). The largest data centers in the world. Network Computing website.

Li, Y. and Goel, A. (2012). Efficient distributed hot sparing scheme in a parity declustered raid organization. Patent US 8,099,623 B1.

Ligato, L. (2016). Here are the 6 largest data centers in the us. Bisnow website.

Linux Foundation (2016). *Page de manuel de limits.conf(5)*.

Linux Kernel Organization (2017). Site web the linux kernel archives. `http://www.kernel.org`.

M-Systems (1999). Trueffs® wear-leveling mechanism.

Manning, C. and Wookey (2001). Yaffs specification. In *Aleph One Limited*.

Micron (2006). An introduction to nand flash and how to design it to your next product.

Micron (2015). Micron and intel unveil new 3d nand flash memory. `http://investors.micron.com/releasedetail.cfm?ReleaseID=903522`.

Micron (2016). Spécifications micron 1100 3d nand. `http://www.micron.com/~/media/documents/products/product-flyer/1100_3d_nand_ssd_product_brief.pdf`.

Muntz, R. R. and Lui, J. C. S. (1990). Performance analysis of disk arrays under failure. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 162–173, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Murugan, M. and Du, D. (2011). Rejuvenator : A static wear leveling algorithm for nand flash memory with minimal overhead. In *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies : Research Track (MSST 2011)*.

National Semiconductor (1989). Dp8459 zoned bit recording.

Numonyx (2011). Numony embedded flash memory (j3 65nm) single bit per cell (sbc).

NVM Express Inc. (2016). Nvm express specifications revision 1.2.1.

ONFI (2014). Open nand flash interface specification 4.0.

Oracle (2016). Site web d'oracle pour vdbench. `http://www.oracle.com/technetwork/server-storage/vdbench-downloads-1901681.html`.

Park, S. and Shen, K. (2009). A performance evaluation of scientific i/o workloads on flash-based ssds. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–5.

Patterson, D., Gibson, G., and Katz, R. (1988). A case for redundant arrays of inexpensive disks (raid). In *the ACM SIGMOD international conference on Management of data*.

Postel, J. (1981). Transmission control protocol. RFC 793, Internet Engineering Task Force.

Red Hat Inc (2016). Ceph site. http ://ceph.com/ceph-storage/file-system.

Reno Luxury Homes (2015). Supernap reno largest data center in the world. Reno luxury homes website.

Rosenblum, M. and Ousterhout, J. K. (1992). The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1) :26–52.

Samsung (2014). Présentation de la technologie 3d nand samsung. `http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf`.

Samsung (2015). White paper samsung ssd950pro. `http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_950_PRO_White_paper.pdf`.

Samsung (2016a). Spécifications samsung pm1633a. `http://www.samsung.com/semiconductor/global/file/insight/2016/06/PM1633a-flyer-0.pdf`.

Samsung (2016b). Spécifications samsung ssd850evo. `http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_850_EVO_Data_Sheet_rev_2_0.pdf`.

Samsung (2016c). Spécifications samsung ssd960pro m.2. `http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_960_PRO_Data_Sheet_Rev_1_0.pdf`.

Schindler, J. and Ganger, G. R. (1999). Automated disk drive characterization. Technical report.

Schmuck, F. and Haskin, R. (2002). Gpfs : A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA. USENIX Association.

Schwabe, E., Holmer, K., and Sutherland, I. (1996). Evaluating approximately balanced parity-declustered data layouts for disk arrays. In *the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*.

Seagate (2011). Spécifications seagate st500dm002. `http://www.seagate.com/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf`.

Seagate (2016). Seagate accelerates enterprise momentum with two new flash products. http ://www.seagate.com/about-seagate/news/seagate-accelerates-enterprise-momentum-with-two-new-flash-products-pr/.

Shanbhag, S. S., Gururaj, P., Shetty, M. K., and Bellore, R. R. (2014). Declustered raid pool as backup for raid volumes. Patent US 0250269 A1.

SNIA (2009). Common raid disk data format specification version 2.0 revision 19.

STMicroelectronics (2006). Wear leveling in single level cell nand flash memories. STMicroelectronics Application Note (AN1822).

T13 Technical Committee (2002). Information technology - at attachment with packet interface - 6 (ata/atapi-6).

T13 Technical Committee (2007). Data set management commands proposal for ata8-acs2 (revision 6).

Tijoe, J. (2011). Making a flash translation layer reliability-aware (ra) : An optimized strategy for wear-leveling and garbage collection. Master's thesis, Faculty of San Diego State University.

Torvalds, L. (1992). Code source de fork.c. `http://github.com/torvalds/linux/blob/master/kernel/fork.c`.

Toshiba (2015). Spécifications toshiba mc04acaxxxe. `http://toshiba.semicon-storage.com/content/dam/toshiba-ss/asia-pacific/docs/product/storage/product-manual/eHDD-MC04ACAxxE-Product-Overview.pdf`.

Toshiba Corporation (1999). Smartmedia™ specification. SSFDC Forum.

Transcend (2015). Spécifications transcend ssd370. `http://www.datarespons.com/wp-content/uploads/2015/05/Product-Sheet_SSD370-SSD340-SSD320_V10.pdf`.

van Heesch, D. (2016). Site web doxygen. doxygen.org.

Veillard, D. (2016). Site web libxml2. www.xmlsoft.org.

Weil, S., Brandt, S., Miller, E., and Maltzahn, C. (2006a). Crush : Controlled, scalable, decentralized placement of replicated data. In *Supercomputing Conference (SC2006)*.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. (2006b).

Ceph : A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA. USENIX Association.

Wikipedia (2016). Page wikipedia du centre de données de l'utah. `http://en.wikipedia.org/wiki/Utah_Data_Center`.

Woodhouse, D. (2001). Jffs : The journalling flash file system. In *Proceedings of Ottawa Linux Symposium*.

Yoo, J., Won, Y., Hwang, J., Kang, S., Choil, J., Yoon, S., and Cha, J. (2013). Vssim : Virtual machine based ssd simulator. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14.

Zhang, J., Liu, H., and Ding, J. (2008). Disk drive storage defragmentation system. US Patent 7,447,836.

# Description des fichiers d'entrée d'OGSSim

Cette annexe traite de l'ensemble des balises utilisables dans les fichiers d'entrée XML d'OGSSim : le fichier de configuration du logiciel, le fichier de description de la configuration architecturale et le fichier de description des disques (HDD, SSD).

| Nom de la balise | Balise parente | Description |
|---|---|---|
| `config` | - | Racine du fichier |
| `path` | `config` | Chemins des fichiers d'entrée/sortie |
| `hardwarefile` | `path` | Chemin du fichier de configuration architectural |
| `resultfile` | `path` | Chemin du fichier de résultat |
| `subresultfile` | `path` | Chemin du fichier de résultat des sous-requêtes |
| `workloadfile` | `path` | Chemin du fichier de requêtes |
| `general` | `config` | Paramètres généraux d'OGSSim |
| `log` | `general` | Paramètres du fichier de journalisation |
| [module] | `config` | Paramètres des modules d'OGSSim – [module] peut être `event`, `workload`, `hardware`, `preproc`, `volumedriver`, `devicedriver`, `execution` ou `performance` |
| `zeromq` | [module] | Paramètres du canal de communication |
| `entry` | `event` | Description d'un événement |
| `reqdut` | `workload` | Taille de la *data unit* |
| `subreq` | `workload` | Taille du tableau de sous-requêtes |
| `cmbus` | `execution` | Modèle de calcul pour le transfert dans les bus |
| `graph` | `performance` | Description d'un graphe de visualisation |

Table 5 – Liste des balises dans le fichier de configuration d'OGSSim

| Nom de l'attribut | Balise parente | Description |
|---|---|---|
| mlvl | log | Niveau de journalisation : 0 pour debug, 1 pour info, 2 pour warning, 3 pour error et 4 pour fatal |
| file | log | Chemin du fichier de journalisation |
| intr | zeromq | Interlocuteur du canal de communication |
| prot | zeromq | Protocole de communication |
| addr | zeromq | Adresse de destination |
| port | zeromq | Port de communication |
| dev | entry | Disque ciblé par l'événement |
| date | entry | Date d'arrivée de l'événement |
| type | entry | Type de l'événement |
| type | graph | Type du graphe de visualisation |
| output | graph | Chemin du graphe de visualisation |

TABLE 6 – Liste des attributs du fichier de configuration d'OGSSim

| Nom de la balise | Balise parente | Description |
|---|---|---|
| architecture | - | Racine du fichier |
| buses | architecture | Description des bus |
| bus | buses | Description d'un bus |
| system | architecture | Description de la configuration architecturale |
| tier | system | Description d'un tier |
| decraid | tier | Description d'un *declustered RAID* |
| volume | tier, decraid | Description d'un volume |
| config | volume | Description de la configuration logique du volume |
| device | decraid, volume | Description d'un disque |

TABLE 7 – Liste des balises du fichier de configuration architecturale

| Nom de l'attribut | Balise parente | Description |
|---|---|---|
| nbbuses | buses | Nombre de bus |
| name | bus | Nom du bus |
| nbports | bus | Nombre maximum de disques connectés |
| bandwidth | bus | Bande-passante du bus |
| type | bus | Type de bus |
| nbtiers | system | Nombre de tiers |
| bus | system | Bus reliant l'hôte aux tiers |
| nbvolumes | tier | Nombre de volumes composant le tier |
| bus | tier | Bus reliant le tier aux volumes |
| type | decraid | Algorithme utilisé pour le placement des données |
| nbsubvol | decraid | Nombre de sous-volumes composant le *declustered RAID* |
| nbdevices | decraid | Nombre de disques composant le *declustered RAID* |
| nbspare | decraid | Nombre de disques *spare* |
| decsize | decraid | Taille du *stripe unit* |
| bus | decraid | Bus reliant le *declustered RAID* aux disques |
| buffersize | decraid | Taille du buffer du contrôleur |
| nbdevices | volume | Nombre de disques composant le volume |
| bus | volume | Bus reliant le volume aux disques |
| type | config | Type de la configuration attachée au volume : JBOD, RAID1, RAID01, RAIDNP |
| stripeunitsize | config | Taille du *stripe unit* (RAID01 et RAIDNP uniquement) |
| nbpardisks | config | Nombre de disques de parité (RAIDNP uniquement) |
| decl | config | Type de *declustering* (RAIDNP uniquement) : none, parity, data |
| sreqoptim | config | Fusion des sous-requêtes si données contigües |
| parityread | config | Vérification des données en lecture (RAIDNP uniquement) |
| buffersize | config | Taille du buffer du contrôleur |

TABLE 8 – Liste des attributs du fichier de configuration architecturale

| Nom de la balise | Balise parente | Description |
|---|---|---|
| device | - | Racine du fichier, informe également sur le type de disque : hdd ou ssd |
| information | device | Informations générales sur le disque |
| name | information | Nom du disque |
| capacity | information | Capacité du disque (en Go) |
| producer | information | Fabriquant du disque |
| year | information | Année de fabrication |
| geometry | device | Informations sur la géométrie |
| sectorsize | geometry | (HDD) Taille du secteur |
| sectorsbytrack | geometry | (HDD) Nombre de secteurs par piste |
| tracksbyplatter | geometry | (HDD) Nombre de pistes par plateau |
| nbplatters | geometry | (HDD) Nombre de plateaux |
| dataheads | geometry | (HDD) Nombre de têtes de lecture |
| pagesize | geometry | (SSD) Taille de la page |
| pagesbyblock | geometry | (SSD) Nombre de pages par bloc |
| blocksbydie | geometry | (SSD) Nombre de blocs par die |
| nbdies | geometry | (SSD) Nombre de dies |
| performance | device | Informations sur les performances du disque |
| minrseek | performance | (HDD) Temps de recherche en lecture minimal |
| avgrseek | performance | (HDD) Temps de recherche en lecture moyen |
| maxrseek | performance | (HDD) Temps de recherche en lecture maximal |
| minwseek | performance | (HDD) Temps de recherche en écriture minimal |
| avgwseek | performance | (HDD) Temps de recherche en écriture moyen |
| maxwseek | performance | (HDD) Temps de recherche en écriture maximal |
| randread | performance | (SSD) Débit en lecture aléatoire |
| randwrite | performance | (SSD) Débit en écriture aléatoire |
| seqread | performance | (SSD) Débit en lecture séquentielle |
| seqwrite | performance | (SSD) Débit en écriture séquentielle |
| erase | performance | (SSD) Temps de service d'un effacement |
| mediatransferrate | performance | Débit de transfert interne |
| rotspeed | performance | (HDD) Vitesse de rotation |
| buffersize | performance | Taille du buffer interne |
| reliability | device | Informations sur la fiabilité du disque |
| mttf | reliability | Temps moyen avant une défaillance |
| nberase | reliability | (SSD) Nombre d'effacements par bloc |

TABLE 9 – Liste des balises du fichier de description des disques

# Algorithmes de décomposition de requêtes

Cette annexe reprend les algorithmes de décomposition des requêtes en sous-requêtes, processus amplement détaillé dans le chapitre 3. Ces algorithmes sont utilisés par les modules de pilote de volume d'OGSSim, pour construire les sous-requêtes correspondantes à une requête ciblant une configuration. Chaque sous-requête cible des données contigües à destination d'un seul disque physique.

**Algorithme 14 :** Redirection dans un JBOD

**Entrées :** $r$ – requête
$s_d$ – taille d'un disque (du)
**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $reste \leftarrow r.taille$
3 $adr \leftarrow r.adresse \bmod s_d$
4 $disque \leftarrow r.adresse/s_d$
5 **tant que** $reste > 0$ **faire**
6 $\quad r'.adresse \leftarrow adr$
7 $\quad r'.taille \leftarrow \min(reste, s_d - (adr \bmod s_d))$
8 $\quad r'.disque \leftarrow disque$
9 $\quad r'.type \leftarrow r.type$
10 $\quad S \leftarrow S \cup \{r'\}$
11 $\quad disque \leftarrow disque + 1$
12 $\quad adr \leftarrow 0$
13 $\quad reste \leftarrow reste - r'.taille$
14 **fin**

**Algorithme 15 :** Redirection dans un RAID-0

**Entrées :** $r$ – requête
$t_s$ – taille d'un *stripe unit* (du)
$n_d$ – nombre de disques
**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $reste \leftarrow r.taille$
3 $ligne \leftarrow r.adresse/(t_s \times n_d)$
4 $disque \leftarrow (r.adresse/t_s) \bmod n_d$
5 $adr \leftarrow ligne \times t_s$
6 **tant que** $reste > 0$ **faire**
7 $\quad r'.adresse \leftarrow adr$
8 $\quad r'.taille \leftarrow t_s$
9 $\quad r'.disque \leftarrow disque$
10 $\quad r'.type \leftarrow r.type$
11 $\quad S \leftarrow S \cup \{r'\}$
12 $\quad disque \leftarrow disque + 1$
13 $\quad$ **si** $disque = n_d$ **alors**
14 $\quad\quad disque \leftarrow 0$
15 $\quad\quad ligne \leftarrow ligne + 1$
16 $\quad$ **fin**
17 $\quad adr \leftarrow ligne \times t_s$
18 $\quad reste \leftarrow reste - t_s$
19 **fin**

**Algorithme 16 :** Redirection dans un RAID-1

**Entrées :** $r$ – requête
$n_d$ – nombre de disques
**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 **si** $r.type = lecture$ **alors**
3 $\quad r'.adresse \leftarrow r.adresse$
4 $\quad r'.taille \leftarrow r.taille$
5 $\quad r'.disque \leftarrow \text{rand}(0, n_d)$
6 $\quad S \leftarrow S \cup r'$
7 **sinon**
8 $\quad$ **pour chaque** $i \in [0, n_d[$ **faire**
9 $\quad\quad r'.adresse \leftarrow r.adresse$
10 $\quad\quad r'.taille \leftarrow r.taille$
11 $\quad\quad r'.disque \leftarrow i$
12 $\quad\quad r'.type \leftarrow r.type$
13 $\quad\quad S \leftarrow S \cup r'$
14 $\quad$ **fin**
15 **fin**

**Algorithme 17 :** Redirection dans un RAID-01

**Entrées :** $r$ – requête
$t_s$ – taille d'un *stripe unit* (du)
$n_d$ – nombre de disques

**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $reste \leftarrow r.taille$
3 $ligne \leftarrow r.adresse/(t_s \times n_d)$
4 $disque \leftarrow (r.adresse/t_s) \bmod n_d$
5 $adr \leftarrow ligne \times t_s$
6 **tant que** $reste > 0$ **faire**
7 $\quad r'.adresse \leftarrow adr$
8 $\quad r'.taille \leftarrow t_s$
9 $\quad r'.type \leftarrow r.type$
10 $\quad$ **si** $r.type = lecture$ **alors**
11 $\quad\quad r'.disque \leftarrow disque +$ rand $(0, 2) \times (n_d/2)$
12 $\quad\quad S \leftarrow S \cup \{r'\}$
13 $\quad$ **sinon**
14 $\quad\quad r'' \leftarrow r'$
15 $\quad\quad r'.disque \leftarrow disque$
16 $\quad\quad r''.disque \leftarrow disque + (n_d/2)$
17 $\quad\quad r''.type \leftarrow r'.type$
18 $\quad\quad S \leftarrow S \cup \{r', r''\}$
19 $\quad$ **fin**
20 $\quad disque \leftarrow disque + 1$
21 $\quad$ **si** $disque = n_d/2$ **alors**
22 $\quad\quad disque \leftarrow 0$
23 $\quad\quad ligne \leftarrow ligne + 1$
24 $\quad$ **fin**
25 $\quad adr \leftarrow ligne \times t_s$
26 $\quad reste \leftarrow reste - t_s$
27 **fin**

**Algorithme 18 :** Redirection d'une lecture dans un RAID-5

**Entrées :** $r$ – requête
$t_s$ – taille d'un *stripe unit* (du)
$n_d$ – nombre de disques
**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $reste \leftarrow r.taille$
3 $l_1 \leftarrow r.adresse/(t_s \times (n_d - 1))$
4 $l_n \leftarrow (r.adresse + r.taille - 1)/(t_s \times (n_d - 1))$
5 **si** $l_1 = l_n$ **alors**
6 | **si** $r.taille = t_s \times (nd - 1)$ **alors**
7 | | $S \leftarrow S \cup$ readFullStripe $(i, n_d, t_s)$
8 | **sinon**
9 | | $S \leftarrow S \cup$ readPartStripe $(i, n_d, t_s,$ mod $(r.adresse, t_s \times (n_d - 1), r.taille)$
10 | **fin**
11 **sinon**
12 | **pour** $i \in [l_1, l_n]$ **faire**
13 | | **si** $i \notin \{l_1, l_n\}$ **alors**
14 | | | $S \leftarrow S \cup$ readFullStripe $(i, n_d, t_s)$
15 | | | $reste \leftarrow reste - (t_s \times (n_d - 1))$
16 | | **sinon si** $i = l_1$ **alors**
17 | | | $taille \leftarrow t_s \times (n_d - 1) -$ mod $(r.adresse, t_s \times (n_d - 1))$
18 | | | **si** $taille = t_s \times (n_d - 1$ **alors**
19 | | | | $S \leftarrow S \cup$ readFullStripe $(i, n_d, t_s)$
20 | | | **sinon**
21 | | | | $S \leftarrow$
| | | | $S \cup$ readPartStripe $(i, n_d, t_s,$ mod $(r.adresse, t_s \times (n_d - 1), taille)$
22 | | | **fin**
23 | | | $reste \leftarrow reste - taille$
24 | | **sinon**
25 | | | **si** $reste = t_s \times (n_d - 1)$ **alors**
26 | | | | $S \leftarrow S \cup$ readFullStripe $(i, n_d, t_s)$
27 | | | **sinon**
28 | | | | $S \leftarrow S \cup$ readPartStripe $(i, n_d, t_s, 0, reste)$
29 | | | **fin**
30 | | **fin**
31 | **fin**
32 **fin**

**Algorithme 19 :** Traitement d'une requête *Part Stripe* en lecture

**Entrées :** $i_s$ – indice du stripe ciblé
$n_d$ – nombre de disques
$adr$ – adresse de début des données sur le stripe
$t$ – taille des données sur le stripe

**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $d_{parite} \leftarrow n_d - \bmod (i_s, n_d) - 1$
3 $r'.disque \leftarrow \bmod (adr, t_s \times (n_d - 1))/t_s$
4 $r'.adresse \leftarrow i_s \times t_s$
5 $r'.taille \leftarrow t_s$
6 $r'.type \leftarrow r.type$
7 **tant que** $t > 0$ **faire**
8 | **si** $r'.disque = d_{parite}$ **alors**
9 | | $r'.disque \leftarrow r'.disque + 1$
10 | $S \leftarrow S \cup \{r'\}$
11 | $t \leftarrow t - t_s$
12 | $r'.adresse \leftarrow i \times t_s$
13 | $r'.disque \leftarrow r'.disque + 1$
14 **fin**

**Algorithme 20 :** Traitement d'une requête *Full Stripe* en lecture

**Entrées :** $i_s$ – indice du stripe ciblé
$n_d$ – nombre de disques
$t_s$ – taille d'un *stripe unit* (du)

**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $d_{parite} \leftarrow n_d - \bmod (i_s, n_d) - 1$
3 $r'.adresse \leftarrow i_s \times t_s$
4 $r'.taille \leftarrow t_s$
5 $r'.type \leftarrow r.type$
6 **pour** $i \in [0, n_d[/\{d_{parite}\}$ **faire**
7 | $r'.disque \leftarrow i$
8 | $S \leftarrow S \cup \{r'\}$
9 **fin**

**Algorithme 21 :** Redirection d'une écriture dans un RAID-5

**Entrées :** $r$ – requête
$t_s$ – taille d'un *stripe unit* (du)
$n_d$ – nombre de disques
**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $reste \leftarrow r.taille$
3 $l_1 \leftarrow r.adresse/(t_s \times (n_d - 1))$
4 $l_n \leftarrow (r.adresse + r.taille - 1)/(t_s \times (n_d - 1))$
5 **si** $l_1 = l_n$ **alors**
6 | **si** $r.taille = t_s \times (n_d - 1)$ **alors**
7 | | $S \leftarrow S \cup$ writeFullStripe $(i, n_d, t_s)$
8 | **sinon si** $r.taille < (t_s \times (n_d - 1)/2)$ **alors**
9 | | $S \leftarrow S \cup$ writeSmallStripe $(i, n_d, t_s,$ mod $(r.adresse, t_s \times (n_d - 1), r.taille)$
10 | **sinon**
11 | | $S \leftarrow S \cup$ writeLargeStripe $(i, n_d, t_s,$ mod $(r.adresse, t_s \times (n_d - 1), r.taille)$
12 | **fin**
13 **sinon**
14 | **pour** $i \in [l_1, l_n]$ **faire**
15 | | **si** $i \notin \{l_1, l_n\}$ **alors**
16 | | | $S \leftarrow S \cup$ writeFullStripe $(i, n_d, t_s)$
17 | | | $reste \leftarrow reste - (t_s \times (n_d - 1))$
18 | | **sinon si** $i = l_1$ **alors**
19 | | | $taille \leftarrow t_s \times (n_d - 1) -$ mod $(r.adresse, t_s \times (n_d - 1))$
20 | | | **si** $taille = t_s \times (n_d - 1)$ **alors**
21 | | | | $S \leftarrow S \cup$ writeFullStripe $(i, n_d, t_s)$
22 | | | **sinon si** $taille < t_s \times (n_d - 1)/2$ **alors**
23 | | | | $S \leftarrow S \cup$ writeSmallStripe $(i, n_d, t_s,$ mod $(r.adresse, t_s \times (n_d - 1), taille)$
24 | | | **fin**
25 | | | **sinon**
26 | | | | $S \leftarrow S \cup$ writeLargeStripe $(i, n_d, t_s,$ mod $(r.adresse, t_s \times (n_d - 1), taille)$
27 | | | **fin**
28 | | | $reste \leftarrow reste - taille$
29 | | **sinon**
30 | | | **si** $reste = t_s \times (n_d - 1)$ **alors**
31 | | | | $S \leftarrow S \cup$ writeFullStripe $(i, n_d, t_s)$
32 | | | **sinon si** $reste < t_s \times (n_d - 1)/2$ **alors**
33 | | | | $S \leftarrow S \cup$ writeSmallStripe $(i, n_d, t_s, 0, reste)$
34 | | | **fin**
35 | | | **sinon**
36 | | | | $S \leftarrow S \cup$ writeLargeStripe $(i, n_d, t_s, 0, reste)$
37 | | | **fin**
38 | | **fin**
39 | **fin**
40 **fin**

**Algorithme 22 :** Traitement d'une requête *Small Stripe* en écriture

**Entrées :** $i_s$ – indice du stripe ciblé
$n_d$ – nombre de disques
$adr$ – adresse de début des données sur le stripe
$t$ – taille des données sur le stripe
$t_s$ – taille d'un *stripe unit* (du)

**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $d_{parite} \leftarrow n_d - \bmod (i_s, n_d) - 1$
3 $r'.disque \leftarrow d_{parite}$
4 $r'.adresse \leftarrow i_s \times t_s$
5 $r'.taille \leftarrow t_s$
6 $r'.type \leftarrow lecture$
7 $S \leftarrow S \cup \{r'\}$
8 $r'.type \leftarrow ecriture$
9 $S \leftarrow S \cup \{r'\}$
10 $r'.disque \leftarrow \bmod (adr, t_s \times (n_d - 1))/t_s$
11 $r'.adresse \leftarrow i_s \times t_s$
12 $r'.taille \leftarrow t_s$
13 **tant que** $t > 0$ **faire**
14 | **si** $r'.disque = d_{parite}$ **alors**
15 | | $r'.disque \leftarrow r'.disque + 1$
16 | $r'.type \leftarrow lecture$
17 | $S \leftarrow S \cup \{r'\}$
18 | $r'.type \leftarrow ecriture$
19 | $S \leftarrow S \cup \{r'\}$
20 | $t \leftarrow t - r'.taille$
21 | $r'.adresse \leftarrow i_s \times t_s$
22 | $r'.taille \leftarrow \min (t_s, t)$
23 | $r'.disque \leftarrow r'.disque + 1$
24 **fin**

**Algorithme 23 :** Traitement d'une requête *Large Stripe* en écriture

**Entrées :** $i_s$ – indice du stripe ciblé
$n_d$ – nombre de disques
$adr$ – adresse de début des données sur le stripe
$t$ – taille des données sur le stripe
$t_s$ – taille d'un *stripe unit* (du)

**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $d_{parite} \leftarrow n_d - \bmod (i_s, n_d) - 1$
3 $r'.adresse \leftarrow i_s \times t_s$
4 $r'.taille \leftarrow t_s$
5 $r'.disque \leftarrow d_{parite}$
6 $r'.type \leftarrow lecture$
7 $S \leftarrow S \cup \{r'\}$
8 $r'.type \leftarrow ecriture$
9 $S \leftarrow S \cup \{r'\}$
10 $t' \leftarrow \bmod (adr, t_s \times (n_d - 1))$
11 $t'' \leftarrow t_s \times (n_d - 1) - t - t'$
12 $r'.disque \leftarrow 0$
13 $r'.type \leftarrow lecture$
14 **tant que** $t' > 0$ **faire**
15 | **si** $r'.disque = d_{parite}$ **alors**
16 | | $r'.disque \leftarrow r'.disque + 1$
17 | $S \leftarrow S \cup \{r'\}$
18 | $r'.disque \leftarrow r'.disque + 1$
19 | $t' \leftarrow t' - t_s$
20 **fin**
21 $r'.type \leftarrow ecriture$
22 **tant que** $t > 0$ **faire**
23 | **si** $r'.disque = d_{parite}$ **alors**
24 | | $r'.disque \leftarrow r'.disque + 1$
25 | $S \leftarrow S \cup \{r'\}$
26 | $r'.disque \leftarrow r'.disque + 1$ $t \leftarrow t - t_s$
27 **fin**
28 $r'.type \leftarrow lecture$
29 **tant que** $t'' > 0$ **faire**
30 | **si** $r'.disque = d_{parite}$ **alors**
31 | | $r'.disque \leftarrow r'.disque + 1$
32 | $S \leftarrow S \cup \{r'\}$
33 | $r'.disque \leftarrow r'.disque + 1$
34 | $t'' \leftarrow t'' - t_s$
35 **fin**

**Algorithme 24 :** Traitement d'une requête *Full Stripe* en écriture

**Entrées :** $i_s$ – indice du stripe ciblé
$n_d$ – nombre de disques
**Sorties :** $S$ – ensemble des sous-requêtes créées

1 $S \leftarrow \emptyset$
2 $r'.adresse \leftarrow i_s \times t_s$
3 $r'.taille \leftarrow t_s$
4 $r'.type \leftarrow r.type$
5 **pour** $i \in [0, n_d[$ **faire**
6 $\quad r'.disque \leftarrow i$
7 $\quad S \leftarrow S \cup \{r'\}$
8 **fin**

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

**Titre :** Simulation générique et contribution à l'optimisation de la robustesse des systèmes de stockage de données à large échelle

**Mots clefs :** Simulation, Stockage de données à large échelle, Robustesse, Disques magnétiques, Disques à mémoire Flash

**Résumé :** La capacité des systèmes de stockage de données ne cesse de croître pour atteindre actuellement l'échelle de l'exaoctet, ce qui a un réel impact sur la robustesse des systèmes de stockage. En effet, plus le nombre de disques contenus dans un système est grand, plus il est probable d'y avoir une défaillance. De même, le temps de la reconstruction d'un disque est proportionnel à sa capacité.
La simulation permet le test de nouveaux mécanismes dans des conditions quasi réelles et de prédire leur comportements. Open and Generic data Storage system Simulation tool (OGSSim), l'outil que nous proposons, supporte l'hétérogénéité et la taille importante des systèmes actuels. Sa décomposition modulaire permet d'entreprendre chaque technologie de stockage, schéma de placement ou modèle de calcul comme des briques pouvant être combinées entre elles pour paramétrer au mieux la simulation.
La robustesse étant un paramètre critique dans ces systèmes, nous utilisons le declustered RAID pour assurer la distribution de la reconstruction des données d'un disque en cas de défaillance. Nous proposons l'algorithme Symmetric Difference of Source Sets (SD2S) qui utilise le décalage des blocs de données pour la création du schéma de placement. Le pas du décalage est issu du calcul de la proximité des ensembles de provenance logique des blocs d'un disque physique.
Pour évaluer l'efficacité de SD2S, nous l'avons comparé à la méthode Crush, exemptée des réplicas. Il en résulte que la création du schéma de placement, aussi bien en mode normal qu'en mode défaillant, est plus rapide avec SD2S, et que le coût en espace mémoire est également réduit (nul en mode normal). En cas de double défaillance, SD2S assure la sauvegarde d'une partie, voire de la totalité, des données.

**Title :** Generic Simulation and Contribution to the Robustness Optimization of Large-Scale Data Storage Systems

**Keywords :** Simulation, Large-Scale Data Storage, Robustness, Hard Disk Drives, Solid-State Drives

**Abstract :** Capacity of data storage systems does not cease to increase to currently reach the exabyte scale. This observation gets a real impact on storage system robustness. In fact, the more the number of disks in a system is, the greater the probability of a failure happening is. Also, the time used for a disk reconstruction is proportional to its size.
Simulation is an appropriate technique to test new mechanisms in almost real conditions and predict their behavior. We propose a new software we call Open and Generic data Storage system Simulation tool (OGSSim). It handles the heterogeneity and the large size of these modern systems. Its modularity permits the undertaking of each storage technology, placement scheme or computation model as bricks which can be added and combined to optimally configure the simulation.
Robustness is a critical issue for these systems. We use the declustered RAID to distribute the data reconstruction in case of a failure. We propose the Symmetric Difference of Source Sets (SD2S) algorithm which uses data block shifhting to achieve the placement scheme. The shifting offset comes from the computation of the distance between logical source sets of physical disk blocks.
To evaluate the SD2S efficiency, we compared it to Crush method without replicas. It results in a faster placement scheme creation in normal and failure modes with SD2S and in a significant reduced memory space cost (null without failure). Furthermore, SD2S ensures the partial, if not total, reconstruction of data in case of multiple failures.

**Université Paris-Saclay**
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France