Optimisation des caches de fichiers dans les environnements virtualisés

Grégoire Todeschi

▶ **To cite this version:**

Grégoire Todeschi. Optimisation des caches de fichiers dans les environnements virtualisés. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Toulouse - INPT, 2020. Français. NNT : 2020INPT0048 . tel-04166072

HAL Id: tel-04166072

https://theses.hal.science/tel-04166072v1

Submitted on 19 Jul 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

M. GRÉGOIRE TODESCHI

le lundi 8 juin 2020

Titre :

Optimisation des caches de fichiers dans les environnements virtualisés

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

M. DANIEL HAGIMONT

M. ALAIN TCHANA

Rapporteurs :

M. CHRISTIAN PEREZ, ECOLE NORMALE SUPERIEURE DE LYON

Mme FABIENNE BOYER, UNIVERSITE GRENOBLE ALPES

Membre(s) du jury :

M. ADRIEN LEBRE, IMT ATLANTIQUE NANTES, Président

M. ALAIN TCHANA, ECOLE NORMALE SUP LYON ENS DE LYON, Membre

M. DANIEL HAGIMONT, TOULOUSE INP, Membre

À Laurène

Le plaisir le plus noble
est la joie de comprendre.
Léonard de Vinci

Remerciements

Je tenais à remercier en premier lieu mon directeur de thèse Daniel Hagimont qui m'a donné ma chance, accueilli dans cette équipe, a su me conseiller et me guider pendant toutes ces années de thèse. Il a été d'une aide précieuse, il m'a apporté son expérience, son esprit critique et son recul avec des remarques et améliorations toujours justes et constructives autant sur la direction de la thèse que sur la rédaction d'articles ou de ce même manuscrit. Je remercie aussi Alain Tchana qui a co-encadré cette thèse et qui a fait avancer cette thèse à l'aide de sa vision, ses idées et son imagination sans fin.

Outre mes directeurs de thèse, je voulais remercier l'ensemble du jury de ma thèse, qui a dû se faire dans des conditions toutes nouvelles et particulières, en visio-conférence. Tout d'abord, Adrien Lebre pour avoir accepté de présider ce jury, et ensuite Fabienne Boyer et Christian Pérez qui ont eu la responsabilité d'évaluer et rapporter mes travaux, et tous ont su apporter leur remarques et critiques constructives mais aussi des perspectives intéressantes.

Je souhaite également remercier mes amis et camarades. Un grand merci à Mathieu Bacou avec qui j'ai partagé cette thèse de bout en bout, et également un grand merci à Boris Teabe qui a apporté son aide infaillible pendant toute la durée de ma thèse dans le bureau mais aussi en dehors. Je voulais aussi remercier le reste des membres de l'équipe SEPIA avec qui j'ai partagé cette aventure, Brice, Djob, Lavoisier, Bao, Tu, Léon et Vlad. Grace à eux le bureau était toujours rempli de discussions animées et enrichissantes à en refaire le monde.

Un immense merci à mes parents pour leur soutien indéfectible depuis toutes ces années, dans tous ce que j'ai pu entreprendre et surtout toujours présent pour une relecture et des conseils.

Enfin, je n'aurais pas assez de mots pour exprimer toute ma gratitude et mon affection à Laurène pour ses encouragements, son réconfort, sa bienveillance et son soutien sans quoi toute cette thèse n'aurait sûrement pas abouti. Elle a su m'inspirer par sa volonté et sa détermination face à toutes les épreuves que représente une thèse. Merci !

Résumé

Les besoins en ressources de calcul sont en forte augmentation depuis plusieurs décennies, que ce soit pour des applications du domaine des réseaux sociaux, du calcul haute performance, ou du big data. Les entreprises se tournent alors vers des solutions d'externalisation de leurs services informatiques comme le Cloud Computing. Le Cloud Computing permet une mutalisation des ressources informatiques dans un datacenter et repose généralement sur la virtualisation. Cette dernière permet de décomposer une machine physique, appelée hôte, en plusieurs machines virtuelles (VM) invitées. La virtualisation engendre de nouveaux défis dans la conception des systèmes d'exploitation, en particulier pour la gestion de la mémoire. La mémoire est souvent utilisée pour accélérer les coûteux accès aux disques, en conservant ou préchargeant les données du disque dans le cache fichiers. Seulement la mémoire est une ressource limitée et limitante pour les environnements virtualisés, affectant ainsi les performances des applications utilisateurs. Il est alors nécessaire d'optimiser l'utilisation du cache de fichiers dans ces environnements. Dans cette thèse, nous proposons deux approches orthogonales pour améliorer les performances des applications à l'aide d'une meilleure utilisation du cache fichiers.

Dans les environnements virtualisés, hôte et invités exécutent chacun leur propre système d'exploitation (OS) et ont donc chacun un cache de fichiers. Lors de la lecture d'un fichier, les données se retrouvent présentes dans les deux caches. Seulement, les deux OS exploitent la même mémoire physique. On parle de duplication des pages du cache. La première contribution vise à pallier ce problème avec Cacol, une politique d'éviction de cache s'exécutant dans l'hôte et non intrusive vis-à-vis de la VM. Cacol évite ces doublons de pages réduisant ainsi l'utilisation de la mémoire d'une machine physique.

La seconde approche est d'étendre le cache fichiers des VM en exploitant de la mémoire disponible sur d'autres machines du datacenter. Cette seconde contribution, appelée Infinicache, s'appuie sur Infiniband, un réseau RDMA à haute vitesse, et exploite sa capacité à lire et à écrire sur de la mémoire à distance. Directement implémenté dans le cache invité, Infinicache stocke les pages évincées de son cache sur de la mémoire à distance. Les futurs accès à ces pages sont alors plus rapides que des accès aux disques de stockage, améliorant par conséquent les performances des applications. De plus, le taux d'utilisation de la mémoire à l'échelle du datacenter est augmenté, réduisant le gaspillage de manière globale.

Abstract

The need for computing resources has been growing significantly for several decades, in application domains from social networks, high-performance computing, or big data. Consequently, companies are outsourcing theirs IT services towards Cloud Computing solutions. Cloud Computing allows mutualizing computing resources in a data center, and generally relies on virtualization. Virtualization allows a physical machine, called a host, to be split into multiple guest virtual machines (VMs). Virtualization brings new challenges in the design of operating systems, especially memory management. Memory is often used to speed up expensive disk accesses by storing or preloading data from the disk to the file cache. However memory is a limited and limiting resource for virtualized environments, thus impacting the performance of user applications. It is therefore necessary to optimize the use of the file cache in these environments. In this thesis, we propose two orthogonal approaches to improve application performance through better use of the file cache.

In virtualized environments, both host and guests run their own operating system (OS) and thus have their own file cache. When a file is read, the data is present in both caches. However, both OSes use the same physical memory. We hence have a phenomenon called pages duplication. The first contribution aims at alleviating this problem with Cacol, a host cache eviction policy, which is non-intrusive for the VM. Cacol avoids these duplicated pages, thus reducing the memory usage of a physical machine.

The second approach is to extend the file cache of VMs by exploiting memory available on other machines in the datacenter. This second contribution, called Infinicache, relies on Infiniband, a high-speed RDMA network, and exploits its ability to read and write remote memory. Implemented directly in the guest cache, Infinicache stores on remote memory pages that have been evicted from its cache. Future accesses to these pages are then be faster than accesses to storage disks, thereby improving application performance. In addition, the datacenter-wide memory utilization rate is increased, reducing overall memory wasting.

Table des matières

Table des figures

Chapitre 2 - Contexte général

Chapitre 3 - Cacol

Chapitre 4 - Infinicache

Chapitre 1

Introduction

Contenu

1.1 Contexte

De nos jours, de plus en plus de domaines d'applications de l'informatique nécessitent une puissance de calcul fournie par une infrastructure calcul, que ce soit les domaines du big data, du machine learning, du calcul haute performance, ou des applications de réseaux sociaux. Pour satisfaire ces besoins, les entreprises se tournent vers le Cloud computing, un moyen d'externaliser leurs services informatiques. Le cloud propose une mutualisation des ressources informatiques et ainsi de réduire les coûts humains, matériels et logiciels grâce à l'économie d'échelle. Un fournisseur de cloud possède un centre de données où il déploie ses nombreuses machines physiques et peut proposer de fournir ses ressources à des clients. Dans le cas d'un modèle IaaS (*Infrastructure as a Service*), le client loue des ressources du centre de données sous la forme de machines virtuelles. Ce modèle s'appuie sur la virtualisation comme composant principal rendant possible le découpage d'une machine physique, appelée hôte, en plusieurs machines virtuelles (VM) invitées. La virtualisation est rendue possible grâce à un hyperviseur, ce dernier fournit une abstraction logicielle pour partitionner une machine physique, isoler les machines virtuelles entre elles, et fournir aux VM un accès au matériel de la machine physique. Le fournisseur

de cloud peut alors allouer au client, sous la forme de VM, un espace de travail de taille pré-définie, en garantissant la qualité de service et les performances.

Le stockage des données du client se fait sur des disques durs qu'ils soient montés en local ou en réseau. L'accès et la vitesse d'accès à ces données sont primordiaux pour le client du cloud. Pour accélérer les accès aux disques, les systèmes d'exploitation (OS) mettent en place un cache en mémoire vive. L'OS conserve un certain nombre de blocs du disque en mémoire pour en accélérer les futurs accès car la mémoire a une latence d'accès bien plus faible qu'un disque (que ce soit un HDD ou SSD). Il peut également pré-charger des données du disque pour anticiper des accès au disque. Malheureusment, la mémoire est une ressource limitée et même limitante des machines des centres de données, affectant ainsi les performances des applications utilisateur. De plus, la virtualisation apporte un nouveau lot de défis quant à la gestion de la mémoire dans ces environnements virtualisés.

Cette thèse se place dans le contexte du cloud computing utilisant un IaaS et s'intéresse au défi de la gestion mémoire et plus particulièrement des caches dans un environnement virtualisé.

1.2 Contributions

Comme nous venons de voir, nous nous intéressons à la gestion de la mémoire dans un environnement de cloud virtualisé. Une gestion efficace de la mémoire est primordiale dans un contexte de cloud et cela passe par une gestion des caches efficace. La mémoire peut être séparée en deux catégories, la mémoire anonyme qui est la mémoire allouée aux applications pour leurs besoins en calculs, et la mémoire du cache de fichiers qui correspond à des blocs du disque copiés en mémoire pour permettre un accès moins coûteux. Le cache de fichiers est ainsi d'autant plus important qu'il peut avoir des conséquences considérables sur les performances des applications du client. Il est donc nécessaire d'optimiser l'utilisation du cache de fichiers dans ces environnements virtualisés. Il existe deux principales techniques pour améliorer les performances et l'efficacité du cache de fichiers dans ce contexte. Le premier axe d'amélioration est lié à la problématique de la virtualisation entraînant la duplication des pages du cache. C'est le cas lorsqu'une page est chargée à la fois dans le cache de l'hôte et le cache de l'invité entraînant un gaspillage de la mémoire. Le second axe est de fournir le plus de mémoire possible aux machines virtuelles et

surtout en éviter le gaspillage, cela peut être le cas grâce à la mutualisation des ressources locales ou la récupération de mémoire disponible sur des machines à distance sur le réseau.

Dans cette thèse, nous proposons deux approches orthogonales, découlant de chacun des axes d'amélioration identifiés, pour améliorer les performances des applications à l'aide d'une meilleure utilisation du cache de fichiers.

La première contribution de cette thèse est Cacol, un système s'exécutant dans le cache hôte, dont l'objectif est de réduire la duplication des pages du cache fichiers dans un environnement virtualisé. Nous proposons une politique d'éviction du cache de fichiers de l'hôte, implémantant une exclusivité entre le cache de l'hôte et le cache de l'invité, en se basant sur les différentes lectures et écritures faites par la machine virtuelle, sans être intrusif vis-à-vis de la VM. Nous pouvons ainsi réduire l'empreinte mémoire des VM sur le cache hôte, libérant de la mémoire et améliorant les performances de l'ensemble des VM.

La seconde contribution est Infinicache, un système exploitant la mémoire disponible sur des machines du centre de données pour étendre le cache de fichiers. Ce système tire parti d'un réseau à haute vitesse RDMA ayant la particularité de pouvoir lire et écrire sur de la mémoire à distance sans intervention de la machine distante. Cette extension du cache de fichiers est directement implémentée dans la machine virtuelle et permet ainsi d'amliorer les performances des applications clientes.

Cette thèse a donné lieu aux publications suivantes :

Todeschi Grégoire, Teabe Boris, Tchana Alain, and Hagimont Daniel. Cacol : A zero overhead and non-intrusive double caching mitigation system. *Future Generation Computer Systems*, 2020, vol. 106, p. 14-21.

Todeschi Grégoire, Nitu Vlad, Teabe Boris, and Hagimont Daniel. CaCol : système collaboratif et dynamique de gestion du cache pour les VMs. *Conférence en Parallélisme, Architecture et Système (Compas)*. 2018.

Todeschi Grégoire, Teabe Boris, and Hagimont Daniel. Infinicache : a VM page cache extension system over RDMA. *Travaux non publiés.*

Participations à d'autres travaux :

Bacou Mathieu, Todeschi Grégoire, Hagimont Daniel, and Tchana Alain. Nested Virtualization Without the Nest. *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*. 2019. p. 1-10.

Bacou Mathieu, Todeschi Grégoire, Tchana Alain, Hagimont Daniel, Lepers Baptiste, and Zwaenepoel Willy. Drowsy-DC : Data Center Power Management System. *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019. p. 825-834.

1.3 Plan de thèse

Cette thèse s'organise comme suit.

- *Contexte général* (Chapitre 2). Nous présentons dans un premier temps le contexte général des travaux de cette thèse qui s'articule autour du Cloud computing et de ses enjeux. Nous présentons ensuite la virtualisation qui est la technologie à la base du cloud, puis nous passons en revue les différents systèmes de fichiers et de stockage, et enfin la gestion de la mémoire dans le noyau Linux.
- *Cacol* (Chapitre 3). Ce chapitre porte sur la première contribution Cacol, une politique de gestion de cache s'exécutant dans le cache hôte, dont l'objectif est d'atténuer la duplication des pages du cache fichiers dans un environnement virtualisé.
- *Infinicache* (Chapitre 4). Ce chapitre présente la seconde contribution Infinicache, un système permettant d'exploiter la mémoire libre du centre de données, en proposant une extension du cache de fichiers dans les machines virtuelles.
- *Etat de l'art* (Chapitre 5). Nous présentons ici les différents travaux en lien avec les contributions de cette thèse.
- *Conclusion* (Chapitre 6). Nous apportons une conclusion et des perspectives d'amélioration des différentes contributions présentées dans cette thèse.

Chapitre 2

Contexte général

Contenu

Je présente ici le contexte général de ma thèse qui s'articule principalement autour du cloud computing (§ 2.1). Nous verrons ensuite la virtualisation, la technique majeure sur laquelle le cloud est basé (§ 2.2), puis les systèmes de stockages et de fichiers (§ 2.3), et enfin la gestion de la mémoire et du cache dans le noyau Linux (§ 2.4).

2.1 Cloud Computing

Les besoins en calculs informatiques sont en forte augmentation depuis plusieurs décennies. Les réseaux sociaux et les applications webs classiques sont de grands consommateurs de ressources informatiques mais aussi des domaines comme la physique avec par exemple, les prévisions météorologiques, ou encore la biologie. Plus récemment, le développement de l'intelligence artificielle, le big data, et le machine learning ont accentué cette tendance.

2.1.1 Définition

La définition du cloud computing a longtemps été débattue et a été source de confusions [86]. Pour simplifier la notion de cloud computing, ou d'informatique en nuage, on peut la considérer comme une externalisation de services. Le but principal est de pouvoir utiliser les mêmes matériels et environnements de travail que dans l'informatique classique mais sous la forme de services à distance. Le matériel et le logiciel s'exécutant alors dans des *centres de données*.

Une définition succincte est alors proposée par la loi française, qui s'appuie principalement sur la relation entre client et fournisseur de services de cloud :

> *Mode de traitement des données d'un client, dont l'exploitation s'effectue par l'internet, sous la forme de services fournis par un prestataire.* [2]

Une définition plus technique et largement acceptée par la communauté scientifique, est donnée par l'Institut national des standards et de la technologie (NIST), une agence du département du Commerce des États-Unis mais aussi un des plus anciens laboratoires de recherche au monde.

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and*

> *services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.* [102]

Ainsi, d'après le NIST, l'informatique en nuage est un modèle qui permet l'accès, via un réseau de télécommunications, de manière pratique et à la demande, à un ensemble de ressources informatiques configurables (e.g. réseaux, serveurs, stockages, applications et services). Ces ressources peuvent être rapidement acquises et rendues avec un effort de gestion minimal et une très faible interaction avec le fournisseur de services. Comme explicité par le NIST, le cloud est composé de cinq principales caractéristiques, trois modèles de services et quatre modèles de déploiements que nous allons détailler.

2.1.2 Caractéristiques

Les principales caractéristiques du cloud computing sont les suivantes :

— *Allocation à la demande.* Un client peut demander et acquérir des ressources de calcul comme du stockage sur un serveur ou de la puissance de calcul, de manière unilatérale et de manière automatique, ainsi sans intervention humaine, auprès du fournisseur de services.

— *Accès via un réseau.* Les ressources proposées sont disponibles via des moyens de communication standards. Elles permettent leurs utilisations via différentes plateformes du point de vue du client (e.g. ordinateurs fixes, téléphones mobiles, etc...).

— *Mutualisation des ressources.* Les ressources du fournisseur de service sont regroupées et mutualisées pour servir plusieurs clients à la fois. Elles sont donc allouées, libérées et réaffectées suivant les demandes des clients. En ce sens, un client ne sait jamais où sont exactement ses ressources car elles peuvent être déplacées à tout moment. D'où la dénomination de nuage/cloud computing.

— *Réduction des coûts.* Les ressources étant mutualisées, cela entraîne par conséquent une réduction des coûts pour le fournisseur grâce à l'économie d'échelle. Dit plus simplement, une ressource de type matériel aura un coût moindre à l'unité grâce à sa présence en grande quantité. Que ce soit pour l'achat du matériel, le refroidissement global du centre de données, la consommation électrique, et toutes autres dépenses liées à la gestion de matériels informatiques qui sont des réductions de coût pour le fournisseur. Le client réduit également ses coûts en supprimant par conséquent

cet investissement lourd qu'est l'achat du matériel.

— *Scalabilité rapide.* Les capacités de calcul fournies par le centre de données doivent s'adapter rapidement à la charge montante ou descendante émise par les clients, et de manière automatisée. Du point de vue du client, les ressources disponibles lui apparaissent comme infinies et il peut donc en demander autant qu'il désire en utiliser.

— *Services fournis mesurables.* Les ressources utilisées doivent être mesurées, contrôlées et rapportées, via des métriques appropriées (e.g. stockage, bande passante) fournissant ainsi une transparence quant à l'utilisation de ces ressources, à la fois au client et au fournisseur de services. En outre, cela permet également au fournisseur d'optimiser l'utilisation qu'il fait de son centre de données.

La principale qualité du cloud computing est ainsi la mutualisation des ressources et puissances de calculs informatiques, permettant une baisse d'un facteur 5 à 7 [86] du coût en électricité, en logiciels et en matériels grâce à l'économie d'échelle. Le client en outre bénéficie d'une apparence de ressources infinies disponibles à la demande, sans engagement et sans investissement de sa part. De plus, de part ses côtés quantifiable et mesurable, le cloud a développé le modèle du *pay-as-you-go*, ou plus simplement du paiement à l'utilisation. Un client paie exactement pour les ressources qu'il utilise sans se soucier de l'état d'utilisation globale du centre de données et surtout sans investissements lourds de sa part.

2.1.3 Classification

Il existe trois modèles de service [128] de cloud que le client peut utiliser. Ces trois modèles sont le IaaS (*Infrastructure as a Service*), le PaaS (*Plateform as a Service*), et le SaaS (*Software as a Service*). Leurs différences reposent sur la répartition des responsabilités entre le client et le fournisseur. En effet, selon qu'un client veuille plus ou moins de contrôle vis-à-vis du matériel et des logiciels, il choisira une des différentes solutions existantes, résumées en Figure 2.1.

2.1.3.1 IaaS

Ici, le client se voit attribuer une machine virtuelle à laquelle il peut accéder via une adresse internet. Généralement, le client peut spécifier les caractéris-

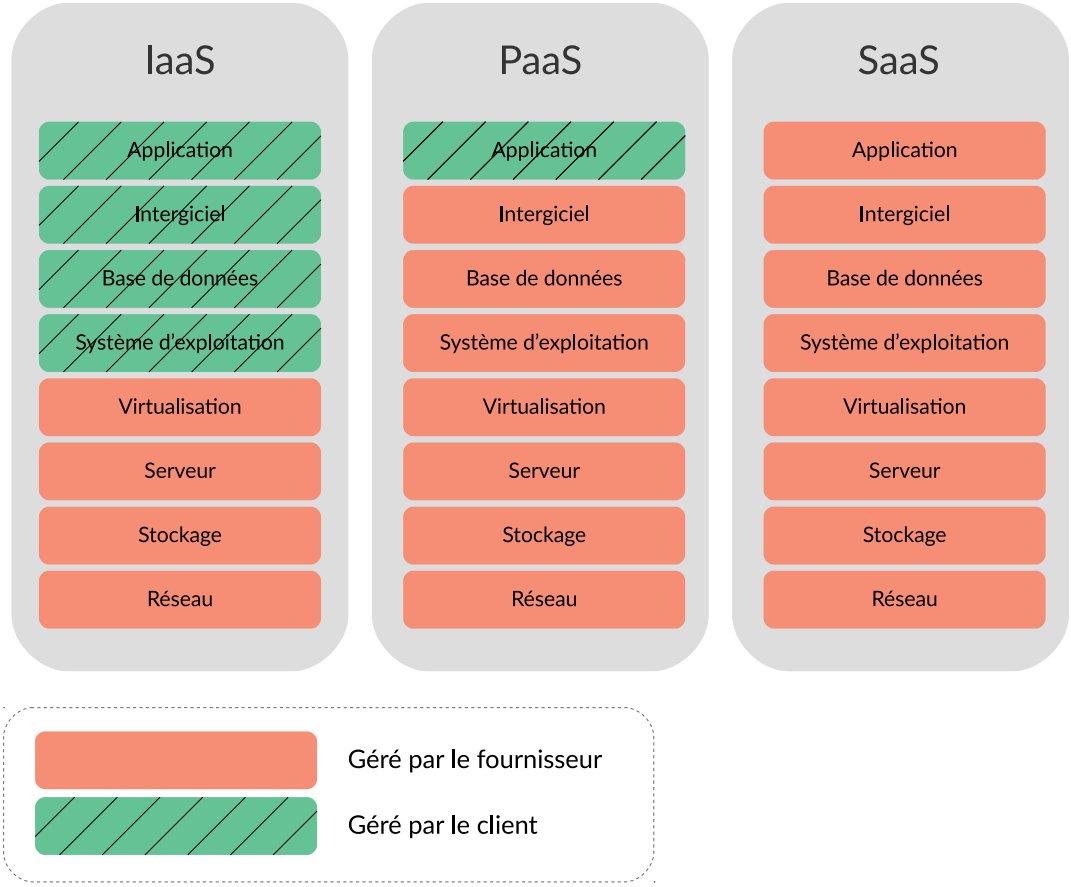


FIGURE 2.1 – **Les différents modèles de service du cloud** catégorisés par les responsabilités réparties entre fourniseur et client du cloud. Il existe trois modèles différents, le IaaS (*Infrastructure as a Service*), le PaaS (*Plateform as a Service*), et le SaaS (*Software as a Service*)

tiques de la machine comme par exemple le nombre de processeurs, la taille de la mémoire, ou encore la taille du disque de stockage. Sur cette dernière, le client peut faire fonctionner le logiciel qu'il désire, que ce soit le système d'exploitation ou les applications. Le fournisseur, quant à lui, a la responsabilité de contrôler et de gérer l'infrastructure du centre de données, c'est-à-dire de tout ce qui est matériel, les machines physiques et toute la connectivité relative au réseau. Par exemple, Amazon EC2 [7] est un IaaS.

2.1.3.2 PaaS

Le client peut, dans ce cas de figure, créer et déployer des applications en utilisant des langages de programmation, des librairies et des services proposés et supportés par le fournisseur de cloud. Le fournisseur est alors responsable de

FIGURE 2.2 – **Quelques exemples de services IaaS, PaaS et SaaS**

l'infrastructure matérielle, du réseau aux serveurs physiques mais aussi des systèmes d'exploitation, de l'environnement d'exécution (e.g. un serveur Tomcat) et du stockage disque. Le client contrôle seulement l'application qu'il souhaite utiliser. On peut citer entre autres Google App Engine [9].

2.1.3.3 SaaS

Le client utilise directement les applications proposées par le fournisseur de cloud, qui fonctionnent sur son infrastructure de cloud. L'application est alors accessible via différents types de support, que ce soit à travers un téléphone mobile ou une station de travail fixe. Le fournisseur de cloud gère entièrement l'application que ce soit l'infrastructure matérielle ou le déploiement de l'application dans le centre de données. Le client peut, dans certains cas, contrôler certains paramètres spécifiques à l'application. Dans cette catégorie SaaS, on peut citer des applications extrêmement connues comme Gmail, Dropbox, Slack, etc..

Quelques exemples de services connus existants dans ces différentes classes sont montrés en Figure 2.2.

2.1.4 Les différents modèles d'infrastructure

Nous venons de voir qu'il existait trois différents modèles de service que le client peut utiliser selon ses besoins. Ces modèles de service peuvent être déployés dans quatre types d'infrastructures [80] que sont le cloud privé, le cloud public, le cloud hybride et le cloud communautaire.

2.1.4.1 Cloud privé

Il s'agit d'une infrastructure destinée à une seule organisation privée et mise à disposition pour plusieurs utilisateurs de cette organisation. Les machines se trouvent dans un lieu appartenant à l'organisation, et sont gérées par eux-mêmes ou par une tierce partie. La principale caractéristique est de pouvoir contrôler où se situent les serveurs et qui peut y avoir accès, généralement les employés de l'entreprise ou de l'organisation. Dans ce genre de cloud, la confidentialité (*privacy*) est très importante, c'est-à-dire d'avoir les données confidentielles dans des lieux propres à l'entreprise, donc d'en limiter l'accès à un usage interne. Par exemple, Intel possède un cloud privé [51] que ses ingénieurs peuvent utiliser.

2.1.4.2 Cloud public

Ce type d'infrastructure a été créé pour être utilisé par le grand public ou des entreprises. Il est généralement géré et administré par une entreprise (Amazon Web services [8] par exemple) et à des fins commerciales. Les serveurs se trouvent sur le site du fournisseur de cloud. Dans le cas général, le fournisseur possède plusieurs sites et le client peut choisir selon sa préférence. Par exemple, Amadeus [6] est le principal fournisseur de solutions informatiques pour l'industrie du voyage, créé sous l'impulsion de plusieurs compagnies aériennes. Aujourd'hui, il permet à de très nombreuses compagnies aériennes d'utiliser des applications métiers en mode SaaS via un unique client. Un autre exemple de cloud public est Amazon EC2, qui est quant à lui un IaaS.

2.1.4.3 Cloud hybride

Cette infrastructure est la composition d'au moins deux autres types d'infrastructures (public, privé ou communautaire). Généralement, il s'agit d'un cloud privé d'une entreprise auquel on a ajouté une capacité de calcul supplémentaire via le cloud public pour pouvoir répondre aux pics de charge. Ces deux clouds restent néanmoins des entités distinctes.

2.1.4.4 Cloud communautaire

Ce cloud permet à un ensemble d'entreprises, organisations, entités ayant un même besoin de se rassembler et de profiter d'une solution de cloud unique. Il peut être géré en interne ou par une tierce partie. On trouvera dans la majorité des cas des clouds gérés par une université, une organisation gouvernementale ou même une combinaison de ces dernières. Par exemple, Grid 5000 [10] est un réseau de centres de données répartis entre plusieurs villes françaises pour permettre à la communauté scientifique spécialisée en informatique d'avoir accès à une puissance de calcul de l'ordre de 12000 coeurs pour des expériences en Cloud, Big Data, intelligence artificielle, etc...

Ainsi, nous pouvons avoir une vue d'ensemble du Cloud computing résumé et schématisé en Figure 2.3. Il est possible d'accéder au cloud computing via internet par différents supports, que ce soit mobile, machines fixes ou autres. Le cloud fournit des applications, du stockage, ou même des machines virtuelles selon le type de services (IaaS, SaaS ou PaaS). Enfin, le cloud s'appuie sur trois principales infrastructures qui sont le cloud public, le cloud privé et le cloud hybride.

Dans le cadre de cette thèse, nous nous intéressons plus particulièrement au IaaS, c'est-à-dire des infrastructures fournissant des machines virtuelles. La virtualisation est la technologie clé et principale permettant de faire fonctionner une telle infrastructure.

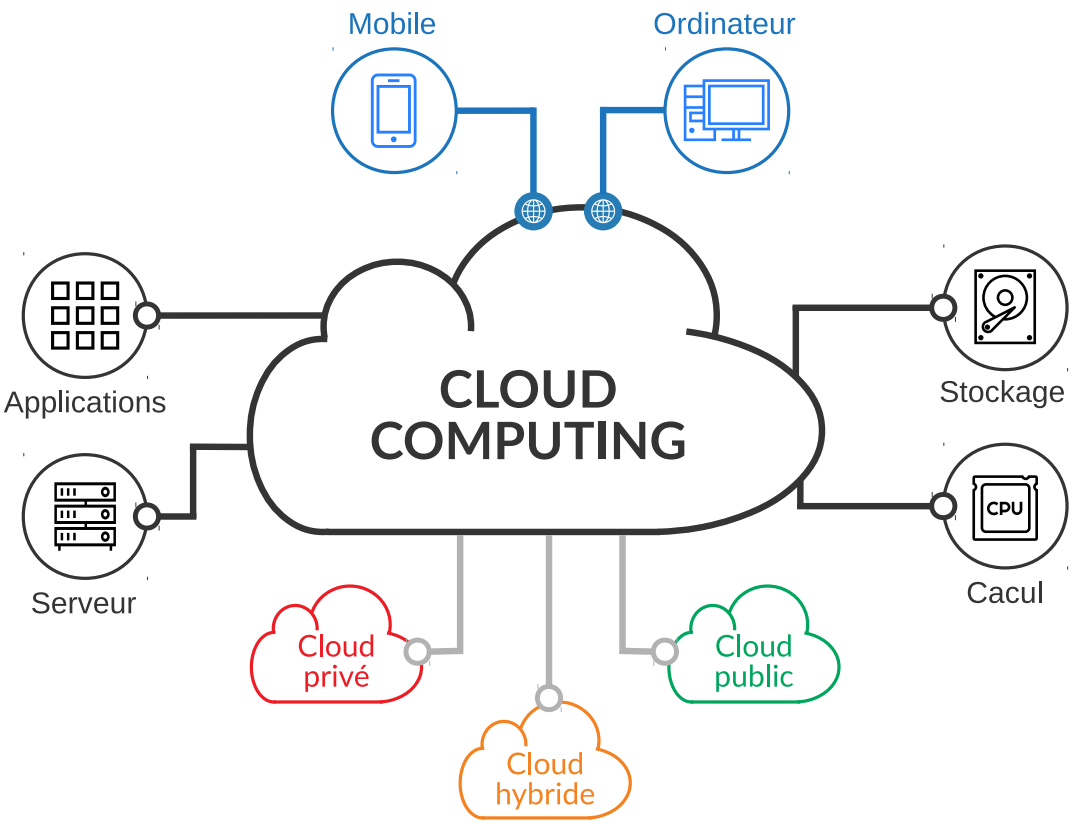


FIGURE 2.3 – **Vue d'ensemble du cloud computing**

2.2 La virtualisation

Nous abordons dans cette partie la virtualisation, fondation du cloud computing. Nous verrons une définition (§ 2.2.1), la couche logicielle principale de la virtualisation qu'est l'hypervieur (§ 2.2.2), et enfin les types de virtualisation existants, leurs défauts et leurs avantages (§ 2.2.3).

2.2.1 Définition

La virtualisation est la base du cloud computing. Malgré un essor relativement récent du cloud, les premiers travaux de recherche sur la virtualisation et les machines virtuelles remontent à une cinquantaine d'années [65, 66, 124].

La virtualisation est l'ensemble des techniques et logiciels permettant de faire fonctionner plusieurs systèmes d'exploitation (OS) sur une même machine physique (PM). Chaque système d'exploitation est isolé des autres dans ce

qu'on appelle une *machine virtuelle* (VM).

La couche permettant de faire l'abstraction du matériel et/ou du logiciel est communément appelée l'*hyperviseur*. Il permet de fournir un partitionnement et une isolation entre les VM. Il s'assure également de partager les ressources matérielles aux VM dans un environnement contrôlé.

Le système d'exploitation présent dans la VM est appelé l'OS invité ou le *guest OS*. Le système d'exploitation présent sur la machine physique est quant à lui appelé l'OS hôte ou *host OS*. Les différents guest OS peuvent communiquer entre eux via un réseau virtuel, purement logiciel, géré par l'hyperviseur mais aussi avec les autres machines physiques présentes sur le réseau auquel est connectée la machine hôte.

Nous pouvons voir une illustration de la différence entre plusieurs machines physiques classiques et une machine physique avec une couche de virtualisation en Figure 2.4. Nous pouvons observer déjà le principal avantage de la virtualisation qui est la mutualisation des ressources matérielles, socle du cloud computing.

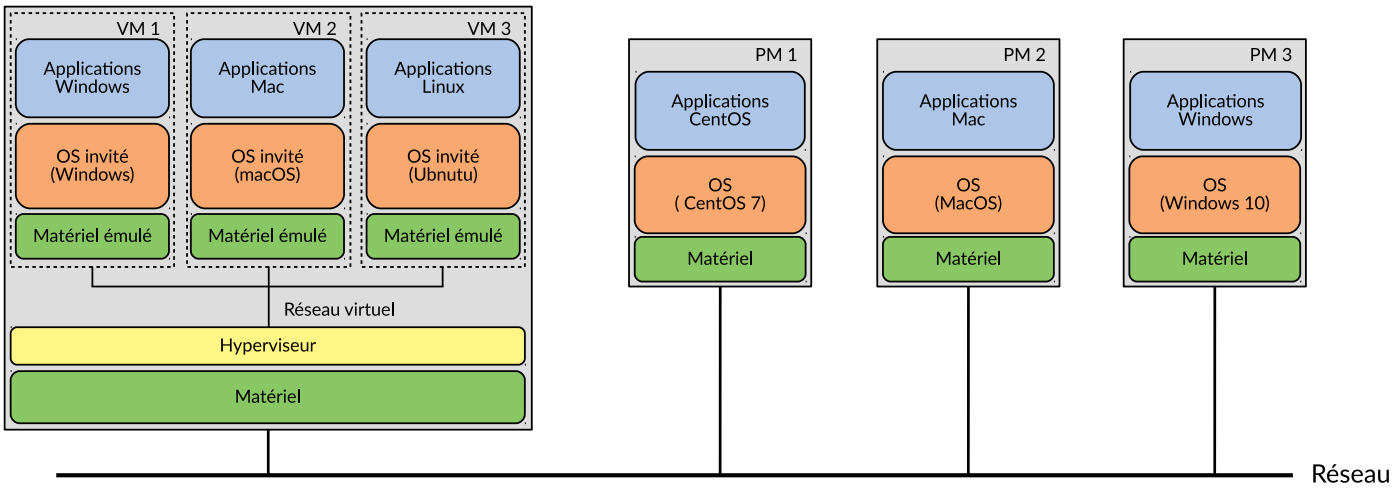


FIGURE 2.4 – **Exemples de machines virtuelles et machines physiques.** Chaque rectangle gris correspond à une machine physique.

2.2.2 L'hyperviseur

L'hyperviseur est la couche logicielle principale dans la virtualisation. Il fonctionne généralement au-dessus de l'OS hôte mais peut aussi bien être en

remplacement de celui-ci. Son rôle va être de gérer les ressources de la machine physique hôte et d'en distribuer l'accès aux VM. En outre, il permet l'isolation entre les VM, et contrôle l'utilisation du matériel. L'hyperviseur met en place l'isolation entre VM à plusieurs niveaux. D'abord, l'isolation des données, c'est-à-dire la confidentialité des données ; une VM ne pourra pas accéder au contenu de la mémoire d'une autre VM malgré le fait que leurs données soient présentes dans la mémoire de la même machine physique (i.e. sa RAM). Ensuite, il y a également une isolation des pannes, une VM ayant une erreur ne doit pas entraîner des pannes sur les autres VM. Enfin, une isolation des performances, les VM ne doivent pas influer sur les performances des autres VM, autrement dit les performances de chaque VM doivent être prédictibles.

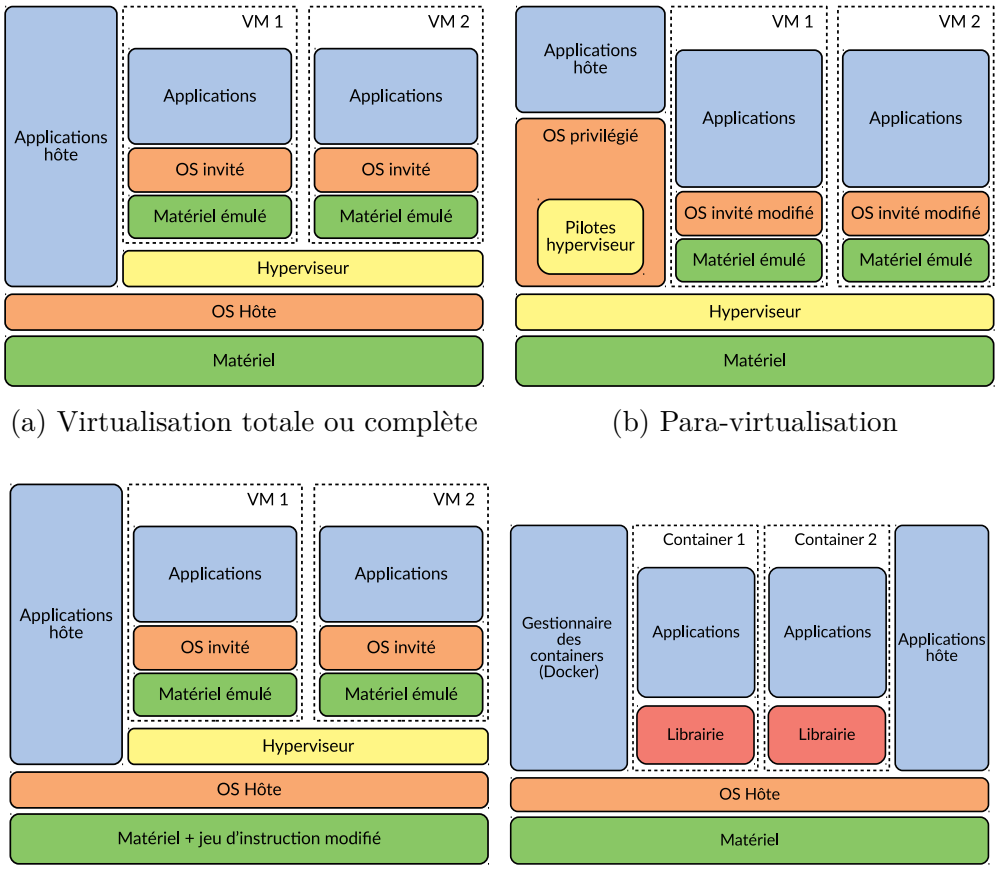
Du point de vue de la VM, le guest OS pense avoir accès à tout le matériel et en avoir le contrôle intégral, comme le processeur, la mémoire et la carte réseau. En réalité, l'hyperviseur fournit une abstraction du matériel qu'il gère lui-même et contrôle l'accès aux ressources. Cette abstraction et donc cet accès aux ressources peut être fait de différentes manières et l'on peut donc catégoriser les hyperviseurs. Il existe deux types d'hyperviseurs, les hyperviseurs de type 1, on parle alors de para-virtualisation, et les hyperviseurs de type 2, on est dans le cas de la virtualisation totale ou assistée par le matériel.

2.2.3 Les différents types de virtualisation

Il existe quatres principaux types de virtualisation : la virtualisation totale ou complète se basant sur un hyperviseur de type 2, la para-virtualisation avec un hyperviseur de type 1, la virtualisation assistée par le matériel (HVM) avec un hyperviseur de type 2 et enfin la conteneurisation qui n'utilise pas d'hyperviseur mais un logiciel utilisateur. Leur mode de fonctionnement est résumé en Figure 2.5.

2.2.3.1 Virtualisation totale

Nous utilisons un hyperviseur de type 2 dans ce cas de virtualisation. Il s'agit d'un logiciel émulant complètement le matériel. Nous pouvons voir en Figure 2.5a l'architecture d'une machine avec hyperviseur de type 2. La machine virtuelle fonctionne avec un système d'exploitation non modifié et n'a pas conscience qu'elle effectue ses opérations sur du matériel émulé. L'hyperviseur



(a) Virtualisation totale ou complète

(b) Para-virtualisation

(c) Virtualisation assistée par le matériel

(d) Conteneurisation

FIGURE 2.5 – **Les différentes types de virtualisation existantes**

se charge de traduire les instructions de la VM pour les faire exécuter sur la machine physique. Cette technique confère l'avantage de pouvoir faire fonctionner autant de types de matériel que l'on souhaite, par exemple il est possible d'avoir une machine virtuelle exécutant un OS avec un jeu d'instructions `x86` et une autre exécutant un jeu d'instruction ARM [81], le tout sur une machine physique avec une architecture classique comme par exemple `x86-64`. Néanmoins, la virtualisation totale est très lourde et lente car elle doit traduire chaque opération de la VM. Il s'agit historiquement du premier type de virtualisation qui a vu le jour car c'est la plus intuitive. Quelques exemples de logiciels de virtualisation totale sont Oracle VirtualBox [115], VMware Workstation [145], et Qemu [24].

2.2.3.2 Para-virtualisation

L'hyperviseur de type 1 s'exécute à la place de l'OS hôte comme indiqué dans la Figure 2.5b. Le développement de ce type de solution vient de la volonté de minimiser la surcouche logicielle de la virtualisation totale et obtenir de meilleures performances. Les guest OS sont modifiés et sont donc conscients qu'ils s'exécutent sur un environnement virtualisé. Ils ont à leur disposition des instructions appelées *hypercall*, fournies par l'hyperviseur, qui leur permettent de communiquer directement avec ce dernier. Cela réduit grandement les temps d'exécution car l'hyperviseur n'a plus qu'à rediriger la requête au matériel, mais tout en assurant l'isolation entre les VM. On exécute donc ici du code natif, c'est-à-dire du même jeu d'instructions. Ainsi, dans ce cas de virtualisation, il est nécessaire de modifier les systèmes d'exploitation invités et ces derniers devront exécuter le même jeu d'instructions que la machine hôte. Heureusement, dans les datacenters, les architectures utilisent principalement le jeu d'instructions `x86-64`. L'exécution d'une VM est alors beaucoup plus rapide que dans le cas de la virtualisation totale. Le précurseur de la para-virtualisation est le système Xen [23], mais il existe d'autres systèmes para-virtualisés comme Microsoft Hyper-V [154] ou VMware vShpere [68].

2.2.3.3 Virtualisation assistée par le matériel

Dans le cas de la para-virtualisation, on constate qu'il est nécessaire de modifier l'OS invité. Ainsi, la virtualisation assistée par le matériel, appelée HVM (*Hardware Virtual Machine*) dans Xen, permet de résoudre ce problème, en modifiant les processeurs pour inclure des instructions spécifiques à la virtualisation. Ainsi, l'OS invité ne sera pas conscient qu'il est exécuté dans un environnement virtualisé mais pourra obtenir des performances équivalentes à la para-virtualisation grâce à des instructions spécialisées notamment au niveau du matériel réseau et du stockage. L'isolation entre VM est donc réalisée au niveau matériel et non plus au niveau logiciel. L'architecture est exactement la même que la virtualisation totale et est présentée en Figure 2.5c à la différence que la HVM nécessite du matériel supportant la virtualisation, c'est-à-dire supportant la technologie Intel VT-x [153] ou AMD-SVM [40] pour un CPU, et par exemple SR-IOV [121] pour une carte réseau. Il faut savoir que ces technologies ne sont disponibles que pour les jeux d'instructions `x86` et sa version 64 bit uniquement. Par exemple, KVM [90] est un logiciel permettant de faire du HVM. Il est souvent associé à l'émulateur Qemu car ce dernier possède une

option pour s'associer à KVM et profiter de la virtualisation assistée par le matériel.

2.2.3.4 Conteneurisation

Il ne s'agit pas d'une technologie de virtualisation à proprement parler, mais plutôt d'un isolateur. Comme nous pouvons le voir en Figure 2.5d, les applications des conteneurs sont isolées entre elles, via un logiciel de conteneurisation. Le plus connu et le plus utilisé étant Docker [29]. Cette solution permet d'avoir des performances proches d'un système natif car les applications s'exécutent directement sur l'OS hôte. L'isolateur a seulement besoin de contrôler les limites d'utilisation de la mémoire et des autres ressources de la machine physique. Il doit également s'assurer de la bonne isolation et de la communication entre *Container*. Un autre logiciel connu de conteneurisation est LXC [131].

Les travaux de cette thèse s'appuient sur de la virtualisation assistée par le matériel, et donc un hyperviseur de type 2. Nous utiliserons KVM et Qemu.

2.3 Les systèmes de stockage dans un environnement virtualisé

Que ce soit dans le cloud ou dans l'informatique classique, le stockage d'information est fait sur un support physique, le disque. Un système de fichiers est placé au-dessus d'un disque, pour pouvoir y stocker les informations sous la forme de fichiers. Il existe différentes architectures de stockage, et de nombreux systèmes de fichiers que nous allons passer en revue.

2.3.1 Les architectures de stockage

Les applications logicielles peuvent conserver et lire leurs données à travers un système de fichiers et une interface que leur fournit le système d'exploitation. Les données sont ensuite enregistrées sur de la mémoire non volatile (des disques), à l'opposé de la mémoire vive qui perd ses informations une fois éteinte. Il existe alors trois architectures de stockage possibles [151, 132], le

DAS (*Direct Attached Storage*), le NAS (*Network Attached Storage*) et le SAN (*Storage Area Network*), résumées en Figure 2.6.

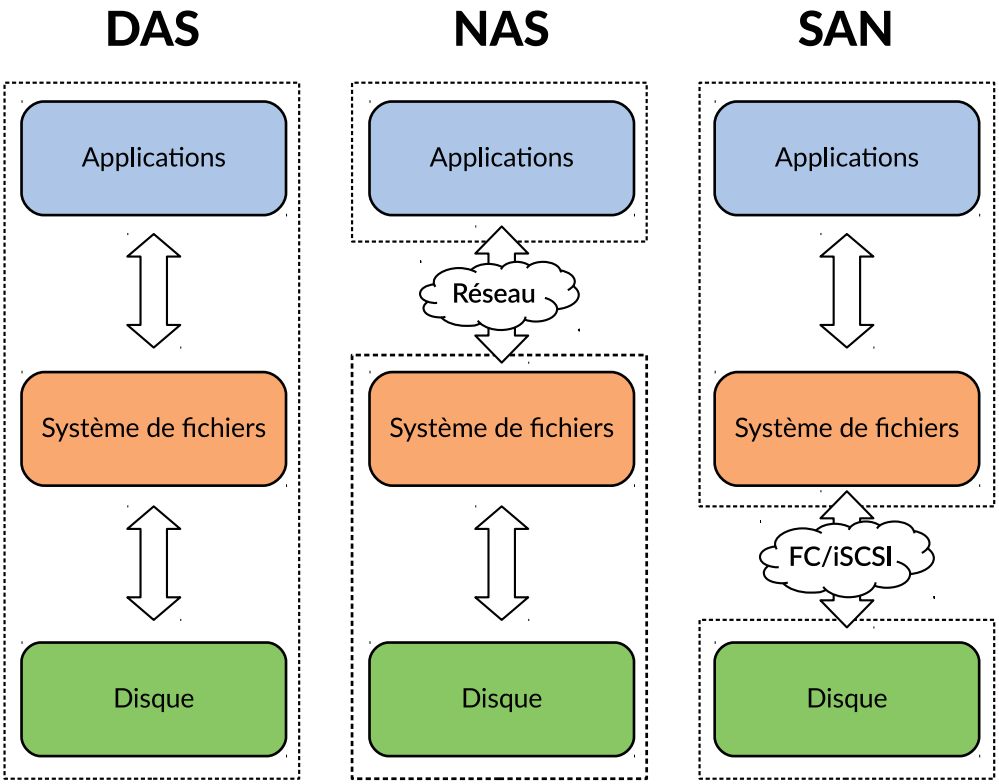


FIGURE 2.6 – **Systèmes de stockage existants**. Le SAN utilise un réseau et un protocole dédiés pour communiquer, comme par exemple le *Fibre Channel* (FC).

2.3.1.1 Direct-Attached Storage

Le Direct-Attached Storage (DAS) est l'attachement local d'un support de stockage. Il n'utilise pas le réseau, contrairement aux NAS et SAN. Ces supports de stockage peuvent être des disques durs magnétiques et électroniques, ou des disques externes comme des clés USB. Un DAS typique est composé simplement d'un disque connecté directement à la machine via un *Host Bus Adapter* (HBA). Les principaux protocoles utilisés sont SATA [76], NVMe [113], SCSI [134], et USB [16].

Le disque n'est accessible que depuis l'hôte auquel il est attaché, c'est sa principale différence avec les autres architectures. Aussi, il est possible de sécuriser les données à l'aide de la technologie RAID [119] (*Redundant Array of Inexpensive Disks*). Cette dernière est suivie d'un numéro entre 0 et 6, par

exemple RAID 1. RAID 1 consiste à dupliquer les données à l'identique sur deux disques distincts. En cas de défaillance d'un disque, les données sont récupérables sur l'autre disque. Les différents niveaux de RAID fournissent différents compromis entre chacune des caractéristiques comme la fiabilité, la disponibilité, les performances et la capacité.

2.3.1.2 Network-Attached Storage

Un NAS, ou serveur de stockage en réseau, est un serveur de fichiers autonome, qui fournit, à travers un réseau Ethernet ou LAN, du service de stockage de fichiers. Il est spécialement développé pour du partage de fichiers. Ces principaux avantages sont d'être accessible simultanément par plusieurs clients tout en garantissant la cohérence des données, et d'être facilement augmenté en taille par l'ajout de disques supplémentaires, de manière transparente pour l'utilisateur.

Les protocoles fournis par un NAS utilisent un réseau IP, et sont par exemple des protocoles réseaux fichiers classiques comme NFS [120] (*Network File System*), CIFS [70] (*Common Internet File System*) ou des protocoles internet comme FTP [125] (*File Transfer Protocol*), et HTTP.

Bien que le NAS soit adapté au partage de fichiers en réseau, les protocoles du NAS ne sont pas adaptés aux applications intensives en I/O comme une base de données, ou du traitement vidéo. Pour cela, il existe le SAN.

2.3.1.3 Storage Area Network

Un SAN, un réseau de stockage, est un réseau spécialisé permettant d'assembler plusieurs disques et proposer une vue homogène d'un ensemble de stockage. L'utilisateur, généralement un système d'exploitation, le voit comme un seul disque, dans le cas de Linux comme un block device. On peut alors y placer le système de fichiers que l'on veut, comme vu en Figure 2.6.

Les principaux avantages d'un SAN sont sa latence, son debit et sa qualité de service grâce à un réseau spécialisé qui le caractérise. En effet, il existe plusieurs types de SAN, et le plus répandu est le *Fibre Channel* (FC) [88] SAN et utilise le protocole FCP (*Fibre Channel Protocol*). D'autres exemples de technologies réseaux existantes sont le Gigabit Ethernet associé au protocole

iSCSI (*Internet Small Computer System Interface*) [133], ou l'Infiniband et son protocole iSER (*iSCSI Extensions for RDMA*) [37].

Un SAN est également évolutif, on peut ajouter ou enlever des disques du réseau à volonté pour ainsi augmenter ou diminuer la taille de stockage. Il est aussi pourvu de fonctionnalités de réplication ou de sauvegarde asynchrone permettant d'empêcher tout point unique de défaillance des disques, c'est-à-dire que lorsqu'un unique disque tombe en panne, le système n'est pas impacté.

2.3.2 Les systèmes de fichiers

Un système de fichiers (FS pour *File System*) est un moyen d'organiser de manière hiérarchique un ensemble de données sous forme de fichiers, au sein d'un volume, qu'il soit logique ou physique. Il en existe de très nombreux et différents, chacun adapté à une situation, mais ils peuvent être classés en deux catagories, les systèmes de fichiers locaux et les sytèmes partagés.

Les systèmes de fichiers locaux sont les plus connus et sont utilisés dans les DAS. Parmi ces systèmes de fichiers, on peut citer `ext2`, `ext3`, `ext4` (*Extented FS version x*) [152, 100] pour Linux, les systèmes `exFAT` [104], `FAT32` [170], `NTFS` [105] pour Windows ou encore `BTRFS` [130] et `xFS` [157]. Un dernier système de fichiers local spécifique et très utile est le `swap`.

Les systèmes de fichiers partagés (*shared disk file systems*) peuvent être montés sur plusieurs machines simultanément. Ils sont utilisés dans les NAS ou SAN. Une grande sous-catégorie des systèmes de fichiers partagés est celle des systèmes de fichiers distribués (DFS pour *Distributed File System*). Ils ont la particularité de stocker les données sur plusieurs disques ou machines. Les systèmes de fichiers distribués sont donc accessibles via la réseau par plusieurs clients à la fois, et le stockage se fait sur plusieurs serveurs. Leur but est de fournir à l'utilisateur, et de manière transparente, le même accès à un système de fichiers qu'un système local. En plus de cela, un système distribué fournit d'autres services comme une transparence de localisation, le client ne sait pas sur quelle machine physique se trouve le fichier ; de l'accès concurrent, les modifications d'un fichier sont cohérentes pour tous les utilisateurs ; ou enfin de la réplication pour éviter les pertes.

Parmi les systèmes de fichiers partagés les plus connus, on retrouve `GPFS` [135] (*General Parallel File System*) d'IBM, et `CSV` [49] (*Cluster Shared Volumes*) de

Microsoft. La plupart des systèmes de fichiers partagés sont également distribués comme `CephFS` [160] et `GlusterFS` [25] de ReadHat, `Lustre` [156] qui vient de la contraction de Linux et Cluster, `GFS` [64] (*Google file system*) le système de fichiers interne de Google, ou encore `HDFS` [142] (*Hadoop distributed file system*) le système de fichiers du framework de big data Apache Hadoop.

Nous avons vu, dans cette section, les différentes architectures de stockage et systèmes de fichiers existants. Dans le cadre de cette thèse, nous n'allons pas utiliser de stockage par réseau mais simplement des DAS pour des raisons de praticité de développement et d'évaluations. Toutefois, il est possible d'étendre ces travaux à des systèmes de fichiers distribués sur un NAS ou SAN. Concernant les systèmes de fichiers, nous utiliserons les systèmes `ext4` sur des machines Ubuntu, et `xFS` pour les machines sur CentOS.

2.4 Gestion de la mémoire et cache

Le système de cache dans un système d'exploitation permet d'accélérer les futurs accès au disque (aussi appelés I/O [1]). Un accès disque étant coûteux en terme de temps d'attente par rapport à un accès à la mémoire, il est important de les éviter en gardant en mémoire les données les plus utilisées ou en anticipant et en pré-chargeant certaines données.

2.4.1 Temps d'accès au disque

Les disques durs magnétiques (HDD [2]) ont toujours été l'un des points faibles des systèmes de part leur lenteur, et ainsi ont été sujets à de nombreuses optimisations [162, 73, 50, 53, 166]. Un HDD est composé de plateaux, d'un bras et d'une tête de lecture. Les plateaux tournent à une vitesse constante, comprise entre 3600 et 15000 tours par minute. Le temps moyen de lecture d'un segment de 512 octets sur un disque dur magnétique est de l'ordre de 5 millisecondes [167]. On appelle également ce temps, la *latence*. La latence est due à deux facteurs : la vitesse de rotation du disque ; il faut dans le pire des cas faire un tour entier pour obtenir l'information que l'on cherche, le deuxième facteur est le temps du positionnement de la tête de lecture.

1. I/O vient de Input/Output c'est-à-dire les Entrée/Sortie
2. Hard disk drive

	Latence lecture de 512 octets	Bande passante	Facteur de latence par rapport au HDD
HDD	5 ms	100 Mo/s	x 1
SDD	70 μs	3.2 Go/s	x 71
RAM	80 ns	60 Go/s	x 62 500

TABLE 2.1 – **Caractérisation de différents types de mémoire** selon leur latence de lecture et leur bande passante.

Les SSD [3] ou disques électroniques sont des disques basés sur une mémoire non volatile, généralement de la mémoire flash [45]. Cette technologie existe depuis plus d'une trentaine d'années mais ne s'est développée réellement que depuis une dizaine d'années pour des raisons de coût et de taille. Son principe est de stocker l'information sur des transistors. Ainsi, à la différence du HDD qui se sert du champ magnétique pour écrire sur le disque, ici ce sont les électrons contenus ou non dans le transistor qui indiquent l'information. Ce principe, fondamentalement différent d'un disque dur classique, fait qu'il ne possède pas d'attente engendrée par la rotation des plateaux et ainsi a une latence bien plus faible. Le temps moyen de lecture d'un segment de 512 octets sur un disque électronique est d'environ 70 microsecondes [164].

Pour pallier ces temps de latence longs et donc pour pouvoir accéder efficacement aux informations stockées sur un disque, la solution la plus répandue a été d'implémenter un cache du disque. Pour cela, les données que l'utilisateur a besoin de chercher sur le disque sont stockées temporairement sur de la mémoire plus rapide, souvent de la mémoire vive volatile (RAM). Ainsi, lorsque l'utilisateur accède plusieurs fois à la même donnée, il est intéressant d'en garder une copie en mémoire et de travailler avec cette dernière pour éviter des accès au disque inutiles. Le temps moyen de lecture d'un segment de 512 octets sur de la mémoire vive est d'environ 80 nanosecondes [164].

Les temps de latence des différents types de mémoire sont résumés en Table 2.1. Ainsi, nous pouvons observer l'importance d'un système de cache dans un système d'exploitation, au vu des différences de latence qui sont de plusieurs ordres de magnitude. L'objectif d'un système de cache est donc d'éviter au maximum des accès aux disques. Pour cela, il est possible de mettre en place différentes stratégies, c'est-à-dire des algorithmes d'éviction de cache, pour savoir quelle information conserver en mémoire, mais aussi se servir du pre-fetching pour anticiper certains accès au disque.

3. Solid-State Drive

2.4.2 Gestion de la mémoire dans Linux

Dans Linux, le sous-système *Memory management*, défini dans le noyau dans le dossier `mm/`, est, comme son nom l'indique, responsable de la gestion de la mémoire. Plus précisément, il est responsable de la mémoire virtuelle, de l'allocation de la mémoire pour le noyau et les programmes utilisateurs, du chargement des fichiers en mémoire et bien d'autres choses.

En terme d'architecture, la mémoire est dans un premier temps découpée et répartie en plusieurs noeuds s'il s'agit d'une machine NUMA (Non-Uniform Memory Access) [30], ensuite entre plusieurs zones mémoire qui distinguent la mémoire accessible seulement au noyau ou à tout le monde, la mémoire accessible en DMA, etc.. La mémoire, dans ces zones, est finalement partagée en pages de 4096 octets. Il existe différents types de pages selon leur utilisation, détaillés ultérieurement. L'unité de travail pour le Memory management est ainsi la page. Il lui est impossible d'allouer ou de modifier des zones mémoires avec une granularité plus fine.

Les processus accèdent à la mémoire à travers des adresses virtuelles. Ces adresses sont automatiquement traduites par le processeur en adresses physiques.

Les nœuds. De nombreuses machines multi-processeurs de nos jours sont des systèmes NUMA. Comme son nom l'indique, l'accès à la mémoire n'est pas uniforme, autrement dit la latence d'accès à la mémoire est différente selon sa "distance" au processeur. Ces différentes portions de mémoire sont appelées banques. Chaque banque mémoire est référencée par un *nœud* dans le noyau Linux, et chaque nœud possède un sous-système de la gestion de la mémoire. Chaque nœud a ainsi son propre ensemble de zones, de listes de pages libres, et un ensemble de compteurs variés.

Les zones mémoire. Souvent le matériel impose des restrictions sur le mode d'accès de chaque portion de mémoire. Dans certains cas, il n'est pas possible d'adresser de manière directe, i.e. en DMA, toute la mémoire adressable. Ainsi, Linux regroupe les pages selon l'usage que l'on peut en faire. La zone ZONE_DMA contient la mémoire utilisable par les périphériques en DMA, ZONE_HIGHMEM contient la mémoire qui n'est pas "mappée" de manière permanente dans la zone d'adressage du noyau et ZONE_NORMAL contient

les pages normalement adressables. La configuration des zones est dépendante du matériel et ainsi certaines architectures peuvent ne pas définir toutes les zones.

Le page cache. La mémoire physique est volatile et le principal cas d'usage pour avoir des données en mémoire est de les lire depuis un fichier sur le disque. A chaque lecture d'un fichier, ces données sont conservées dans le page cache pour éviter une grande latence lors de futures lectures ou écritures, car les données n'auront plus à être lues depuis le disque.

De même, lors de l'écriture dans un fichier, les données sont écrites dans le page cache et seront écrites plus tard de manière asynchrone sur le disque, évitant ainsi la latence du disque. On parle de politique d'écriture *writeback*. La politique *write-through*, quant à elle, écrit de manière synchrone toute modification de fichier sur le disque. Par défaut, la politique writeback est utilisée. Dans ce cas, la page écrite est marquée comme *dirty* et lorsque le noyau a besoin de réutiliser cette page, il prend soin de synchroniser son contenu sur le disque avant de la récupérer pour d'autres usages. Ce genre de page est appelée *page fichier* car elle contient des données de fichiers.

La mémoire anonyme. Dans le cache, nous pouvons distinguer un deuxième type de page, appelé *page anonyme*. Ces pages représentent la mémoire qui n'est pas enregistrée sur un disque par un système de fichiers. Elles peuvent être allouées de manière implicite pour le tas ou la pile (*stack* ou *heap*) d'un programme ou bien explicitement via l'appel système `mmap`. De manière générale, ces pages définissent la mémoire virtuelle à laquelle un programme peut accéder. Les pages peuvent être écrites et lues comme n'importe quelle autre page. Ces pages seront de même marquées comme dirty après une écriture, et lorsque le noyau désire récupérer les pages, leur contenu sera, dans ce cas, sauvegardé sur le disque swap, on parle de *swap out*.

Politique d'éviction. Durant la vie du système, une page physique peut contenir différents types de données. Elle est caractérisée de différentes manières, comme une page avec le contenu d'un fichier du disque, une page avec de la mémoire anonyme, une page de l'allocateur, etc... Les données peuvent être les données internes du noyau, des données lues depuis un système de fichiers, de la mémoire allouée par `kmalloc` pour des programmes utilisateurs ou simplement

la page peut être inutilisée.

Selon l'utilisation actuelle de la page, elle est traitée de manière différente par le système de la gestion de la mémoire. Ainsi, les pages qui peuvent être libérées, donc évincées du cache à tout moment, sont appelés des pages récupérables (*reclaimable*). Ce sont les pages dont le contenu est déjà présent sur le disque et qui servent donc de cache pour un accès plus rapide, et les pages dont on peut écrire le contenu sur le disque, c'est-à-dire dans les zones de swap. Ce sont les pages fichiers et les pages anonymes.

Linux utilise une politique d'éviction du cache appelée LRU-2 [114] (*Least recently used*). Les pages récupérables sont conservées dans deux listes différentes, les pages actives et les pages inactives [4], et rangées de la plus ancienne à la plus récente. Les pages accédées une seule fois sont ainsi placées dans la liste inactive et celles plus d'une fois dans la liste active. Pour récupérer de la mémoire, la politique d'éviction consiste à sélectionner les pages les plus anciennes, et donc en tête, de chacune des deux listes. Les pages de la liste inactive sont évincées du cache, et celles de la liste active sont déplacées en queue de la liste inactive.

Comme il existe deux catégories de pages récupérables, quatre listes sont utilisées par la politique d'éviction : la liste des pages actives et la liste des pages inactives anonymes, et la liste des pages actives et la liste des pages inactives des fichiers.

Les pages contenant les structures internes du noyau ou des buffers DMA sont généralement des pages que l'on ne peut pas récupérer tant que l'utilisateur ne les libère pas. On parle de pages fixées en mémoire (*pinned memory*) et elles sont classées dans une cinquième liste, les pages non récupérables (*unreclaimable*).

Réclamation mémoire. Il existe plusieurs chemins pour récupérer de la mémoire dans Linux et les principaux sont compilés dans la Figure 2.7. On y trouve deux techniques principales de récupération des pages mémoire : l'une est synchrone et l'autre asynchrone. Un troisième et dernier moyen de récupérer de la mémoire est lors de l'hibernation de la machine physique, c'est-à-dire lorsque la machine passe en veille profonde. Dans ce cas, l'objectif est de récupérer toute la mémoire et de la sauvegarder sur un support de disque dur.

4. On parlait aussi avant de pages chaudes et froides

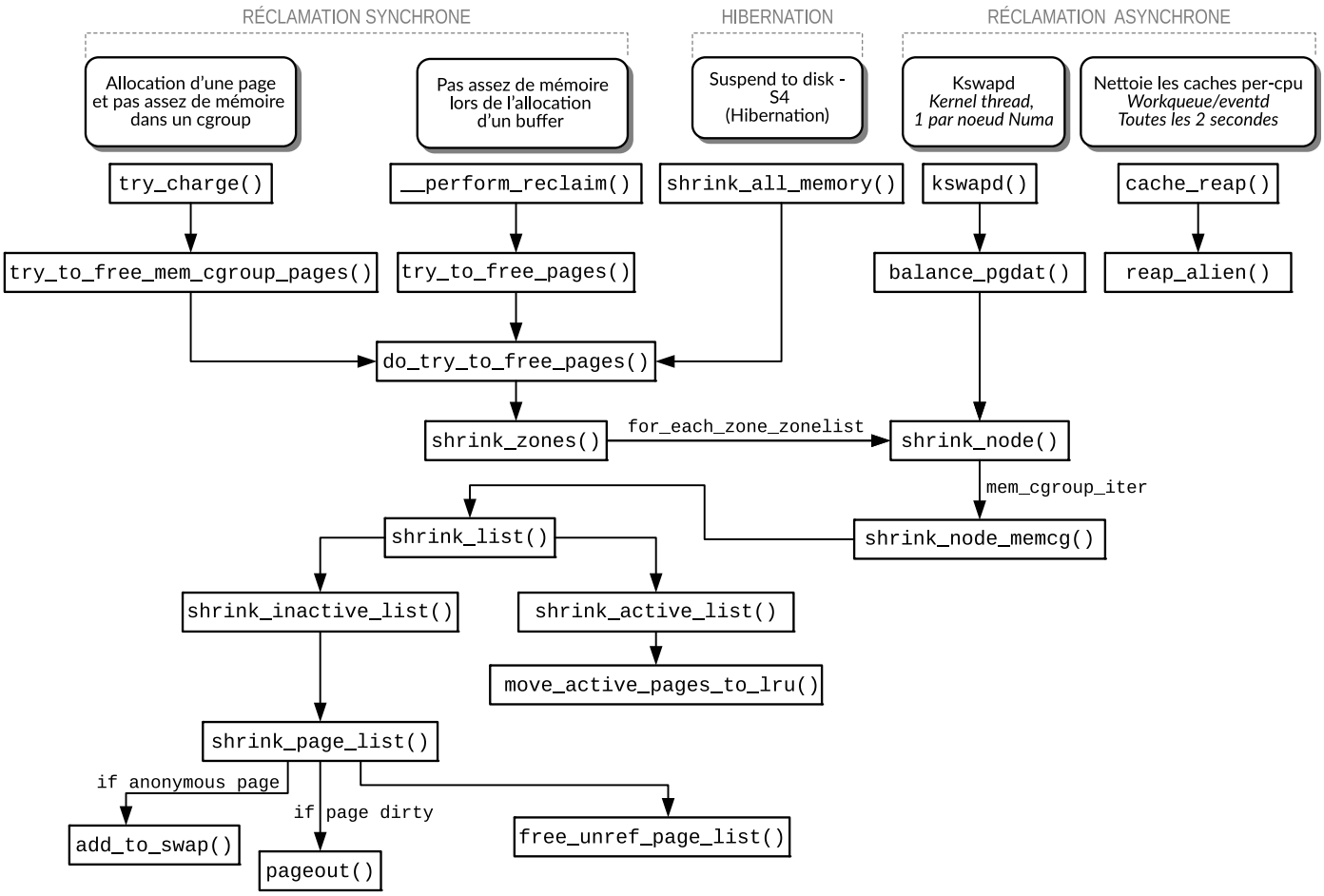


FIGURE 2.7 – **Chemin du code de la réclamation mémoire dans Linux**. Conforme à la version 5.1.0

Toutes les pages fichiers sont écrites sur le disque si besoin, puis jetées, les pages anonymes sont quant à elles écrites sur la partition swap, qui est aussi généralement un disque dur. La mémoire vive ne contient alors que des informations qui peuvent être retrouvées sur un disque dur, la mémoire est alors éteinte et perd toutes ses données. La machine peut alors se mettre en veille profonde, appelée plus techniquement l'état ACPI S4 [75].

Récupération asynchrone. Il s'agit du chemin de récupération de la mémoire le plus utilisé car le module noyau `kswapd` se déclenche, en arrière-plan pour récupérer de la mémoire, dès qu'il y a une pression sur cette dernière. Comme nous pouvons le voir sur la Figure 2.8, il existe trois paliers (*watermarks*) qui activent ou désactivent la récupération de mémoire faite par `kswapd`. Ce module est lancé pour chaque nœud mémoire, il y a donc plusieurs modules `kswapd` s'exécutant en parallèle sur une machine NUMA.

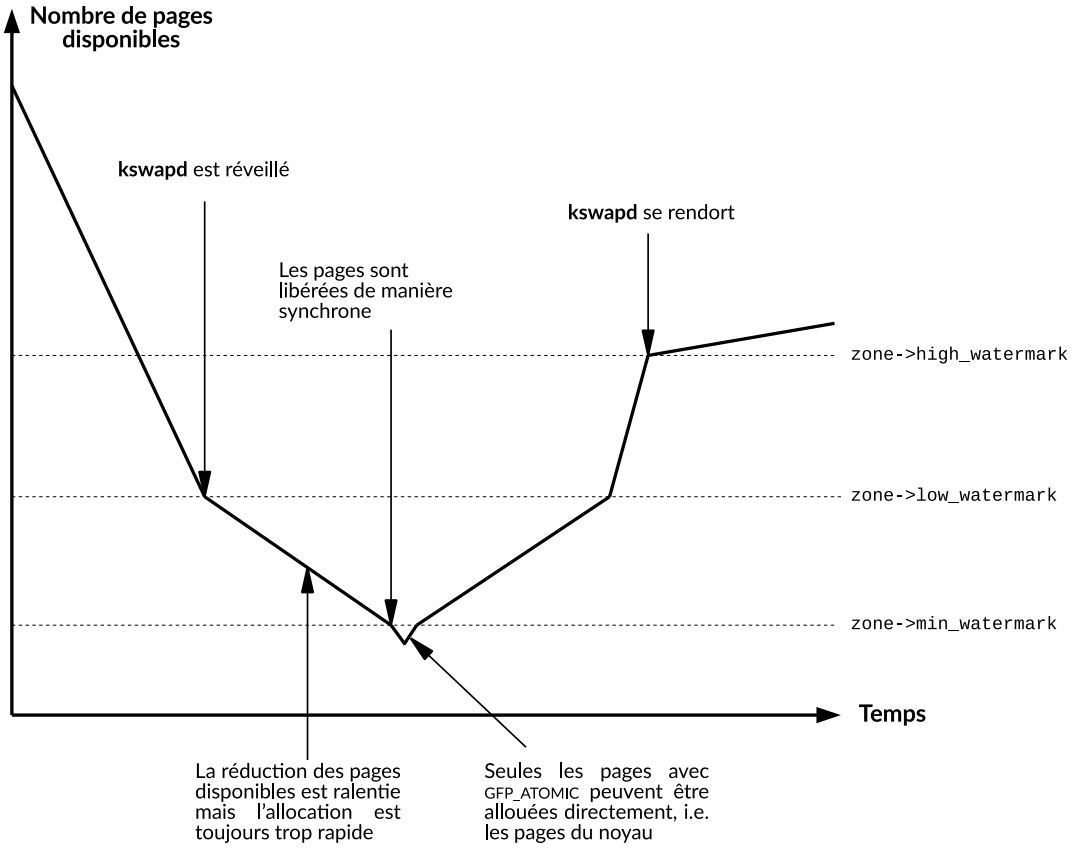


FIGURE 2.8 – **Les différents paliers utilisés pour la gestion de la mémoire**. Aucune récupération asynchrone de mémoire (faite par `kswapd`) n'est effectuée au-dessus du premier palier (`high_watermark`). Il est par défaut, pour les systèmes Linux, d'une valeur de 5% de la mémoire disponible. C'est la raison pour laquelle la mémoire, dans les systèmes Linux, apparaît très souvent comme entièrement utilisée [13].

Son fonctionnement est relativement simple, il scanne chaque zone mémoire (`shrink_node`) les unes après les autres (un *round-robin* en somme) et essaie d'en récupérer des pages. Dans chaque zone, il y a quatre listes de pages récupérables précédemment définies. `Kswapd` réduit chaque liste (`shrink_list`) et récupère donc les pages les plus anciennes de chacune. Sa politique est de maintenir un ratio de 1/3 entre pages actives et pages inactives. Le ratio entre pages fichiers et pages anonymes est défini par l'utilisateur via l'option `swapiness` de Linux. De ces ratios, `swapd` en déduit alors sur laquelle des quatre listes, il doit récupérer le plus de pages.

Enfin, une page évincée du cache est ajoutée au swap s'il s'agit d'une page anonyme (`add_to_swap`), ou elle est écrite sur le disque s'il s'agit d'une page fichier qui a été modifiée (`page_out`). Toutes ces pages sont finalement réinitia-

lisées et placées dans la liste des pages libres et disponibles grâce à la méthode `free_unref_page_list`.

Récupération synchrone. Elle apparait dans les cas de pénurie de mémoire. Ainsi, lorsqu'une application tente d'allouer de la mémoire et qu'elle dépasse sa limite de mémoire disponible (vérifiée par `try_charge`), l'allocateur va, de manière synchrone, evincer du cache des pages de l'application. Ainsi, on parle d'allocation lente (*slow path*) car l'allocation n'est pas instantanée comme c'est le cas de toutes les autres allocations normalement. Pourtant, le système global peut posséder assez de mémoire libre et disponible, seulement une application peut être contrainte par une limite d'utilisation de la mémoire, définie à travers des Cgroup. Un Cgroup est simplement un regroupement de processus auquel on peut appliquer des limitations, que ce soit mémoire, cpu ou autres.

De la même manière, s'il n'y a plus assez de mémoire globalement, c'est-à-dire lorsque le palier `min_watermark` est atteint, l'allocation se fait de manière synchrone (`__perform_reclaim`). Comme indiqué, les allocations du noyau ne sont pas concernées par cette limite pour des raisons évidentes d'inter-blocages, en particulier lorsque le module noyau `kswapd` cherche justement à libérer des pages en allouant les quelques buffers nécessaires à son fonctionnement.

Pour résumer, la mémoire est partitionnée, dans un premier temps, entre les nœuds NUMA, s'ils existent, puis en différentes zones. Il est possible de créer des Cgroup pour restreindre l'utilisation de la mémoire. Ainsi, par zone et par Cgroup, il existe 5 listes contenant les pages ayant du contenu actuellement en mémoire, les pages actives et inactives anonymes, les pages actives et inactives de fichiers, et les pages fixées en mémoire. Des statistiques sont mises en place pour désigner quelle file a le plus besoin d'être réduite en taille. Un module noyau, `kswapd`, est créé pour récupérer de la mémoire en arrière-plan dès qu'il y a une pression mémoire. Dans le cas d'une pression trop importante, l'allocation de la mémoire ne se fait plus instantanément mais est contrainte de patienter le temps que le système soit dans un état plus stable.

Ainsi, l'allocation de la mémoire et sa réclamation sont, dans le cas général, décorrélées et vivent indépendamment. L'allocation de mémoire apparait comme instantanée et comme s'il y avait tout le temps de la mémoire disponible. Le module noyau `kswapd` récupère la mémoire des pages récupérables pour faire en sorte qu'il y ait toujours de la mémoire disponible et prête à être allouée.

2.5 Synthèse

Dans ce chapitre, je vous ai présenté le contexte général de ma thèse. Cette thèse se place dans l'environnement du cloud computing (§ 2.1), un terme désignant l'infrastructure matérielle et logicielle fournissant des ressources qui peuvent être des services logiciels ou des machines virtuelles, à travers un réseau. Ces ressources paraissent infinies et lointaines, comme un nuage. Nous nous intéressons dans ces travaux, plus particulièrement, aux infrastructures IaaS, qui fournissent des machines virtuelles aux utilisateurs.

La technologie majeure du cloud computing est la virtualisation (§ 2.2). Cette dernière rend possible la division d'une machine physique en plusieurs machines virtuelles. Dans notre cas, nous utilisons de la virtualisation assistée par le matériel, avec donc un hyperviseur de type 2. Il s'agit du logiciel Qemu qui, à la base, est conçu pour de la virtualisation totale, mais qui, couplé avec KVM, permet de faire de la virtualisation assistée par le matériel et ainsi d'obtenir de très bonnes performances sur le noyau Linux.

Nous employons différentes technologies de système de fichiers et de stockage (§ 2.3) selon les contributions. En terme de système de fichiers, mes travaux reposent sur `ext4` et `xfs`, qui sont les systèmes par défaut sur les systèmes d'exploitation Ubuntu et CentOS. Les travaux présentés sont évalués à l'aide d'un système de stockage DAS, c'est-à-dire tout simplement un disque directement connecté à la machine et non un stockage sur le réseau, bien que certains travaux puissent aussi s'appliquer sur des NAS ou SAN.

Enfin, nous avons vu comment est gérée la mémoire dans le noyau Linux (§ 2.4). La partie qui nous intéresse est celle liée à la réclamation de la mémoire faite par le cache. Plus particulièrement, les pages liées à des fichiers, et non celles liées à de la mémoire anonyme.

2.6 Problématique

Comme nous avons pu le voir, nous nous intéressons à la gestion de la mémoire dans un environnement de cloud virtualisé. La mémoire est une ressource limitée et aujourd'hui même la ressource limitante [97]. Nous essayons donc d'en optimiser l'utilisation et donc d'améliorer les performances globales

du système d'exploitation. Nous nous posons ainsi la problématique suivante : Comment maximiser l'utilisation du cache de fichiers dans un environnement de cloud virtualisé ?

Les travaux dans cette thèse sont composés de deux contributions :

— *optimiser l'utilisation du cache*. La première, Cacol (Chapitre. 3), va répondre à la problématique de duplication des pages du cache. En effet, dans le contexte d'un environnement virtualisé, et de part le principe même d'un cache, les pages sont amenées à être présentes dans le cache de l'hyperviseur et dans le cache de la machine virtuelle. Les deux caches étant sur la même mémoire physique, il y a donc une perte de mémoire. Notre système Cacol tente donc d'atténuer ce phénomène. Sa principale caractéristique est de s'exécuter au niveau de l'hyperviseur, avec la machine virtuelle agissant comme une boîte noire, dont on ne peut voir que les entrées et les sorties.

— *étendre la capacité du cache*. La seconde contribution, Infinicache (Chapitre. 4), étend la mémoire du cache à l'aide d'un réseau à très haute vitesse (Infiniband). Pour cela, il exploite la mémoire non utilisée du centre de données, pour la fournir aux machines virtuelles qui en ont besoin. Ainsi, l'extension du cache permet de réduire significativement les accès disques et d'améliorer ainsi les performances.

Chapitre 3

Cacol

Contenu

Nous nous intéressons dans ce chapitre à Cacol, un système évitant la duplication des pages du cache dans un environnement virtualisé, sans modification de l'OS invité. La duplication des pages du cache entraîne une perte importante de mémoire, sachant que la mémoire est une ressource critique aujourd'hui dans les centres de données. Nous analysons dans un premier temps le problème de duplication des pages du cache et ses conséquences (§ 3.1), puis nous décrivons la conception et l'implémentation de la solution Cacol (§ 3.2) et enfin son évaluation (§ 3.3).

3.1 Analyse du problème

3.1.1 Exemple de la lecture d'un fichier

Nous nous intéressons ici à l'utilisation de la mémoire par le page cache, en particulier le problème de duplication des pages lié à l'environnement virtualisé des serveurs. Pour illustrer ce problème, considérons une application d'une machine virtuelle qui initie une lecture sur le disque pour obtenir les données `d` contenues dans un fichier. Les étapes suivantes sont exécutées, et résumées dans la Figure 3.1, se concentrant exclusivement sur le cas où `d` est demandée pour la première fois par l'application.

① La requête est émise par l'application et prise en compte par le système d'exploitation invité via un appel système (syscall).

② L'OS vérifie qu'il ne possède pas déjà l'information dans son page cache.

③ Dans le cas contraire, il sollicite le driver du disque et lui demande de fournir la donnée `d` qui doit être présente sur le disque.

④ Le driver du disque envoie une requête de lecture à son disque, qui est dans ce cas virtuel.

⑤ Le disque virtuel est contrôlé par le processus Qemu (à l'intérieur de l'hôte) associé à la VM. En effet, le disque dur de la VM est émulé par le processus Qemu qui va ainsi intercepter et interpréter la demande de lecture de l'OS invité et va transmettre cette I/O à l'hôte.

⑥ Comme le processus Qemu est un processus Linux comme les autres, sa demande de lecture va être exécutée de la même manière. De cette façon, l'OS hôte vérifie en premier lieu s'il ne possède pas déjà l'information dans son page cache.

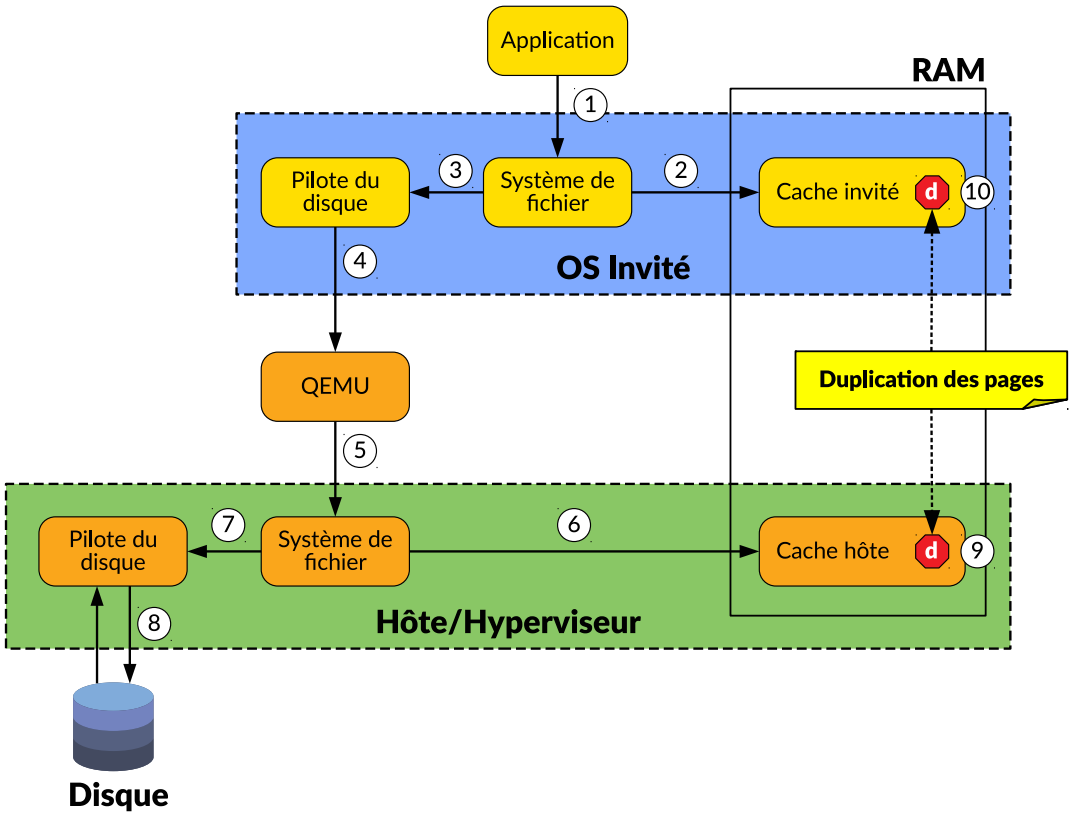


FIGURE 3.1 – **Duplication des pages dans le cache.** La donnée `d` demandée par une application dans une machine virtuelle se trouve à la fois dans le cache de l'invité et dans le cache de l'hôte.

(7) Dans le cas contraire, l'OS hôte sollicite le driver du disque et lui demande de lui fournir `d`. L'exécution est identique à l'étape (3).

(8) Le driver procède à la requête sur le disque et obtient les données `d`.

(9) A la réception des données, l'OS conserve `d` dans son page cache pour d'éventuelles futures requêtes, et les transmet à l'application Qemu qui les a demandées.

(10) Le driver de l'invité reçoit `d`, et de même que l'OS hôte, il stocke `d` dans son propre page cache et transmet finalement une copie à l'application. Le syscall de l'application est par conséquent terminé.

Nous pouvons remarquer simplement que la donnée `d` est désormais présente dans deux caches, celui de l'hôte et celui de l'invité. On parle alors du problème de duplication des pages du cache [161]. Ce dernier est source du gaspillage d'une part importante de la mémoire dans les centres de données, tout en sachant que la mémoire est un facteur limitant dans le cloud lors de la consolidation des machines virtuelles [110, 118].

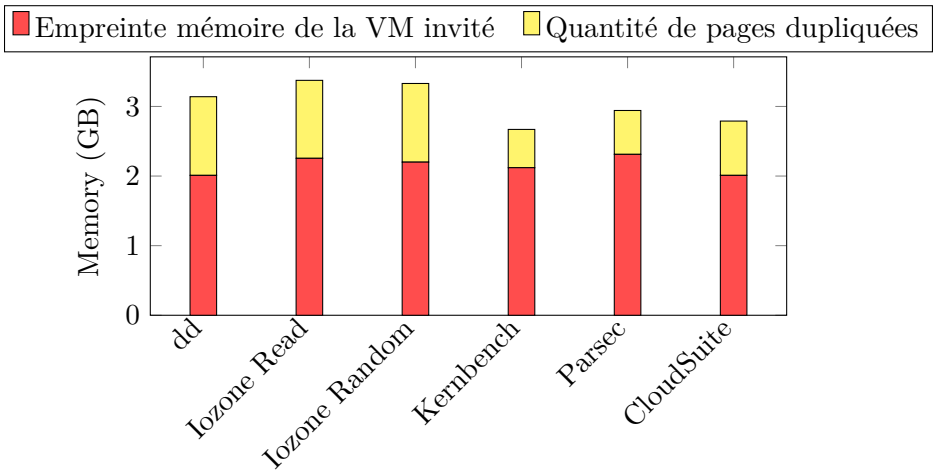
3.1.2 Impact de la duplication des pages

Afin d'évaluer l'importance de ce problème, nous avons réalisé un ensemble d'expériences pour mesurer de manière quantitative et qualitative l'impact du cache hôte, mais aussi de la duplication des pages dans ce même cache.

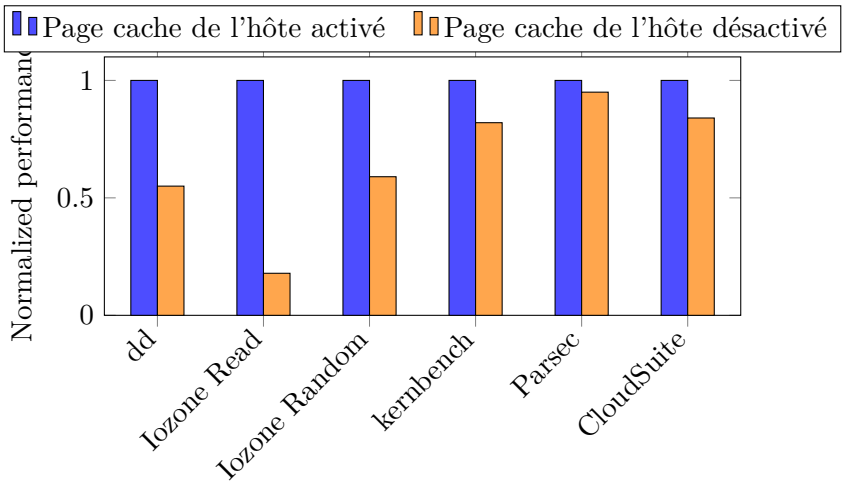
Tout d'abord, nous souhaitons estimer la quantité de mémoire gaspillée à cause de la duplication des pages. Pour cela, nous utilisons une machine virtuelle avec une mémoire de 2 Go et 4 CPU virtuels (vCPU) qui s'exécute sur une machine physique possédant 8 Go de mémoire et 8 CPU physiques. Pour chaque expérience, la VM et son application sont seules sur la machine physique. Ainsi, seules la VM et son application utilisent la mémoire de la machine physique. En réalité, l'OS utilise également un peu de mémoire pour gérer ses données internes, qui reste stable tout le long de l'expérience. Les types d'applications et benchmarks utilisés sont détaillés plus tard en Table 3.1.

La Figure 3.2a présente les résultats quantitatifs, c'est-à-dire la quantité de mémoire dupliquée dans le cache de l'hôte. Il s'agit ici de pages présentes à la fois dans le cache de l'invité et aussi dans celui de l'hôte. Nous pouvons observer que la quantité de mémoire gaspillée varie entre 840 Mo pour des opérations aléatoires, à 1680 Mo pour le benchmark `dd`. Cette dernière dépend grandement du type d'application qui s'exécute. Il est intéressant de remarquer que cette quantité de mémoire perdue est relativement élevée par rapport à la taille maximale des VM qui est de 2 Go. Cela représente jusqu'à 82% de la taille de la VM. Si nous considérons un centre de données de large envergure exécutant ce genre d'applications, les pertes de mémoire seraient énormes et entraîneraient ainsi des difficultés de consolidation des machines virtuelles, et donc des machines physiques seraient allumées inutilement.

Évidemment, cette quantité de mémoire perdue l'est dans un cas extrême, c'est-à-dire lorsqu'une seule VM s'exécute sur la machine physique et avec un type d'application spécifique, alors que le but premier de la virtualisation est de faire fonctionner plusieurs VM à la fois. Mais cela permet de mettre en évidence ce phénomène de duplication de pages. Dans une configuration plus classique avec 4 VM s'exécutant simultanément, la quantité de perte est proche de 20% de la mémoire de la VM. Néanmoins, d'un point de vue plus global, donc du point de vue de la machine hôte, il reste tout de même 65% de sa mémoire qui sont des pages dupliquées et donc de la mémoire inutile comme nous pourrons le voir plus tard dans l'évaluation en Section 3.3.3.1.



(a) Quantité de mémoire utilisée en plus (en jaune) par l'hôte à cause de la duplication des pages dans le cache.



(b) Incidence du page cache de l'hôte sur les performances d'une application s'exécutant dans une machine virtuelle.

FIGURE 3.2 – **Évaluations de la duplication des pages du cache.** Utilisation en grande quantité de la mémoire mais baisse plus importante proportionnellement des performances si le cache est désactivé.

Une solution simple pour résoudre ce problème serait de désactiver l'utilisation du cache hôte pour les VM. Malheureusement, cette solution a une incidence bien trop importante sur les performances des applications comme nous pouvons le voir en Figure 3.2b. Nous observons une dégradation des performances allant de 5% pour l'application Parsec raytrace jusqu'à 79% pour une lecture séquentielle effectuée par Iozone. Cette perte de performances provient des avantages fournis par le cache hôte, notamment le pre-fetching des pages du disque, c'est-à-dire le pré-chargement et la lecture par groupe du disque réduisant ainsi les problèmes inhérents aux disques durs (cf. § 2.4.1). Une diminution, pouvant aller jusqu'à 5 fois moins, des performances d'une application est inenvisageable dans le cloud.

L'objectif est donc de créer une solution permettant d'atténuer l'effet de la duplication des pages du cache dans l'hôte, sans être intrusif, c'est-à-dire sans modifier l'OS invité de la machine virtuelle, et sans désactiver pour autant le cache de l'hôte.

3.2 Contributions

Pour répondre au problème de la duplication des pages du cache sans modification du système d'exploitation invité, nous proposons Cacol. Cette section est organisée de la manière suivante. Tout d'abord, je vous présente une description générale et algorithmique de Cacol (§ 3.2.1), ensuite je vous décris l'implémentation dans le noyau Linux et les difficultés rencontrées (§ 3.2.2).

3.2.1 Description générale et algorithmique

3.2.1.1 Description générale

Comme il est impossible de modifier l'OS invité, ce dernier agit comme une boîte noire dont on ne connaît pas le comportement. Les points d'entrées pour notre système seront l'interception des demandes de lecture et d'écriture de la part de la VM via Qemu.

Une différence majeure entre le cache de l'hôte et celui de l'invité tient au fait que les pages ne sont pas accédées de la même manière. En effet, une page

demandée fréquemment en lecture par une application de la VM est servie de fait par le cache de l'invité. Ainsi, l'OS hôte n'en sait rien car il ne voit aucune requête de lecture lui parvenir, car déjà satisfaite par l'invité. Par conséquent, le motif d'accès aux pages du cache hôte est plus sporadique que celui de l'invité et il est donc préférable de ne pas appliquer la même politique de cache. Une politique de type LRU (*Least Recently Used*) [114] telle que c'est le cas dans Linux et dans l'hôte par défaut est inadaptée ici à un cache que l'on peut qualifier de *seconde chance*.

L'idée générale est donc de ne pas conserver dans le cache de l'hôte les pages qui sont déjà présentes dans le cache de l'invité et en particulier de conserver les pages que la VM est susceptible de demander.

Comme le but est de gérer les pages du cache de l'hôte uniquement, ma solution Cacol est donc une politique d'éviction de cache, non intrusive vis-à-vis de la VM, s'exécutant dans l'OS hôte. Du fait que la VM est perçue comme une boîte noire, cette politique ne peut reposer que sur les indications de lectures et d'écritures qui lui parviennent.

Concernant le fait de ne conserver que les pages susceptibles d'être redemandées, il est intéressant d'évaluer la probabilité qu'une page soit demandée à nouveau par l'OS invité à l'OS hôte. En effet, une page sera forcément demandée une première fois pour que la VM ait accès à son contenu, et sera alors conservée dans le cache de l'invité, puis elle pourra être évincée et redemandée à l'hôte, et cela plusieurs fois. La Figure 3.3 présente la répartition en pourcentage du nombre de fois qu'une page est demandée par l'OS invité lors de l'exécution de Kernbench, une compilation de noyau.

Nous pouvons observer d'abord, que de nombreuses pages ne sont demandées qu'une seule fois (i.e. première barre). Elles le sont soit parce qu'elles sont conservées durant toute la durée de vie de l'application dans le page cache invité, soit parce qu'elles ne sont pas réutilisées après avoir été évincées. La seconde observation est que les pages redemandées le seront en général plus d'une fois, et cela peut s'expliquer par la nature du cache hôte qui est de type seconde chance. Si une page s'est faite évincée et est redemandée, elle a une bonne chance d'être évincée à nouveau et redemandée.

Pour la solution Cacol, les pages sont conservées dans le cache de l'hôte seulement à partir de la deuxième requête. Cela permet de réduire l'empreinte mémoire sur l'hôte car comme nous pouvons le voir, 60% des pages ne seront

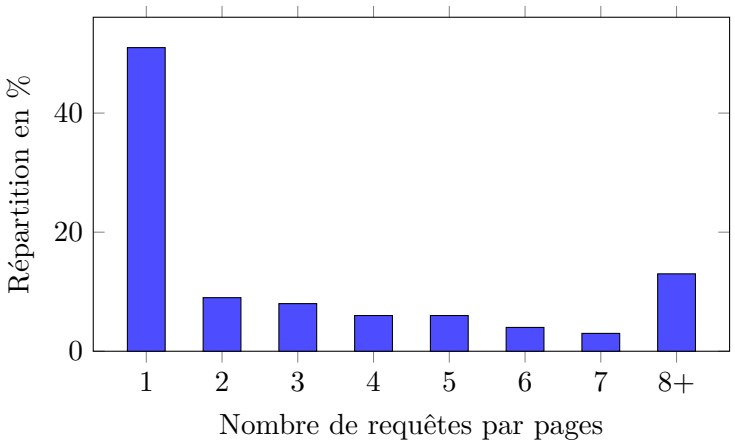


FIGURE 3.3 – **Répartition des requêtes de pages faites par la VM** sur le cache hôte lors de l'exécution de Kernbench.

pas conservées (première et deuxième barre). Évidemment, en contre-partie, les pages demandées deux fois (9%) devront être requêtées deux fois au disque, mais cela reste moindre comparé au gain de mémoire. La deuxième barre correspond ainsi à l'overhead de Cacol par rapport à une politique par défaut qui aurait trouvé la page dans le cache. Par contre, les pages redemandées fréquemment (barres 3 et plus) auront un taux de hit du cache augmenté car le cache est moins saturé qu'avec la politique par défaut.

Finalement, la solution envisagée, Cacol, est une politique d'éviction de cache au niveau de l'hôte ne conservant que les pages ayant au minimum eu deux requêtes de la part de la machine virtuelle, et le tout sans avoir à modifier l'OS invité.

3.2.1.2 Description algorithmique

L'algorithme de traitement lors d'une lecture de page par Cacol est résumé en Figure 3.4. Ainsi, lorsqu'une page est demandée par la VM dans l'hôte pour la première fois, cette dernière n'est pas conservée dans le cache de l'hôte. Si la VM demande la page une deuxième fois, elle est cette fois-ci conservée dans le cache. En effet, l'idée avancée est qu'une page demandée au moins deux fois est une page qui a peu de chance d'être gardée dans le cache de l'invité. Du point de vue de l'hôte et donc de Cacol, c'est une page intéressante à sauvegarder dans son cache car elle a de grandes chances d'être redemandée.

Le nombre de fois où l'OS invité demande une page est stocké dans une

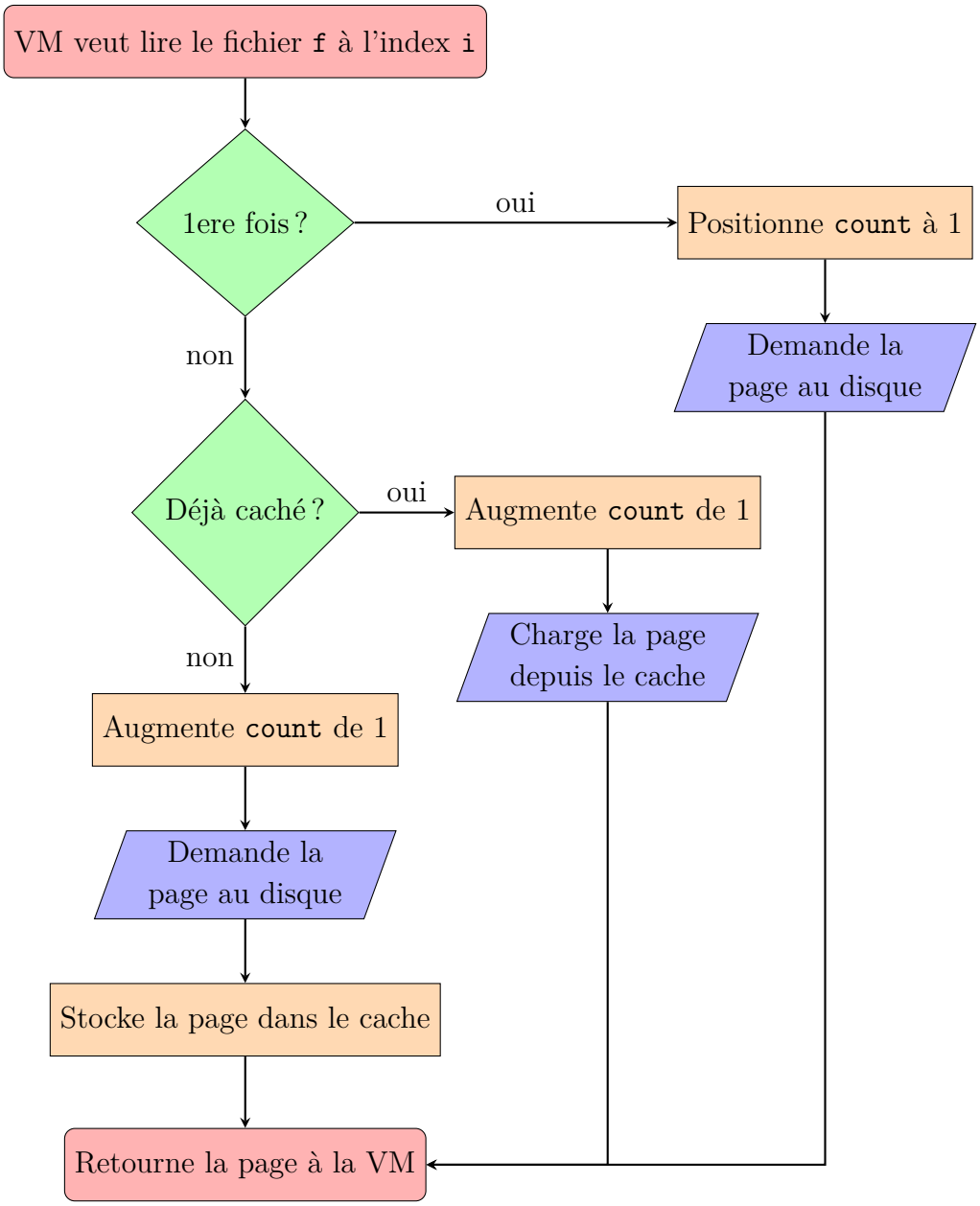


FIGURE 3.4 – **Fonctionnement simplifié de Cacol lors d'une lecture d'une page**

variable appelée `count` dans la Figure 3.4. Dès que la page est demandée une seconde fois, le `count` est incrémenté et la page reste dans le cache. Par conséquent, le cache est maintenant peuplé des pages les plus sollicitées et les plus récentes, mis à part les pages accédées une seule fois. L'algorithme est basé sur les fréquence d'accès et est similaire à une politique d'éviction LFU (*Least Frequently Used*) [138].

Un autre point important concerne le cas des pages accédées de nombreuses fois dans un court laps de temps mais plus ensuite. En effet, dans un algorithme d'éviction purement LFU, i.e. se basant seulement sur la fréquence, ce genre de page reste en haut de la liste et n'est jamais évincée. Il est nécessaire d'utiliser un algorithme mêlant fréquence et temporalité [92]. Une page qui n'est plus sollicitée par la VM, l'est pour deux raisons : soit elle est présente dans le cache de la VM car très utilisée par cette dernière, soit elle n'est plus dans le cache de la VM car elle n'en a plus besoin. Dans les deux cas, il est inutile d'utiliser de la mémoire pour la garder dans le cache hôte.

Les pages chargées en prefetch par le noyau sont initialisées avec un compteur à 0. Ainsi, lorsqu'une de ces pages est réellement accédée pour la première fois, son compteur est positionné à 1.

Enfin, il est important de garantir une certaine équité entre les VM, autrement dit une répartition *juste* des ressources entre VM, car une VM peut par exemple effectuer beaucoup d'I/O et accaparer la quasi-totalité de la mémoire disponible dans l'hôte. Dans notre cas, une répartition juste se définit comme une garantie minimale d'accès à la ressource mémoire et donc a priori des performances minimales garanties.

Exemple d'exécution de Cacol La Figure 3.5 montre l'état des caches lors de la requête d'une page par une application dans une VM et l'action de Cacol par rapport à celle-ci.

① L'application fait une lecture de la donnée `d` à son système de fichiers. La page contenant `d` est chargée dans le cache de l'OS invité depuis le disque. L'OS hôte en garde une trace, initialisant le compteur à 1, mais ne conserve pas la page dans son cache.

② Durant la vie de la machine virtuelle, la page se retrouve en fin de liste LRU de l'invité et est évincée du cache. Le cache hôte ne le sait pas.

③ Dans cet état, la page a été évincée.

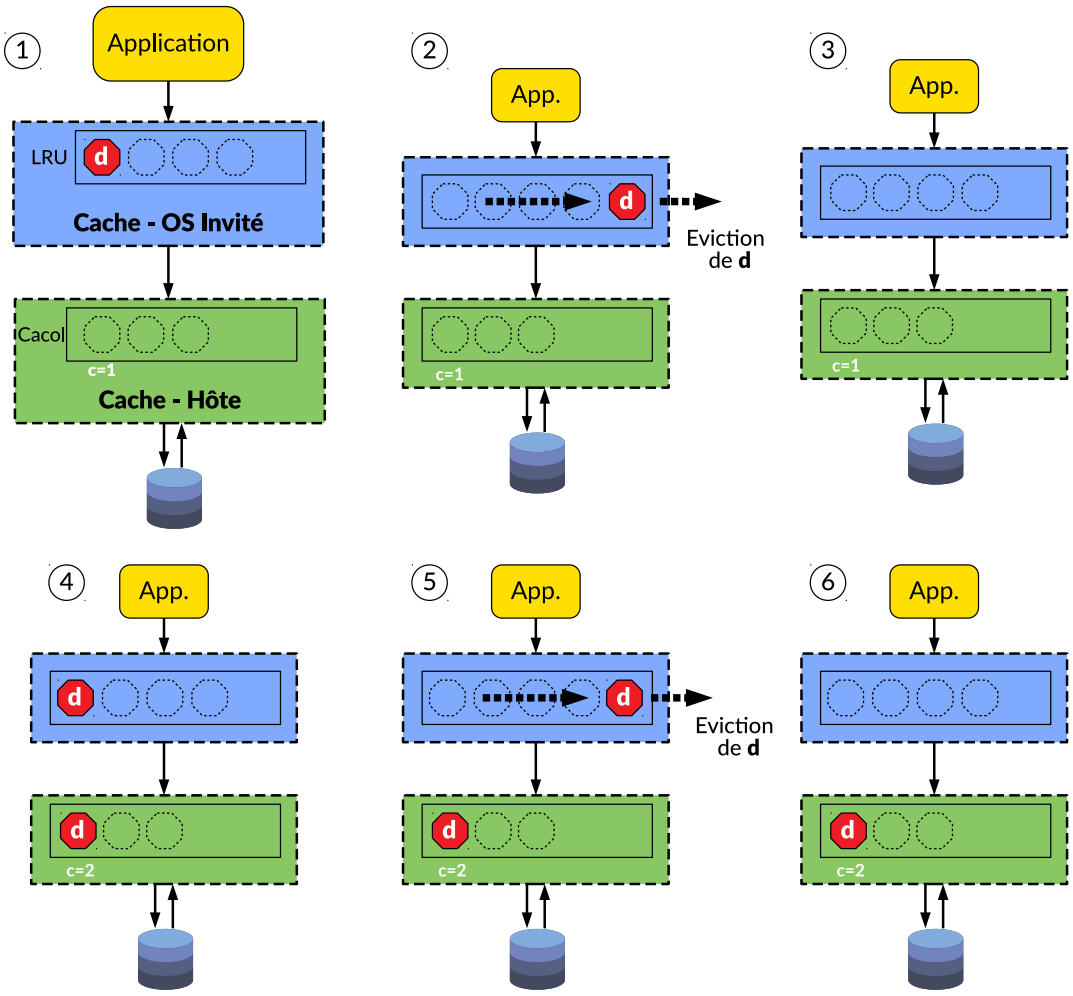


FIGURE 3.5 – **Fonctionnement de Cacol lors de la requête de pages**. La page contenant `d` n'est conservée dans le cache de l'hôte qu'à partir de la deuxième requête. Elle est d'abord servie depuis le disque, puis finalement depuis le cache hôte.

④ L'application redemande la même donnée. Cette fois-ci, la page est conservée dans les deux caches, et le cache hôte incrémente son compteur.

⑤ La page se fait de nouveau évincer du cache fichier de l'OS invité.

⑥ A la prochaine demande de la donnée `d` par l'application, cette dernière sera servie directement depuis le cache fichier de l'OS hôte.

Le cache de l'hôte est très peu modifié car peu de pages y sont conservées. Ainsi, il ne reste que des pages importantes car utiles aux machines virtuelles.

3.2.2 Implémentation

Une des principales difficultés rencontrées lors de l'implémentation dans le noyau Linux de Cacol est la modification du noyau pour incorporer une nouvelle politique d'éviction. En effet, la politique par défaut est étroitement liée à l'ensemble du système de gestion de la mémoire, que ce soit par ses structures de données et par son code d'exécution. Il n'y a pas de module d'éviction à proprement parler mais des optimisations de code. Par exemple, l'ajout dans le cache et l'éviction sont totalement décorrélés, les deux s'exécutant en parallèle (cf. § 2.4.2). Les points importants discutés ici sont les suivants :

- comment identifier une page appartenant à une VM ;
- comment tenir à jour le nombre d'accès d'une page par une VM ;
- comment définir et s'assurer d'une répartition juste du cache.

3.2.2.1 Identification des pages d'une VM

Cacol a besoin de différencier une page appartenant à une VM et une page de l'OS hôte, dans le but de leur appliquer un traitement différent sans avoir à modifier l'OS invité. Pour cela, Cacol garde une référence de l'`inode` du disque virtuel utilisé par Qemu au démarrage d'une VM. Chaque inode est la structure de représentation interne d'un fichier dans un système de fichiers. Chaque inode possède un numéro unique. Ce numéro unique est alors conservé dans une liste au démarrage de la VM. Cette liste recense donc les numéros inode des disques virtuels des VM. A chaque requête de lecture ou écriture sur un fichier faite par une application d'une VM, il nous est possible de savoir, via le numéro inode du fichier, s'il est question d'une requête d'une VM ou non, et de quelle VM il s'agit.

3.2.2.2 Compter le nombre d'accès

Il est nécessaire pour le fonctionnement de Cacol (la politique d'éviction) de compter combien de fois une même page de la VM a été accédée. Pour cela, nous utilisons une table de hachage (hash table) pour compter le nombre de références aux pages des VM. Cette table est définie dans la structure `address_space` rattachée à un fichier ouvert. Cette structure est la représentation linéaire d'un fichier ou d'un block device comme présentée en Figure 3.6.

Simplement un fichier est découpé en section de 4096 octets dont le contenu sera copié en mémoire dans une page du cache. Chaque section peut être identifiée par son offset, donc son décalage par rapport au début du fichier en octets, ou par son index.

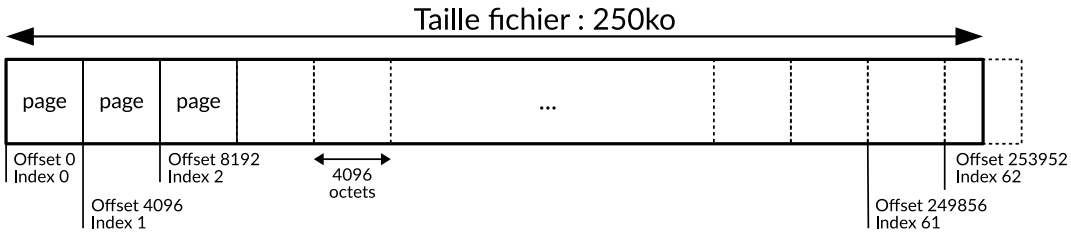


FIGURE 3.6 – **La structure `address_space`** est la représentation linéaire d'un fichier dans Linux.

Pour compter le nombre d'accès à une même portion du fichier, Cacol stocke dans le tableau le nombre d'accès en utilisant l'index de la page comme indice du tableau. La valeur est simplement incrémentée à chaque appel système concernant le fichier et la page en question.

Il est important de noter que le compte du nombre d'accès d'une page ne se fait pas directement sur la page elle-même pour des raisons techniques. En effet, les pages sont des structures de données internes du système d'exploitation. Leur contenu peut être à tout moment modifié, supprimé ou déplacé par le memory manager. On perdrait ainsi le compte associé à une page si ce dernier était stocké directement avec les méta-données de la page. Ainsi, Cacol conserve le nombre d'accès à une page dans un tableau à part.

Une autre incidence de l'architecture du système d'exploitation Linux est que la structure `address_space` est détruite dès que le fichier ou block device est fermé par le processus qui l'a ouvert. Par conséquent, le comptage du nombre d'accès des pages est réinitialisé à chaque ouverture de fichier car le tableau est lié à cette structure. Dans notre cas, ce n'est pas un problème car le processus QEMU qui gère une VM garde toujours son disque virtuel ouvert durant l'exécution de la VM.

L'algorithme de Cacol lors d'une lecture est défini en Figure 3.4. Lors d'une requête d'une page par une VM, Cacol vérifie la valeur du nombre d'accès dans le tableau à l'indice `i`. Deux possibilités ici :

- La valeur est de 0, la page est donc demandée pour la première fois. Le compte est donc mis à 1, le contenu du fichier est copié dans la page, la page est servie à celui qui l'a demandée et la page est supprimée du

cache.

- La valeur est 1 ou plus, la page a déjà été demandée dans le passé. Le compte est incrémenté, la page est servie depuis le cache si elle s'y trouve et chargée depuis le disque dans le cas contraire. La page est finalement conservée dans le cache.

Un autre problème rencontré vient du fait que, par défaut, la page fournie par le système d'exploitation pour y mettre le contenu d'un fichier est directement placée dans une liste LRU. Il s'agit d'une optimisation du système d'exploitation Linux car la politique implémentée est justement le LRU. La page étant fraîchement allouée, elle est donc la plus récente et placée en tête de liste pour pouvoir être évincée du cache en dernière. Dans notre cas, cela ne nous arrange pas du tout, car justement nous ne souhaitons pas conserver les pages fraîchement allouées comme expliqué précédemment. Pour cela, j'ai modifié le code d'ajout dans le cache pour qu'une page n'y soit pas ajoutée et il faut donc vérifier a priori s'il s'agit d'un premier accès ou non. La page n'est alors plus gérée par le memory manager mais par Cacol. Il faut ensuite rendre la page à l'allocateur, une fois que les données ont été copiées et transmises au programme.

3.2.2.3 Politique d'éviction pseudo-LFU

Comme expliqué précédemment (§ 3.2.1), Cacol est une politique d'éviction du cache au niveau de l'hôte qui est utilisée comme un cache de seconde chance. Par conséquent, nous avons choisi une politique se basant sur la fréquence d'accès (LFU) et non sur la temporalité des accès (LRU) car cela permet d'obtenir de meilleures performances [169].

On ne peut pas qualifier cette politique de purement LFU, déjà car elle ne conserve pas les pages lors de leur premier accès mais aussi de part son implémentation. En effet, l'implémentation suit grandement celle de base dans le noyau Linux et en particulier des optimisations faites.

Dans Linux, les évictions du cache se déroulent sous forme de scan, c'est-à-dire que le memory manager sélectionne et isole une partie des pages susceptibles d'être évincées du cache, les traite et, le cas échéant, les replace dans le cache si elles doivent rester dans le cache ou les récupère et les vide de leur contenu. Linux gérant une LRU, il lui est simplement nécessaire de maintenir leur ordre relatif de dernier accès. Les pages les plus anciennes se retrouveront naturellement en bas de la liste et donc seront les pages à évincer. Seulement

lorsqu'une page est accédée, elle n'est pas replacée en haut de la liste. A la place, un drapeau est mis en place pour indiquer que la page a été accédée, il est appelé `accessed flag`. Ainsi, lors du scan d'éviction, une page se trouvant en bas de la liste mais qui possède son drapeau de levé, se verra replacée en haut de la liste et sera donc considérée comme la plus récente.

Ainsi, les pages ne sont pas triées temporellement selon leur dernier accès mais selon leur date d'ajout dans la liste que ce soit la première fois ou après un scan.

C'est le même phénomène qui s'applique ici avec Cacol, les pages ne sont pas triées par ordre d'accès mais par leur date d'ajout. Par contre, la différence est que lors d'un scan, le memory manager ne vérifie pas le drapeau d'accès de la page mais le compteur d'accès lié à cette page pour déterminer si elle doit être conservée ou plutôt évincée du cache.

J'ai ainsi défini cette politique d'éviction comme pseudo-LFU, car elle se base sur la fréquence d'accès mais ce n'est pas une politique LFU [93] au sens strict du terme.

Un dernier point important à lever est le problème inhérent des politiques LFU déjà détaillé précédemment (cf. § 3.2.1), c'est-à-dire le cas des pages fortement demandées sur un court laps de temps, et qui ne le sont plus ensuite. Pour pallier simplement ce problème, lors des scans, le compte d'une page est diminué et la page est replacée dans la liste. Par conséquent, le compte va diminuer au fur et à mesure du temps et des scans, l'empêchant de rester indéfiniment dans le cache.

3.2.2.4 Équité entre VM

Pour garantir une certaine équité quant à l'utilisation du cache hôte, Cacol garantit un minimum de cache à chaque VM. L'idée est d'empêcher une VM d'utiliser et de polluer l'intégralité du cache de l'hôte. Pour cela, la gestion du cache et de la politique d'éviction sont effectuées par VM.

La mémoire est divisée en deux parties égales. La première partie du cache est disponible pour utilisation par tous les processus. La seconde partie est elle-même divisée en parties égales et distribuées équitablement à l'ensemble des VM. Par exemple, si 4 VM sont exécutées simultanément sur la même machine

physique, l'hôte garantit un minimum de 12.5% du cache à chaque VM. Il s'agit d'un taux minimum de cache de l'hôte alloué aux VM, appelé `low_watermark`.

Au démarrage d'une VM, cette dernière n'utilisera que très peu de mémoire et sera donc très certainement en-dessous de son taux minimum garanti. La partie des pages qui lui sont réservées peuvent tout de même être utilisées par les autres VM. Le memory manager garantit par contre qu'aucune de ses pages actuelles ne puissent être évincées, et que dans le cas où la VM a besoin de plus de mémoire, les pages lui seront forcément attribuées jusqu'à atteindre au moins son minimum.

Lorsque la pression mémoire est forte, le système a besoin de récupérer de la mémoire disponible et évince donc des pages du cache. Ici, il choisira parmi les VM qui n'ont pas atteint leur `low_watermark`, les autres seront exclues du mécanisme de récupération des pages. Parmi les VM éligibles, Cacol choisit de scanner en premier la VM la moins récente, c'est-à-dire celle dont la page la plus récente est la moins récente. Il scanne ensuite les autres VM chacune à leur tour.

3.2.2.5 Changement de politique durant l'exécution

Pour faciliter l'expérimentation de Cacol, j'ai décidé d'ajouter une interface dans le système de fichier `/proc` pour pouvoir changer la politique d'éviction durant l'exécution de l'hôte sans avoir à redémarrer la machine physique. Ainsi, après une première installation du noyau modifié incluant Cacol dans l'hôte, il est possible, via une simple écriture sur un fichier, de tester la politique d'éviction Cacol ou celle par défaut dans Linux.

3.3 Évaluations

Cette section présente l'évaluation de Cacol. Nous décrirons, dans un premier temps, l'environnement des expériences et la méthodologie utilisés (§ 3.3.1), puis les résultats de l'exécution d'un benchmark seul (§ 3.3.2) et enfin les résultats avec plusieurs applications simultanées (§ 3.3.3).

Workload	Description
Sequential read	Iozone est utilisé ici pour générer et mesurer une lecture séquentielle d'un fichier [3]. La taille du fichier est supérieure à la taille du cache de l'invité pour permettre un débordement et ainsi voir l'impact du cache de l'hôte.
Random operations	Iozone génère un flux mixte d'opérations de lecture et d'écriture aléatoirement [3].
CloudSuite	Le sous-benchmark utilisé de cette suite de benchmarks est le *Data caching bench*. Ce dernier consiste en un serveur Memcached simulant le comportement d'un serveur de cache de Twitter et utilisant le dataset de Twitter. La métrique qui nous intéresse ici est le débit exprimé en nombre de requêtes par seconde [59, 117].
Parsec	Parsec est une suite de benchmarks composée de programmes multi-threadés [26]. Nous avons utilisé *raytrace*, un outil temps réel de calcul de raytracing avec, en entrée la donnée *native* fournie par Parsec.
dd	La commande Unix `dd` est utilisée ici pour effectuer une copie (et donc une écriture séquentielle) d'un gros fichier généré aléatoirement dans `/dev/null`. La taille du fichier est supérieure à la taille du cache de l'invité.
gzip	La commande Unix `gzip` est utilisée pour réaliser la compression du code source de Linux dans une archive. Le code source étant supérieur en taille au cache de l'invité.
kernbench	Un noyau Linux (v4.0) est compilé avec le script `kernbench` [91]. Il compile le noyau avec la configuration *allnoconfig*. La métrique utilisée ici est le temps de compilation.
Filebench	Filebench [148] est un benchmark de système de fichiers et de stockage. La workload utilisée est le *webserver* prédéfini et est configuré pour servir 200 000 entrées.

TABLE 3.1 – **Liste des benchmarks utilisés pour évaluer Cacol**

Les objectifs de ces évaluations sont doubles :

(1) Montrer que Cacol réduit le nombre de pages dupliquées entre les deux niveaux de cache ;

(2) Montrer l'impact de la politique d'équité, qui permet un meilleur partage des ressources entre VM.

3.3.1 Environnement d'expériences et méthodologie

3.3.1.1 Environnement

Notre système d'évaluation est composé d'une machine avec 8 Go de mémoire et un processeur Intel i7-4800MQ avec 8 cœurs. Le système d'exploitation invité est un Ubuntu Server 16.04 avec un noyau Linux 4.4.0. L'hôte a comme OS un Ubuntu 16.04, Qemu 2.11.50 comme hyperviseur et un noyau Linux modifié 4.4.0 contenant le code de Cacol.

La Table 3.1 présente la liste des différents benchmarks utilisés pour l'évaluation. Nous comparons Cacol à deux solutions :

- L'implémentation par défaut du cache dans Linux, noté *Default* ;
- Une solution naïve, noté *naive*, consistant à désactiver le cache de l'hôte.

A part contre-indication, le protocole d'évaluations est le suivant. Chaque benchmark est exécuté seul sur la machine physique, c'est-à-dire qu'il n'y a qu'une seule application dans la VM et une seule VM à la fois. Cette VM est configurée avec 4 vCPU et 4 Go de mémoire. Chaque benchmark est répété 10 fois, en prenant soin de redémarrer la machine physique entre chaque exécution pour avoir le même environnement.

3.3.1.2 Métriques d'évaluation

Les métriques d'évaluation sont (1) la quantité de mémoire utilisée par chaque VM pour sa propre exécution et la quantité de mémoire dupliquée dans le cache hôte, (2) les performances des applications dans la VM, définies en fonction du benchmarks et détaillées dans la Table 3.1.

3.3.1.3 Profilage du cache

Pour nos évaluations, nous avons besoin de connaître la quantité de mémoire utilisée par la VM et plus particulièrement la quantité de mémoire dupliquée c'est-à-dire présente dans les deux caches à la fois.

Nous utilisons les outils internes fournis par Linux pour compter le nombre de pages appartenant à chaque fichier et donc à chaque VM. A chaque syscall,

les statistiques d'ajout dans le cache sont mis à jour par le noyau. De même lors des évictions, les suppressions du cache sont mises à jour. Il est donc possible de connaître à chaque instant le quantité de pages ajoutées, et de pages évincées du cache depuis le début.

Nous avons également utilisé l'utilitaire `vmtouch` [72] pour connaître exactement les pages présentes dans le cache. Il utilise le syscall `mincore` qui permet de fournir un tableau avec tous les indices des pages présentes à l'instant t dans le cache. Nous pouvons faire de même dans la VM directement et obtenir le contenu du cache. Il ne nous reste alors plus qu'à comparer les deux contenus et en déduire les doublons. Nous obtenons ainsi le nombre de pages dupliquées, et donc selon notre métrique, la taille totale de mémoire perdue à cause de la duplication.

3.3.2 Résultats avec un seul benchmark

3.3.2.1 Impact sur les performances des applications

La Figure 3.7 présente la quantité de mémoire dupliquée et donc gaspillée lors de l'exécution des benchmarks et la Figure 3.8 présente leurs performances. Plus c'est haut, meilleures sont les performances. Les barres d'erreurs représentent la déviation standard des 10 exécutions.

La première chose à noter est que dans la Figure 3.7, l'évaluation de la solution naïve n'apparaît pas. Tout simplement, comme le cache de l'hôte est désactivé dans ce cas-là, aucune mémoire ne peut être dupliquée et donc la valeur vaut tout le temps 0.

Sur la Figure 3.7, nous pouvons observer une duplication mémoire proche de 0 pour certaines charges comme `dd` et `gzip` pour Cacol. Pour les autres benchmarks, il y a toujours des duplications de pages malgré les efforts de Cacol. En effet, Cacol est une solution non intrusive vis-à-vis des VM, et s'appuie donc seulement sur les informations à sa disposition (cf. § 3.2.1). De ce fait, Cacol évite la duplication des pages accédées la première fois dans l'hôte, mais les pages accédées dans l'hôte plus d'une fois sont dupliquées si l'invité les cache. Malgré cela, nous pouvons constater que la politique par défaut peut gaspiller par exemple 87.5% de plus de mémoire que si elle était gérée par Cacol avec le benchmark Filebench.

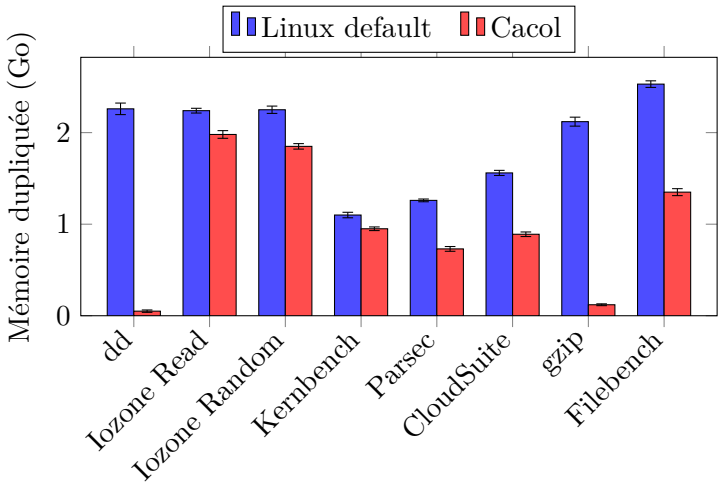


FIGURE 3.7 – **Quantité de mémoire du cache dupliquée**

L'impact sur les performances d'une application dépend fortement de la taille qu'utilise le benchmark pour s'exécuter, appelée *working set size*, par rapport à la taille du cache de l'invité dans lequel elle s'exécute. Pour illustrer ceci, considérons un benchmark dont le working set est plus grand que la taille du cache invité. Les données ne seront pas entièrement contenues dans le cache invité, et le cache hôte aura alors un impact positif sur les performances. Grâce à Cacol, les données gardées dans le cache hôte seront plus pertinentes, car elles ne seront pas déjà présentes dans le cache invité. La probabilité d'avoir un *hit* du cache de manière globale, c'est-à-dire qu'une page demandée soit déjà présente dans l'un des caches, va alors automatiquement augmenter. En partant du principe qu'un accès à la mémoire est bien plus avantageux qu'un accès disque (cf. § 2.4.1), les performances s'améliorent également. C'est le cas par exemple des benchmarks dd et Filebench. Nous observons sur la Figure 3.8, une hausse des performances jusqu'à 11%.

D'un autre côté, si le working set tient entièrement en mémoire du cache invité, il n'y aura que très peu d'impact de la part de la politique utilisée dans le cache hôte. On peut avoir autant de pages dupliquées dans l'hôte que l'on veut, les pages seront servies directement depuis l'invité. C'est ce que l'on observe sur les autres benchmarks. Cela permet de mesurer l'overhead de Cacol, ce qui est discuté juste après en Section 3.3.2.2.

Concernant l'approche naïve, c'est-à-dire désactiver totalement le cache hôte, on pourrait penser que les performances seraient similaires sur des bench-

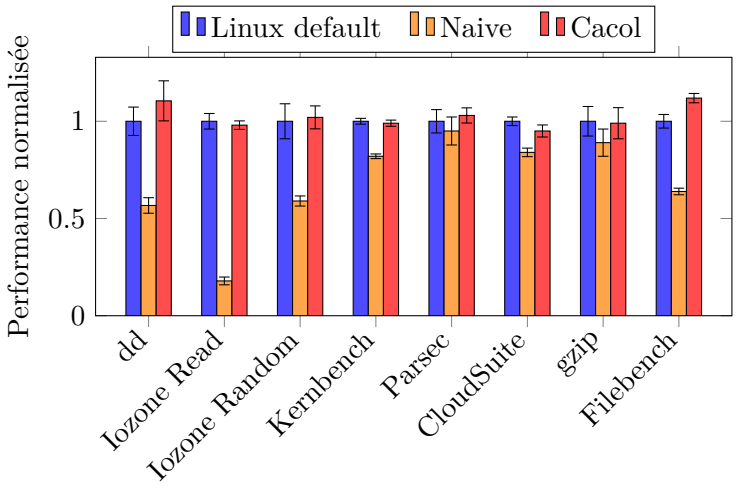


FIGURE 3.8 – **Efficacité de Cacol face au problème de duplication des pages du cache**

marks dont le working set tient en mémoire du cache invité. Ce n'est pas le cas. Dans le cas d'une lecture séquentielle, cela peut s'expliquer par la manière dont l'OS charge les données en avance en ayant repéré un motif d'utilisation d'un fichier. On parle de *prefetch*. Idem pour les autres types de benchmarks, le prefetch et la bufferisation des requêtes aux disques permettent d'augmenter drastiquement les performances. Nous pouvons observer une diminution de 45% des performances par rapport à Linux et de 50% par rapport à Cacol pour le benchmarks dd par exemple.

3.3.2.2 Overhead de Cacol

Nous pouvons parler de deux overheads différents, c'est-à-dire surcoûts de notre solution. Le premier est sur les performances des applications, et le second est sur l'empreinte mémoire de Cacol sur le système hôte.

Concernant les performances, nous avons vu, dès lors que le working set d'une application peut être entièrement contenu dans le cache de l'invité, la gestion du cache hôte devient presque inutile. Ainsi, nous pouvons observer sur la Figure 3.8 un impact de Cacol pouvant aller jusqu'à 5% sur le benchmark CloudSuite par rapport à la politique par défaut. Le surcoût lié à la solution Cacol n'est pas entièrement négligeable dans certains cas de figure.

Le surcoût mémoire lié à Cacol pour son fonctionnement provient d'une seule source. Pour chaque page de la mémoire, Cacol utilise un entier long sur

4 octets (32 bits) pour stocker la fréquence d'accès (cf. § 3.2.2.2). De plus, pour les pages qui ne sont pas dans le cache actuellement mais qui l'ont déjà été, leur compte d'accès est également comptabilisé de la même manière que les pages du cache. Les pages faisant une taille de 4096 octets, cela représente une utilisation supplémentaire de mémoire de 0.1% (4/4096) de la taille totale du disque et non de la taille de la mémoire comme c'est le cas de Linux. En utilisant un disque virtuel de 50Go par exemple, cela représente potentiellement une utilisation de 50Mo de mémoire supplémentaire. C'est une valeur haute, la réalité est souvent différente car une machine virtuelle ne va pas forcément utiliser l'intégralité du disque pendant sa durée de vie. On remarque donc que pour des disques de taille plus grande, notre solution pourrait ne pas passer à l'échelle.

Une première solution simple et non implémentée, serait de diminuer la taille sur laquelle l'information du compte est stockée, en utilisant par exemple un `short int` (2 octets) ou même un `char` (1 octet), car une page risque peu d'être accédée plus de 255 fois. Et quand bien même la page serait demandée plus de 255 fois, l'information n'est pas forcément pertinente à avoir. Cacol utilise de la mémoire pour diverses autres structures de données mais leur empreinte est anecdotique à côté.

3.3.3 Résultats avec des benchmarks colocalisés

3.3.3.1 Analyses des performances

Pour cette expérience, 4 VM s'exécutent simultanément avec une application chacune qui sont dd, Kernbench, CloudSuite et Filebench. Chaque VM possède 1.5 Go de mémoire et 1 vCPU. L'hôte possède alors un total de 2 Go de mémoire restant et 4 CPU de libre pour fonctionner. Les résultats des performances sont regroupés en Figure 3.9.

La mémoire est ici le facteur limitant et donc le point de contention de ces différentes applications gourmandes en mémoire. Nous pouvons observer que Cacol obtient de meilleures performances sur l'ensemble des applications en fonction de leur métrique respective comparé au système par défaut. Par exemple, les performances de Filebench sont améliorées de 21% avec Cacol.

Les performances de Cacol sont meilleures lorsque les applications sont lan-

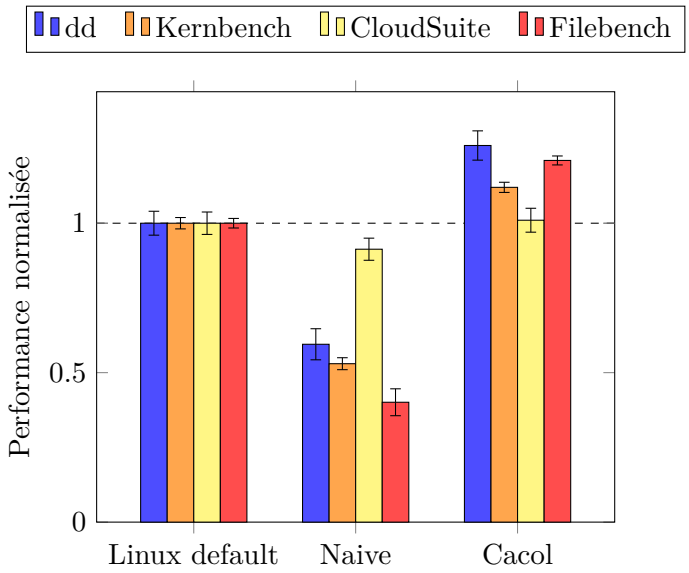


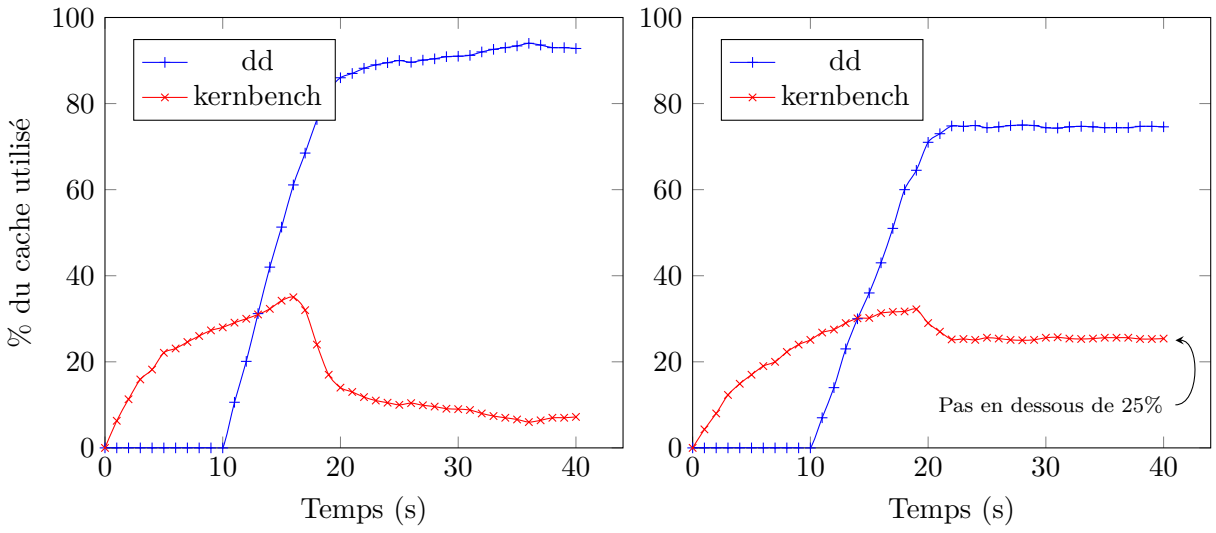
FIGURE 3.9 – **Performances de Cacol pour 4 VM s'exécutant simultanément** sur la même machine physique

cées ensemble comme le montre la Figure 3.9 par rapport à la Figure 3.8. La suppression des pages dupliquées du cache libère de la mémoire qui est utilisable directement par les autres applications colocalisées sur la machine physique. De plus, les VM ont accès à moins de mémoire du cache de l'hôte car elles doivent le partager entre elles. Ainsi, Cacol bénéficie grandement d'une pression mémoire plus grande car la politique LRU de Linux va évincer très souvent les pages et entraîner une trop grande rotation de son cache. Les pages utiles n'auront pas le temps de rester assez longtemps dans le cache pour être redemandées et ainsi faire un cache hit.

Concernant la solution naïve, ses différences de performances comparées aux deux autres solutions proviennent du *readahead*. En effet, le cache hôte étant désactivé, ce dernier ne peut pas pré-charger des pages du disque et ainsi profiter de la proximité spatiale des pages qui seront demandées par la VM. De plus, les pages seront souvent demandées une par une par la VM, supprimant les avantages de la bufferisation.

3.3.3.2 Équité entre VM

Les workloads, avec une activité I/O intensive, peuvent affecter significativement l'état du cache hôte. En effet, une demande excessive de pages va entraîner une pression mémoire et donc l'éviction des pages des autres VM.



(a) Linux default (b) Cacol

FIGURE 3.10 – **Utilisation du cache hôte avec 2 VM s'exécutant sur le même hôte**. Dans l'exemple Cacol, comme il y a deux VM en parallèle, j'ai défini le `low_watermak` à 25% (cf. § 3.2.2.4). Le memory manager garantit alors à chaque VM l'utilisation d'au moins 25% de la mémoire de l'hôte, pour garantir de l'équité.

L'hôte sera submergé par ces requêtes et les demandes des autres VM se termineront alors par une lecture sur le disque et donc un *cache miss*.

Pour mettre en lumière ce phénomène, l'expérience consiste à lancer deux VM simultanément, une à forte empreinte mémoire (dd) et l'autre réalisant une activité normale, ici de la compilation du noyau (Kernbench). Les deux VM possèdent 2 Go de mémoire et 2 vCPU chacune. La Figure 3.10 montre comment le cache est utilisé lors de l'exécution. La Table 3.2 rassemble les résultats quantitatifs.

La Figure 3.10a montre bien que le benchmark dd utilise plus le cache hôte que l'autre benchmark. Ici, il s'agit de la configuration par défaut de Linux, et comme prévu le cache est submergé par dd, qui prend jusqu'à 94% du cache hôte disponible. Les performances de la VM colocalisée avec lui en sont automatiquement réduites.

Pour contrer ce phénomène, Cacol garantit un minimum d'utilisation de cache pour chaque VM comme cela est détaillé en Section 3.2.2.4. La Figure 3.10b montre cela, car il est possible d'observer la `low_watermark`, c'est-à-dire la barre en-dessous de laquelle la mémoire de la VM Kernbench ne peut

	Workload `dd` (Vitesse de lecture)	Kernbench (Temps de compilation)
Linux	46.9 MB/s	153,6 s (std dev 2.27)
Cacol	46.2 MB/s	142,7 s (std dev 2.85)

TABLE 3.2 – **Performance de deux applications s'exécutant simultanément.** Cacol réduit l'agressivité du workload `dd` pour booster les performances de Kernbench.

plus se faire réclamer ses pages. Cette barre est atteinte à 22 secondes, et à partir de ce moment-là, ce sont les pages de l'autre VM, celle de dd, à qui le système réclame les pages.

Les performances des deux benchmarks exécutés en simultané sont détaillées en Table 3.2. En particulier, le temps d'exécution de Kernbench est réduit et ses performances sont donc améliorées. L'amélioration est de 7.6%. A contrario, la VM avec le workload dd voit sa bande passante diminuée de 1.5% car il a tout simplement moins de mémoire utilisable sur le cache de l'hôte. L'agressivité, vis-à-vis de l'utilisation du cache hôte, de cette dernière VM est limitée et permet à l'autre VM de réduire cette pénalité. In fine, on constate une amélioration globale des performances grâce à un partage plus équitable et plus juste de la ressource mémoire.

3.4 Synthèse

Dans ce chapitre, nous avons vu Cacol, un système s'exécutant dans le cache hôte, dont l'objectif est d'atténuer la duplication des pages du cache fichiers dans un environnement virtualisé.

Dans un premier temps, nous avons mis en lumière le problème de duplication des pages du cache de fichiers (§ 3.1) à travers un exemple et la décomposition de la lecture d'un fichier dans un environnement virtualisé. Aussi, nous avons pu évaluer l'impact de cette duplication lors de l'exécution de différents benchmarks, montrant une perte de mémoire pouvant aller jusqu'à 82% de la taille de la VM. Une solution naïve pour empêcher la duplication est de désactiver l'utilisation du cache hôte pour les VM, mais cela occasionne des baisses de performances trop drastiques.

Pour résoudre ce problème, nous proposons Cacol, une politique d'éviction

du cache de l'hôte (§ 3.2). Le cache de l'hôte est un cache de seconde chance et a donc besoin d'une politique adaptée. Nous avons observé la répartition des requêtes de lecture sur un tel type de cache. Ainsi, Cacol ne conserve les pages que lorsqu'elles sont accédées au moins deux fois par la machine virtuelle, réduisant l'empreinte mémoire de la VM sur le cache de l'hôte. Cette politique, Cacol, est implémentée dans le noyau Linux.

Enfin, nous avons montré son utilisabilité et évalué son efficacité (§ 3.3). Cacol permet de réduire dans le meilleur des cas de 87.5% la quantité de cache de l'hôte utilisée par les VM lorsque l'on exécute un benchmark seul. Les performances des applications sont également améliorées jusqu'à 12%. Cacol tire son épingle du jeu lors de fortes pressions sur la mémoire, montré par l'exécution de plusieurs VM simultanément. Dans ce cas, la réduction de la duplication des pages va engendrer une meilleure utilisation du cache de l'hôte et améliorer les performances jusqu'à 21% pour des workloads générant de nombreuses I/O. Finalement, nous évaluons la capacité de Cacol d'assurer une équité entre VM quant à l'utilisation du cache de l'hôte. Nous proposons de garantir une part minimale du cache hôte pour chaque VM définie en fonction du nombre de VM s'exécutant simultanément.

Chapitre 4

Infinicache

Contenu

Nous avons pu voir au Chapitre 3, la première contribution Cacol qui consistait en une politique de cache pour une meilleure utilisation de ce dernier. Notre objectif est toujours le même, c'est-à-dire améliorer les performances des applications s'exécutant dans une machine virtuelle, en optimisant et améliorant l'utilisation du page cache. Ici, nous allons nous placer du point de vue de l'ensemble du centre de données et non plus d'une seule machine physique. L'idée principale présentée dans ce chapitre est d'exploiter la mémoire non utilisée des autres machines du centre de données via un réseau à très haute vitesse pour agrandir le page cache. Dans un premier temps, nous présentons le contexte de ces travaux et analysons le problème (§ 4.1), puis la Section 4.2 détaille la solution apportée, appelée Infinicache, et enfin nous présentons son évaluation (§ 4.3).

4.1 Analyse du problème

4.1.1 Cas d'utilisation de la mémoire dans un datacenter

Nous nous plaçons ici dans le contexte d'un centre de données typique contenant plusieurs milliers de machines physiques [52] où des dizaines de milliers de machines virtuelles fonctionnent. Le principal problème de ce genre de centre est la consolidation des machines virtuelles sur le plus petit nombre d'hôtes physiques, les machines inutilisées étant mises en veille. Ainsi, cela permet de réduire la consommation globale du data-center. En effet, comme nous pouvons le voir sur la Figure 4.1, une machine physique ayant peu d'activité consomme relativement la même quantité d'énergie qu'une machine utilisant 100% de ses ressources. Une machine éteinte, quant à elle, ne consommera rien. Ainsi, il est bien plus avantageux d'avoir une machine utilisant 100% de ses ressources avec une machine éteinte, que deux machines utilisant chacune 50% de leurs ressources.

Partant de ce constat, il est évident que le but est de faire fonctionner le moins de machines physiques possibles à l'échelle du data-center. Malheureusement, le placement des machines virtuelles dépend de plusieurs de leurs paramètres, entre autre la mémoire réservée, la quantité de processeur utilisée, etc.. Pour illustrer le problème du placement de VM, la Figure 4.2 présente le placement de plusieurs machines virtuelles sur trois machines physiques. Dans les 3 cas de figures, les quantités de ressources utilisées sont les mêmes. Il est

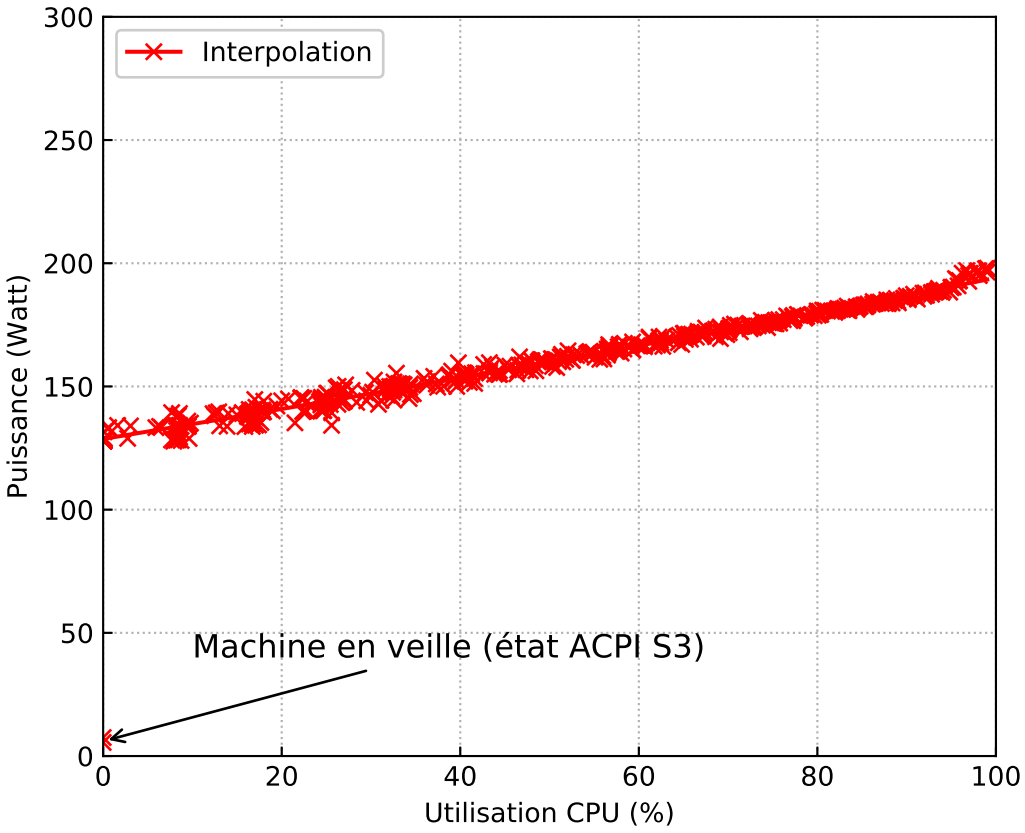


FIGURE 4.1 – **Puissance consommée en fonction de l'utilisation CPU d'un serveur.** Les mesures ont étaient effectuées sur un serveur PowerEdge R430 (28 cœurs E5-2650Lv4@1.70GHz, 64Go de RAM). La puissance est obtenue avec le PowerSpy v2 de Alciom [4]. La charge CPU est générée grâce à la commande `stress-ng`. Le serveur a une fréquence CPU fixe.

intéressant de remarquer que l'état (2) est plus intéressant du point de vue énergétique que les deux autres états. Mais cela n'est pas toujours possible. Nous pouvons constater que la transition de l'état (1) à l'état (2) est possible mais pas de l'état (3) vers l'état (1).

Il est surtout important de remarquer que des ressources sont inutilisées dans tous les cas, et qu'il est impossible d'utiliser au maximum toute la mémoire d'une machine virtuelle. En effet, la mémoire d'une machine virtuelle est définie comme de la mémoire réservée et non de la mémoire réellement utilisée. La mémoire réservée est la mémoire définie à l'allocation comme la taille maximale que la VM peut utiliser lors de son exécution. La mémoire réellement utilisée est simplement la quantité effective de mémoire utilisée par la VM et varie au cours de l'exécution. En particulier, l'hyperviseur ne peut pas allouer plus de mémoire qu'il n'en possède et est obligé de se restreindre par rapport à

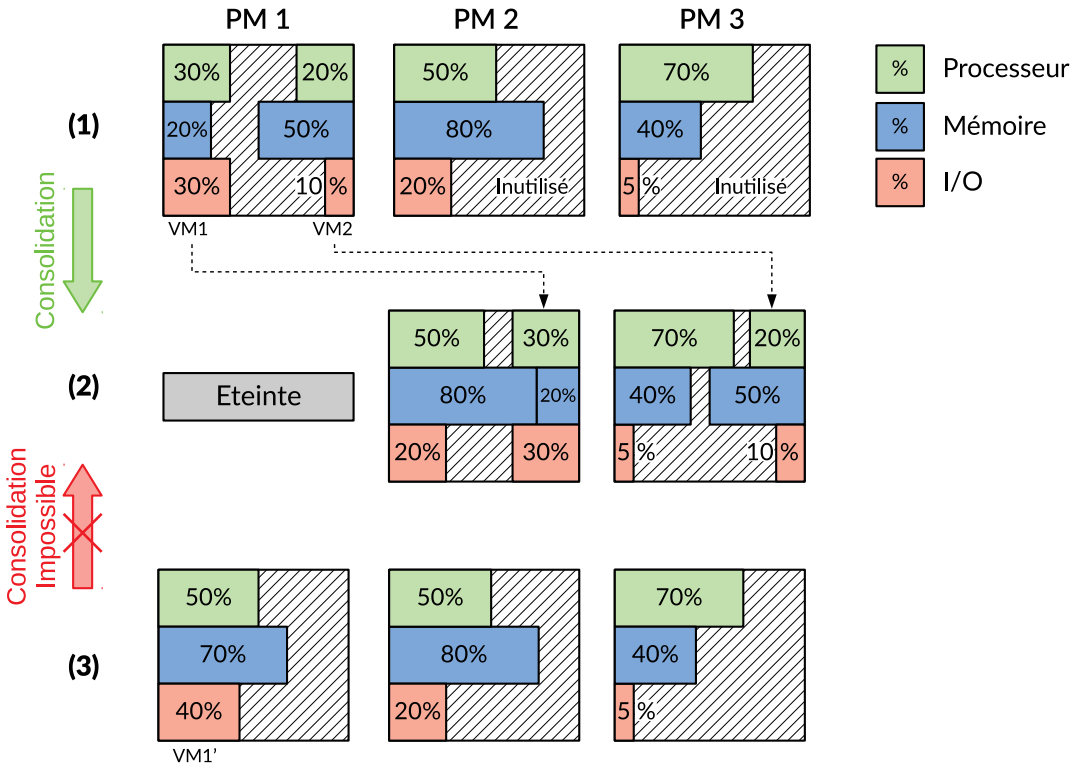


FIGURE 4.2 – **Consolidation de machines virtuelles.** La transition (1) → (2) est possible car les VM1 et VM2 sont assez petites pour pouvoir être transférées vers les autres machines physiques, PM2 et PM3. A contrario, VM1' est trop grosse pour pouvoir être placée entièrement sur une autre machine physique.

la taille de la mémoire réservée par les machines virtuelles. Ainsi, si les VM n'utilisent pas toute la mémoire, il y a tout simplement de la perte de mémoire que l'hyperviseur ne peut pas allouer à d'autres machines virtuelles.

Il existe diverses techniques pour exploiter au mieux cette mémoire non utilisée. Une d'entre elles est le ballooning [155, 136], qui consiste à reprendre la mémoire déjà réservée par les VM mais qui est non utilisée et la placer dans un ballon. Ce ballon est donc une réserve de mémoire et permet d'allouer de la mémoire pour accueillir des VM supplémentaires. La taille du ballon peut diminuer en cas de forte pression sur la mémoire. Lorsque la pression est terminée, le ballon reprend de la mémoire et regonfle. Il est même possible de gonfler fortement le ballon, de sorte que non seulement la mémoire non utilisée soit récupérée, mais les pages allouées mais faiblement utiliséees soient vidées dans le swap. Cette technique permet d'accueillir plus de VM sur un serveur physique, c'est-à-dire améliorer son taux de consolidation, mais en contrepartie peut, dans le cas de fortes pressions sur la mémoire, réduire les performances

de ces mêmes VM.

Notre idée est d'exploiter la mémoire non utilisée d'une machine physique, pour étendre le cache d'une autre machine. Ainsi, lorsqu'une machine possède de la mémoire en trop, et qu'aucune machine virtuelle de cette machine physique ne l'utilise, elle peut l'allouer et la mettre à disposition des autres machines physiques à distance. Contrairement aux techniques de consolidation, nous nous intéressons seulement au cache de fichiers. Ainsi pour récupérer de la mémoire, les pages n'ont pas besoin d'être copiées sur un support swap, car les fichiers sont déjà présents sur un support de stockage, généralement un disque.

Il est indispensable, pour que cette solution soit efficace, que la mémoire à distance soit plus rapide d'accès que de la mémoire sur disque local. Pour cela, nous allons utiliser des cartes réseaux RDMA qui permettent d'accéder à de la mémoire à distance avec des temps de latence plus faibles que de la mémoire sur disque.

4.1.2 Infiniband

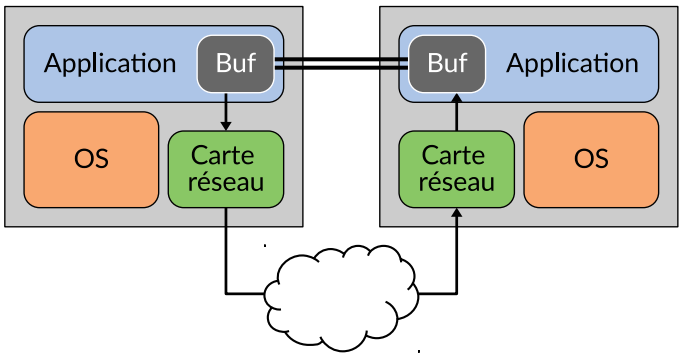


FIGURE 4.3 – **Communication Infiniband**. Une application peut directement utiliser de la mémoire distante sans l'intervention de l'OS.

Infiniband est une technologie permettant d'avoir un accès direct à de la mémoire à distance d'une autre machine sans l'intervention du processeur distant, grâce à des cartes spécifiques, nommées HCA (*Host Channel Adapter*). Ces cartes sont utilisées de manière native via l'interface VERBS et garantissent des temps de latence très faibles. Des requêtes DMA (*Direct Memory Access*) à distance, appelées *RDMA*, peuvent être lancées par des applications aussi bien du noyau que de l'utilisateur. Leur principale caractéristique est la possibilité d'écrire dans une mémoire à distance et donc de communiquer directement avec

une autre application à distance sans avoir besoin de l'aide du système d'exploitation, seulement du driver de la carte réseau pour transmettre les messages. La communication directe est illustrée en Figure 4.3, RDMA est un protocole s'exécutant ainsi directement à la couche transport [123].

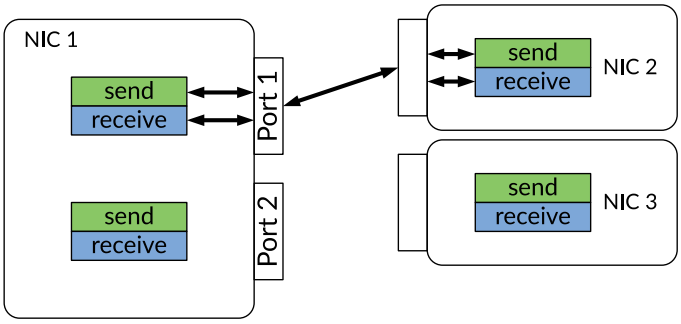


FIGURE 4.4 – **Communication entre Queue Pairs**. Les applications envoient tous leurs messages via des files d'émission et de réception regroupées en *Queue Pair*. Chaque port possède une queue pair, composée des deux files send et receive.

La communication se fait à travers des files appelées *work queues*, comme on peut le voir en Figure 4.4. Il en existe 3 différentes, *send*, *receive* et la queue de *completion*. Les files de réception et d'envoi sont toujours liées et sont aussi appelées *Queue pair* (QP). Chaque QP est associée à un port de la carte Infiniband.

Il est intéressant de voir sur la Figure 4.5 que les messages, sous forme de *Work request* (WR), sont placés par l'application dans l'une des deux files de la queue pair à travers l'interface verbs. Ces requêtes sont alors transformées en *Work Request Element* (WQE). Ensuite, le matériel (HCA), c'est-à-dire la carte réseau Infiniband ici, traite ces WQE et crée une *Work completion* (WC) qu'il place dans la *Work completion queue* (CQ). Ainsi, chaque Queue pair, et donc chaque port de la carte, possède une completion queue. L'application peut alors lire les WC et obtenir le résultat de ses requêtes. Ce comportement est différent des communications classiques où une interruption est générée à chaque réception de paquet.

Nous pouvons différencier deux types de requêtes associées à l'interface verbs, unilatérale et bilatérale. La première est une opération unilatérale (*one-sided*), c'est-à-dire que l'écriture et la lecture de la mémoire à distance se font sans l'intervention de l'autre machine et en particulier de son CPU. Les opérations en question ont les codes `IB_WR_RDMA_WRITE` ou `IB_WR_RDMA_READ` ajoutés dans la WR. Le deuxième type de requête est une opération bilatérale

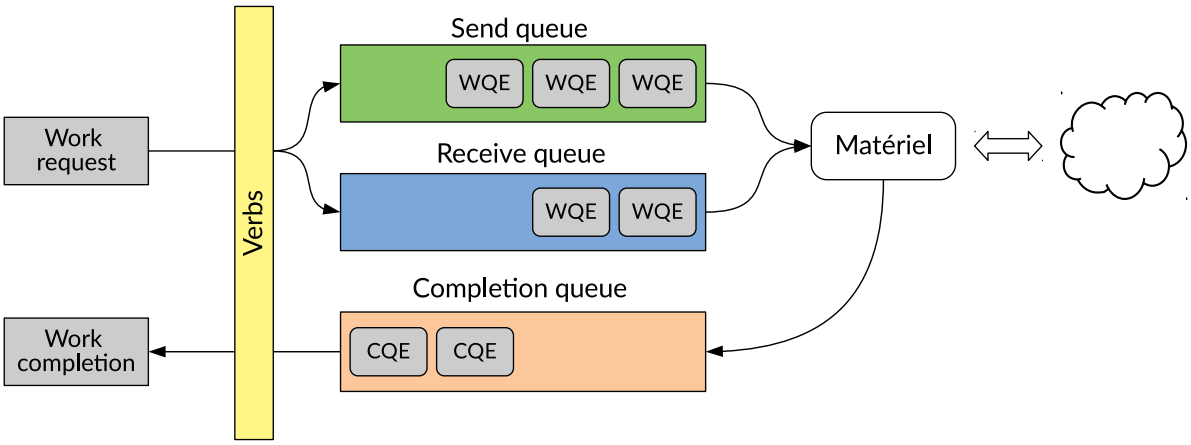


FIGURE 4.5 – **Fonctionnement interne d'une carte Infiniband**. Une application utilise l'API Verbs pour pouvoir utiliser une carte Infiniband et donc communiquer. Une requête de travail (WR) est transformée de manière interne en élément (WQE) et placée sur une des deux files (WQ) selon le type de demande. Le matériel exécute ces éléments et place un élément de complétion (CQE) dans la file de complétion (CQ) après avoir effectivement accompli le travail demandé.

(*two-sided*), où les deux parties interviennent. Dans ce cas-là, l'application crée une work request avec le code `IB_WR_SEND`, et envoie le message qu'il désire. La machine cible doit, quant à elle, créer une WR de réception avec le code `IB_WR_RECV` et allouer un buffer de réception. La WR est placée dans la file de réception. A la réception du message, une work completion est placée dans la CQ, le buffer contient le message et peut être traité par l'application cible.

La mémoire fournie par une machine est mise à disposition à travers des régions de mémoire, préalablement enregistrées par le matériel, appelées *memory regions (MR)*. L'enregistrement de la mémoire par la carte RDMA est faite en demandant au système d'exploitation le mapping physique de la région mémoire et aussi en fixant la mémoire, c'est-à-dire l'empêchant d'être évincée de la mémoire centrale et par conséquent d'être envoyée sur un support de swap. L'enregistrement crée également deux clés appelées L_key et R_key. La première est la clé locale et permet aux applications en local, c'est-à-dire sur la machine physique, d'accéder à la memory region. La seconde est la clé à distance (*Remote key*) et peut être envoyée aux autres machines pour qu'elles puissent avoir accès à la MR en question, via des requêtes RDMA de lecture et d'écriture. Enfin, une MR fait partie d'un *Protection Domain* (PD). Ce PD rassemble les MR et les QP entre elles et peut être vu comme une entité d'agrégation. Les memory regions et les queue pairs restent néanmoins des entités

Abr.	Nom	Description
PD	Protection Domain	Ensemble regroupant les queue pairs et les memory regions
MR	Memory Region	Zone mémoire enregistrée sur laquelle le matériel peut directement lire et écrire. Possède des R_Key et L_Key (clés à distance et locales)
QP	Queue Pair	File pour l'envoi et la réception de requêtes de travail (WR). Il y a donc la *send queue* et la *receive queue*.
CQ	Completion Queue	Les complétions des work requests, appelés Work Completion, sont placés dans cette file. Cette file est associée aux QP.
WR	Work Request	Requêtes pouvant être utilisées pour l'envoi ou la réception de données. Décrit l'action à effectuer et est placée dans la Queue pair associée (la send ou receive). Possède une référence vers un SGE.
WQE	Work Queue Element	Représentation interne des Work request lorsqu'elles sont placées dans les QP.
SGE	Scatter/Gather Element	Définit une adresse mémoire pour lire ou écrire. Doit avoir la L_Key ou la R_Key de la memory region pour pouvoir s'authentifier.
WC	Work Completion	Après avoir traité une WR, une work completion est créée pour contenir le résultat de la requête. Elle est placée dans la Completion queue.

TABLE 4.1 – **Abréviations du lexique Infiniband**

totalement distinctes, et leur seule caractéristique commune est d'appartenir au même domaine.

Toutes ces abréviations relatives à Infiniband et RDMA sont rassemblées et compilées dans la Table 4.1.

Ainsi, comme nous l'avons vu, l'utilisation des cartes réseau Infiniband se fait à l'aide de l'interface verbs, souvent appelée *ib_verbs* pour InfiniBand verbs [22]. Il existe de nombreuses fonctions dans cette interface, comme par exemple `ibv_reg_mr` pour enregistrer une memory region. Pour communiquer, une application utilise principalement deux verbs, `ib_post_send` pour faire une requête d'envoi de message et `ib_post_recv` pour une demande de réception.

L'exécution pas à pas de la communication entres deux applications, à l'aide d'opérations bilatérales, peut se décrire comme suit :

— Une application veut envoyer un message à une application à distance. Elle alloue un buffer de mémoire DMA de taille souhaitée.
— L'application crée un objet work request et l'initialise. En particulier, elle lie le buffer à la WR, et fournit l'adresse et le port de destination.
— L'application pose la WR dans la file d'envoi, via l'interface verbs, en appelant la fonction `ib_post_send`. L'application peut désormais soit attendre la complétion de sa requête ou simplement faire autre chose.
— Le matériel traite la requête (WQE).
— Lorsque le matériel reçoit l'acquittement de la bonne réception du message par le matériel destinataire, il dépose alors une completion queue element (CQE) dans la file de completion (CQ).
— L'application peut finalement lire la work completion, lui garantissant la bonne exécution ou non de sa requête.

Pour une communication à l'aide d'opérations unilatérales, l'application a besoin de la R_key de la zone mémoire à laquelle elle souhaite accéder. Elle crée de la même manière une WR et la dépose dans la file d'envoi. Lorsque la machine cible reçoit ce type de paquet, c'est le matériel de la carte réseau qui va directement traiter la demande et lire ou écrire la mémoire locale sans l'intervention du processeur, et l'émetteur recevra une work completion.

Un point important à noter à propos du design d'Infiniband est que la communication se fait de manière asynchrone. Une fois une requête déposée sur une queue de la QP, l'application n'est pas bloquée et peut exécuter autre chose. Elle devra ensuite soit lire en continu la CQ et attendre qu'une work completion y soit placée, soit demander qu'une interruption la réveille lorsqu'un nouveau WQE est placé dans la CQ.

4.2 Infinicache

4.2.1 Description générale

Nous supposons ici que la mémoire inutilisée par des VM a déjà pu être réutilisée par d'autres VM sur la même machine et qu'il existe des disparités d'utilisation de la mémoire entre les machines.

L'idée principale d'Infinicache est d'exploiter la mémoire physique à distance d'une autre machine pour étendre le cache des applications s'exécutant sur

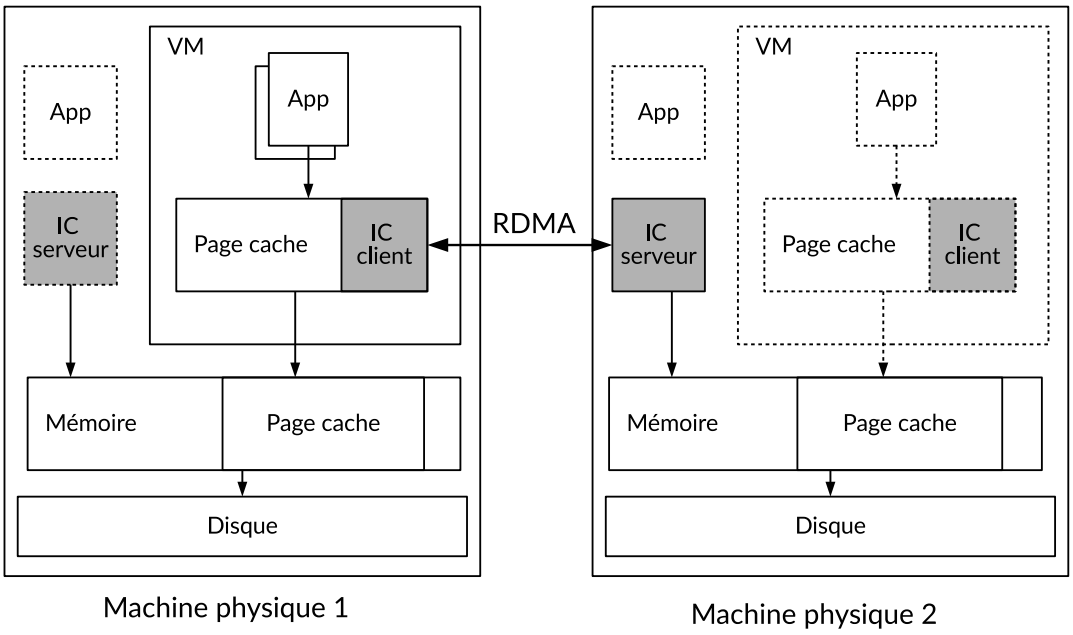


FIGURE 4.6 – **Architecture d'Infinicache**. Elle est composée de deux composants, le client directement implémenté dans le page cache de la machine virtuelle, permet d'étendre ce dernier, et le serveur, une application utilisateur, fournit de la mémoire aux machines qui en ont besoin.

la machine locale. En effet, comme nous avons pu le voir en Section 4.1.1, la mémoire n'est pas toujours utilisée entièrement sur une machine physique, ainsi elle pourrait servir pour d'autres machines. Une machine met alors à disposition des autres machines du centre de données son surplus de mémoire auquel il est possible d'accéder via un réseau à haute vitesse en utilisant le protocole RDMA. Cette mémoire supplémentaire, que l'on appellera *Cluster memory*, va permettre d'étendre le page cache d'une machine virtuelle qui en a besoin. Les OS des machines virtuelles sont modifiés pour permettre l'utilisation de la Cluster memory, mais dans une moindre mesure, seulement une cinquantaine de lignes de code dans le noyau et un module à charger.

La Figure 4.6 présente le positionnement des deux composants d'Infinicache. Le premier, Infinicache client, s'exécute dans l'espace noyau de la machine virtuelle, et permet de demander des pages supplémentaires de mémoire à d'autres machines physiques. Il gère ensuite le placement des pages dans la mémoire distante. Le second, Infinicache serveur, s'exécute en espace utilisateur de l'hôte, fournit et alloue de la mémoire que la machine possède en trop pour pouvoir être exploitée par les autres machines du réseau.

Pour le composant principal, c'est-à-dire le client, directement implémenté

Algorithm 1 Fonctionnement d'Infinicache lors de la requête d'une page

```
1:  function GET_PAGE(offset)
2:      page ← null
3:      if page ∈ cache then
4:          page ← get_from_cache
5:      else if page ∈ distant_cache then
6:          page ← get_from_distant_cache
7:      end if
8:      if !page then
9:          page ← alloc_and_get_from_disk
10:     end if
11: return page
12: end function
```

dans le page cache, son objectif est d'augmenter la taille du page cache en requêtant de la mémoire à distance à des machines qui en fournissent. Une fois la mémoire obtenue, il devient le responsable de la gestion des pages à distance, que ce soit pour l'envoi ou pour la récupération des données. Son algorithme de fonctionnement peut être décrit de manière simple à l'Algorithme 1.

Le noyau doit charger la page depuis le disque si cette dernière ne se trouve pas dans le cache, c'est-à-dire si la fonction a renvoyé un pointeur `null` de page.

Le serveur, quant à lui, est un composant relativement simple. Son but est de vérifier la quantité de mémoire actuellement utilisée, d'allouer de la mémoire libre et de communiquer au client l'adresse de cette mémoire. Lors de pressions mémoire, le serveur peut également récupérer la mémoire allouée pour la rendre au noyau de sa machine physique.

4.2.2 Protocole de communication

Deux implémentations différentes ont été faites. Une première, en utilisant les verbs `send` et `receive` et une seconde en utilisant les verbs `read` et `write`. Les deux étant fondamentalement différentes dans leur fonctionnement. En particulier en send/receive, la machine mettant à disposition ses pages doit traiter les messages et donc être active et utiliser ses processeurs. Un diagramme de séquence représentant les deux modes de fonctionnement est présenté en Figure 4.7.

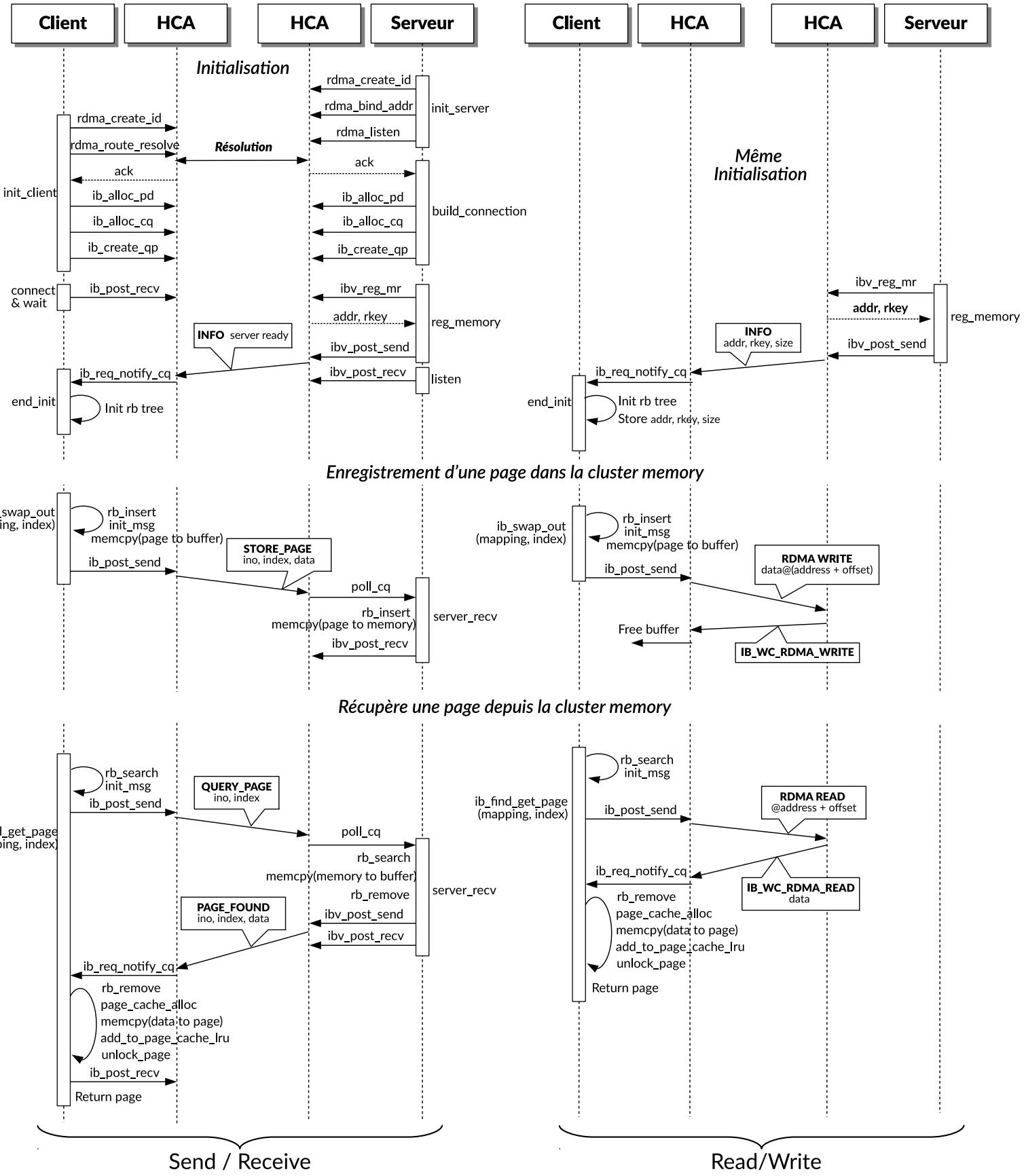


FIGURE 4.7 – **Diagramme de séquence de la communication d'Infinicache**. En particulier, on remarque que le machine serveur n'utilise aucune ressource processeur en mode read/write, à part pour l'initialisation.

Dans un premier temps, nous faisons une initialisation globale des structures Infiniband. Le serveur alloue les outils qui lui permet d'écouter les demandes de connexion. Il est nécessaire, par exemple de créer un id auprès de la carte RDMA (`rdma_create_id`), qui permet d'identifier l'application auprès de la carte. Le client crée alors une connexion avec le serveur et les deux cartes réseaux peuvent commencer à communiquer. Ensuite, client et serveur vont initialiser tous les objets liés à Infiniband, comme le protection domain, la completion queue, et les queue pairs.

Une fois la connexion établie, le serveur peut allouer dynamiquement de la mémoire si la machine physique en a de disponible et notifie le client à travers un message d'information (INFO).

4.2.2.1 Mode send receive

Les verbs utilisés dans ce mode sont les work request avec le code `IB_WR_SEND` et `IB_WR_RECV`. Le serveur est actif et attend des messages dans des buffers de réception qu'il poste avec le verbs `ib_post_recv`.

Récupération d'une page La Figure 4.8 présente les étapes et les structures de données intervenant lors de la récupération d'une page à distance (`ib_find_get_page`) en mode send/receive. Le récupération de la page se déroule comme suit et correspond à la récupération d'une page depuis la cluster memory sur la Figure 4.7 :

① L'application veut lire le contenu de la page verte. Le page cache ne la trouve pas dans sa propre mémoire car la référence de la page n'est pas présente dans le cache local en vérifiant la structure `mapping`.

② Le client Infinicache cherche alors la présence de la page dans la cluster memory en regardant si elle est présente dans sa structure de données locale (`rb_search`). Nous utilisons pour Infinicache un red-black tree [69].

③ Le client initialise (`init_msg`) et envoie (`ib_post_send`) une requête au serveur (en jaune) avec les informations de la page recherchée à savoir le numéro d'inode (ino) et l'index.

④ Le serveur reçoit la demande du client (en jaune) en vérifiant sa completion queue (`poll_cq`).

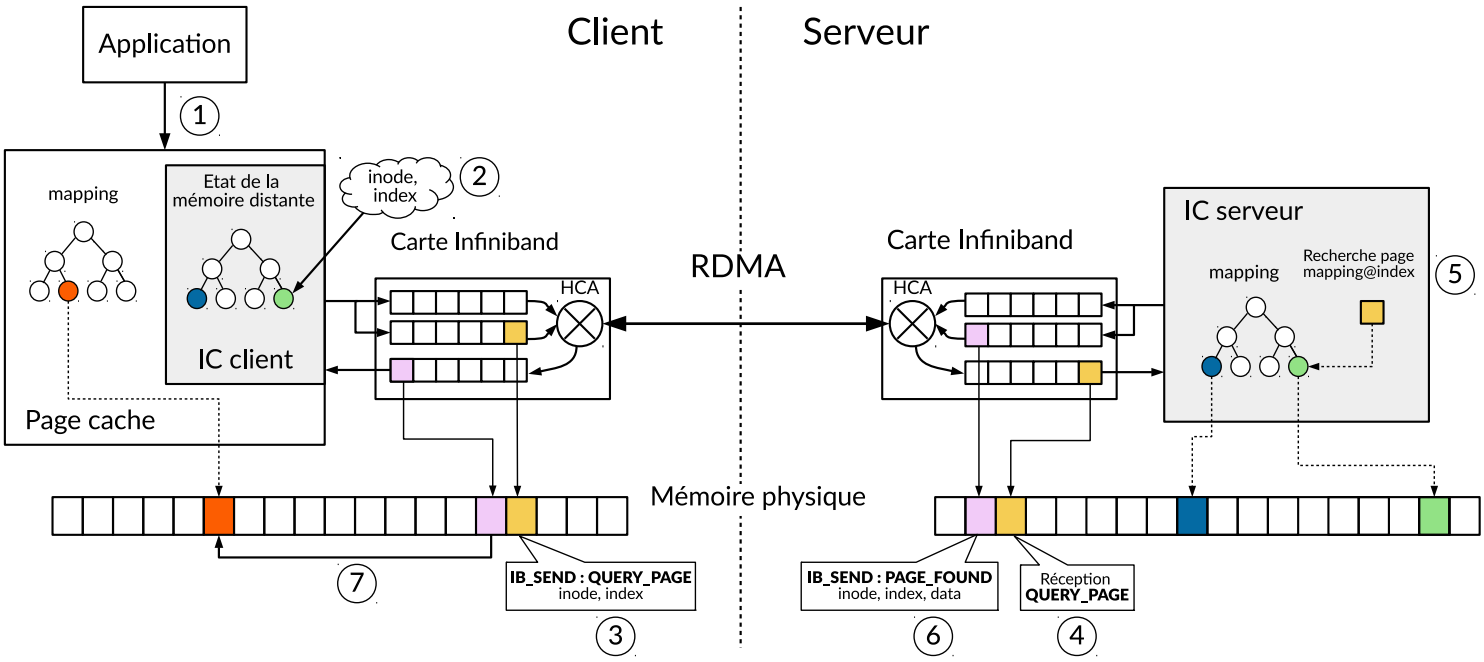


FIGURE 4.8 – **Infinicache en mode send/receive**. Le client conserve l'index des pages à distance. Ici il récupère la page verte présente dans la cluster memory, à l'aide d'une requête QUERY_PAGE.

(5) Le serveur cherche dans son arbre local la présence de la page (`rb_search`) et la récupère. Il supprime alors la référence de la page de son arbre (`rb_remove`) car cette dernière sera présente dans le cache du client après envoi.

(6) Le serveur copie (`memcpy`) le contenu de la page dans un buffer (en rose) et envoie le message (`ib_post_send`). Il initialise par ailleurs un nouveau buffer de réception (`ib_post_recv`).

(7) Le client reçoit la notification (`ib_req_notify_cq`) que la page a été retrouvée sur la mémoire distante. Il va ensuite allouer une page du cache (`page_cache_alloc`) (en rouge), recopier le contenu dans la nouvelle page (`memcpy`), ajouter la page dans la liste LRU du cache (`add_to_page_cache_lru`), et rendre la page disponible à l'application (`unlock_page`).

Eviction d'une page Lorsque le client veut évincer une page du cache et la stocker sur la cluster memory (`ib_sawp_out`), il conserve dans un premier temps sa référence dans la structure de données (`rb_insert`). Ensuite, nous initialisons un buffer d'envoi (`init_msg`) et le contenu de la page est copié dans ce buffer (`memcpy`). Il ne reste plus qu'à envoyer le message (`ib_post_send`) contenant l'identification de la page (numéro d'inode et index) et son contenu

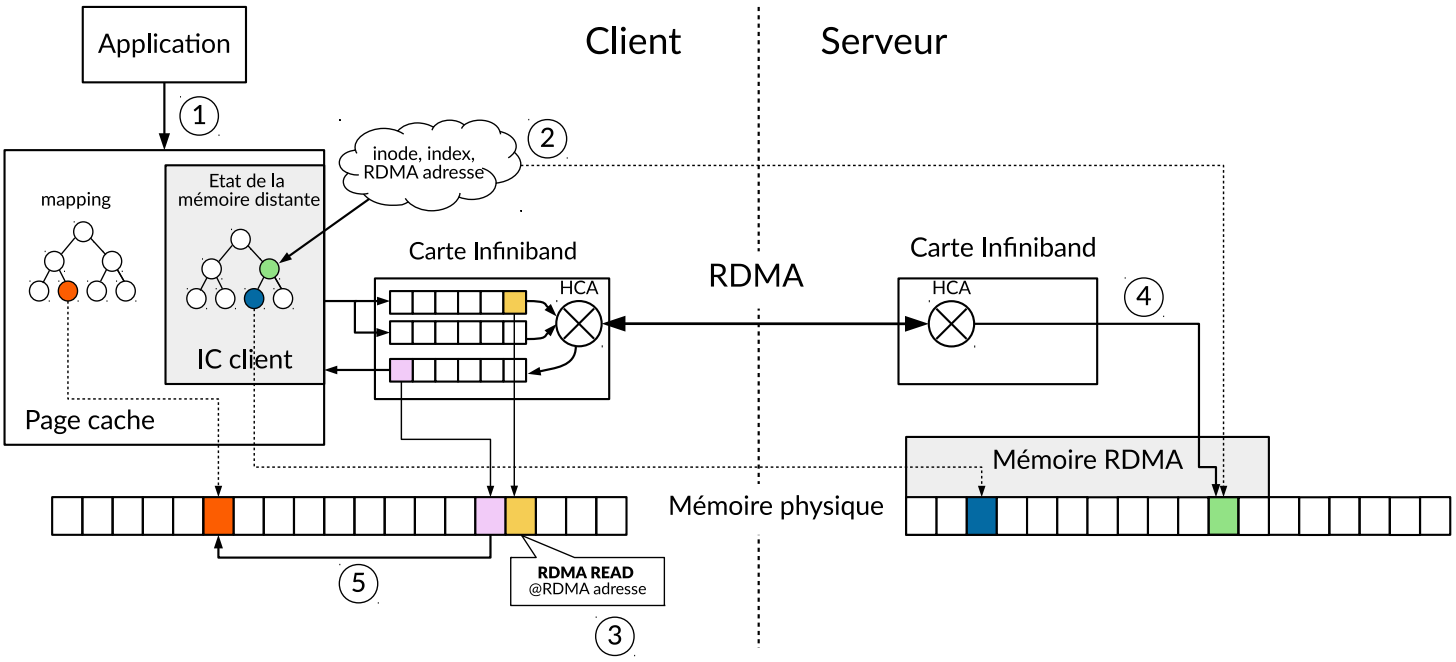


FIGURE 4.9 – **Infinicache en mode read/write**. Dans ce mode-ci, le client gère le placement des pages sur la mémoire à distance. Le serveur initialise seulement la mémoire RDMA et n'intervient plus après.

(data). Enfin, le client peut finir d'évincer la page du cache sans avoir à attendre la réponse du serveur.

4.2.2.2 Mode RDMA read et write

La logique d'initialisation est similaire au mode send/receive de communication. La seule différence est qu'à la fin de l'initialisation, le serveur enregistre la mémoire auprès de la carte réseau comme une mémoire qui sera désormais RDMA, via l'enregistrement d'une memory region (`ib_reg_mr`). Cette dernière lui fournit alors une clé distante `rkey` et une adresse `addr`. Ces deux informations vont être transmises au client, qui lui permettront d'accéder directement à la mémoire à distance. Nous pouvons voir sur la Figure 4.9 le fonctionnement de la récupération d'une page dans ce mode.

① L'application veut lire le contenu de la page verte. Le page cache ne la trouve pas dans sa propre mémoire car la référence de la page n'est pas présente dans le cache local en vérifiant la structure `mapping`.

② Le client Infinicache cherche alors la présence de la page dans la cluster memory en regardant si elle est présente dans sa structure de données

locale (`rb_search`). Le client récupère ainsi la machine sur laquelle est présente la page et l'adresse dans cette mémoire à distance.

③ Le client initialise un buffer de lecture RDMA (`init_msg`) avec l'adresse de la page à lire (en jaune).

④ La requête est traitée directement par le carte Infiniband du serveur et renvoie la section de la mémoire demandée (la page verte).

⑤ Le client reçoit la complétion de sa requête RDMA (`ib_req_notify_cq`) (en rose) et peut ensuite mettre à jour sa structure de données (`rb_remove`), allouer une page du cache (`page_cache_alloc`) (en rouge), recopier le contenu dans la nouvelle page (`memcpy`), ajouter la page dans la liste LRU du cache (`add_to_page_cache_lru`), et rendre la page disponible à l'application (`unlock_page`).

La différence majeure de ce mode est la non-intervention du processeur de la machine distante. Les requêtes sont directement traitées par la carte Infiniband qui lit ou modifie sa mémoire locale.

4.2.3 Hook noyau

Comme vu en Section 4.2.1 et particulièrement sur la Figure 4.6, le composant Infinicache principal existe à l'intérieur du page cache de la VM. Pour cela, il est nécessaire de modifier le code du système d'exploitation pour pouvoir exécuter le code d'Infinicache. Par soucis de développement et déploiement, nous avons tout d'abord créé un hook kernel, c'est-à-dire un moyen d'exécuter du code défini dans un module noyau compilé à part. Ce code est alors exécuté directement dans le noyau grâce à une recherche dans la table des méthodes du noyau `kallsyms` [1] [1].

Les deux méthodes principales définies par Infinicache, et qui sont visibles en Figure 4.7, sont `ib_swap_out` pour l'envoi d'une page sur la cluster memory, et `ib_find_get_page` pour récupérer une page depuis la cluster memory.

Le code noyau est donné à la Figure 4.10. La recherche dans la table des méthodes s'effectue à deux endroits différents, à la ligne 20 et à la ligne 47. La signature de la méthode recherchée est fournie en amont, ligne 10 et ligne 31. Ainsi, il est possible de modifier les deux fonctions principales d'Infinicache

1. Diminutif de Kernel All Symbols

```
--- filemap.c        2019-07-03 10:04:54.241923408 +0200
+++ linux-5.1/mm/filemap.c        2019-07-03 16:41:21.507544952 +0200
@@ -113,6 +113,10 @@
+volatile int hook_infinicache = 0;
+EXPORT_SYMBOL(hook_infinicache);
@@ -227,9 +231,24 @@
        void __delete_from_page_cache(struct page *page, void *shadow)
        {
                struct address_space *mapping = page->mapping;
+        typedef void (*pFunc)(struct page *page, struct address_space *mapping);
+        static pFunc ib_swap_out;
                trace_mm_filemap_delete_from_page_cache(page);

+        if (!hook_infinicache) {
+                ib_swap_out = NULL;
+                goto next;
+        }
+
+        if (!ib_swap_out)
+                ib_swap_out = (pFunc)kallsyms_lookup_name("ib_swap_out");
+        ib_swap_out(page, mapping);
+next:
                unaccount_page_cache_page(mapping, page);
                page_cache_delete(mapping, page, shadow);
        }
@@ -1601,14 +1621,51 @@
        struct page *pagecache_get_page(struct address_space *mapping, pgoff_t offset,
                int fgp_flags, gfp_t gfp_mask)
        {
                struct page *page;
+        typedef struct page* (*pFunc)(struct address_space *mapping, pgoff_t offset);
+        pFunc ib_find_get_page = NULL;

        repeat:
                page = find_get_entry(mapping, offset);
                if (xa_is_value(page))
                        page = NULL;
-        if (!page)
-                goto no_page;
+        if (!page) {
+                if (!hook_infinicache) {
+                        ib_find_get_page = NULL;
+                        goto no_page;
+                }
+
+                if (!ib_find_get_page)
+                        ib_find_get_page = (pFunc)kallsyms_lookup_name("ib_find_get_page");
+
+        page = ib_find_get_page(mapping, offset);
+        if (!page) {
+                        goto no_page;
+        }

                if (fgp_flags & FGP_LOCK) {
                        if (fgp_flags & FGP_NOWAIT) {
```

FIGURE 4.10 – **Patch du kernel Linux 5.1**. Modification du fichier `mm/filemap.c` et permettant d'appeler une fonction externe via la table kallsyms si la variable `hook_infinicache` vaut 1.

en une simple compilation et chargement de module, sans réinstallation ni redémarrage de la machine. Enfin, le dernier avantage de cette manière de faire est qu'il est possible de désactiver Infinicache à chaud simplement en déchargeant le module.

Il est important de savoir où se placer pour exécuter le code d'Infinicache au bon endroit. Pour cela, il y a deux moments importants qui correspondent aux deux méthodes d'Infinicache : lorsqu'une page est sortie du cache et lorsque l'on cherche une page dans le cache. Il faut également prendre en compte seulement les pages de fichiers et non pas les pages anonymes. Nous avons identifié les méthodes `__delete_from_page_cache` et `pagecache_get_page` comme les méthodes du noyau répondant à ces critères.

Ainsi, la fonction `ib_swap_out` (ligne 21) est appelée lorsqu'une page est sur le point d'être supprimée du cache, et la fonction `ib_find_get_page` (ligne 49) après que la page ait été recherchée dans le cache local, c'est-à-dire en regardant sa présence dans la structure `mapping`, comme on peut le voir à la ligne 35.

L'implémentation réelle est alors laissée au module. A noter également l'importance de la signature des fonctions. En effet, pour la recherche d'une page, ses seules caractéristiques disponibles sont le numéro d'inode du fichier et son offset/index dans ce fichier. Elles servent donc de clé pour le stockage à distance des pages.

4.3 Evaluations

Pour évaluer notre système Infinicache, nous avons procédé à plusieurs évaluations. Dans un premier temps, nous décrivons l'environnement de test et les métriques employées (§ 4.3.1). Nous utilisons d'abord des micro-benchmarks pour évaluer le comportement du système pour le stockage et la récupération d'une seule page de données (§ 4.3.2). Ensuite, nous évaluons Infinicache à l'aide de macro-benchmarks, c'est-à-dire en utilisant Infinicache avec 10 à 100 pages de données pour voir le passage à l'échelle (§ 4.3.3). Enfin, nous évaluons en exécutant en situation réelle avec des benchmarks plus intensifs (§ 4.3.4).

4.3.1 Environnement et métriques

4.3.1.1 Environnement d'évaluation

Pour évaluer le système Infinicache, nous avons besoin de deux machines. La première qui a besoin de mémoire sera notée dans la suite le client, et la seconde qui fournit de la mémoire sera notre serveur. Les deux machines possèdent chacune une carte réseau Mellanox ConnectX-2 [150] et sont connectées via des câbles avec un bus Infiniband QDR 4X à un routeur Infiniband.

Les machines sont identiques et sont équipées de 4 processeurs Intel i7-3770@3.40GHz et 2 threads par cœur. Chaque machine possède 8 Go de mémoire vive. Le système d'exploitation de l'hôte et des machines virtuelles est un CentOS 7.7 64 bit, avec un noyau Linux 5.1.0, modifié contenant le code de notre solution. Enfin, la version de Qemu est la 2.11.50. Le choix de CentOS a été fait car ce dernier possède des drivers compatibles avec les cartes réseaux ConnectX-2.

Le module client d'Infinicache s'exécute dans une machine virtuelle de la première machine physique, alors que le serveur s'exécute sur l'hyperviseur de la seconde machine.

4.3.1.2 Méthodologie

Chaque évaluation exécute un benchmark seul. La machine virtuelle est limitée en taille à sa création avec une mémoire de 1.3 Go. Cette taille a été choisie empiriquement pour garantir une taille du cache de fichiers d'environ 1 Go. En effet, le système d'exploitation ainsi que d'autres modules de la machine virtuelle consomme une partie de la mémoire disponible de la VM. L'utilisation du cache de l'hôte est désactivé pour la VM, mais son impact est étudié en Section 4.3.4.3.

La taille des benchmarks varie selon les évaluations et nous avons défini trois configurations de taille. La première configuration correspond à un benchmark de 1 Go en garantissant que toutes les données puissent tenir en mémoire dans le cache de la VM. Pour les deux autres configurations, nous avons augmenté la taille des benchmarks pour que leurs données représentent une empreinte mémoire de 1.5 Go et 2 Go, ce qui dépasse la taille du page cache de la VM.

La taille fournie par le serveur Infinicache est fixe et de 1 Go, pour nos tests d'évaluation. Cela permet d'évaluer l'impact d'Infinicache comme d'une extension, en taille, du page cache.

4.3.1.3 Métriques

Il est intéressant de comparer deux groupes de métriques différentes. Le premier groupe est relatif au taux d'utilisation du cache distant, c'est-à-dire de notre solution Infinicache, et des disques. Ainsi, les métriques employées ici sont :

— Le nombre de cache hit, le nombre de fois qu'une page a été servie par le cache local ;

— Le nombre d'écritures de page sur le cache distant ;

— Le nombre de cache hit à distance, donc le nombre de fois que la page a été servie depuis le cache distant ;

— Le nombre de lectures sur le disque, les pages qui ne sont servies par aucun des deux caches.

— L'occupation du cache, c'est-à-dire la quantité de pages utilisées et leur répartition entre cache local et cache distant.

Il sera également intéressant de comparer leur pourcentage relatif entre eux, la répartition en pourcentage entre cache hit local, distant et les demandes de lectures du disque.

Le deuxième groupe de métriques utilisées est celui des performances nominales des applications, dans notre cas nous utilisons le débit de l'application. En effet, les benchmarks s'exécutent avec des tailles différentes, prendre le temps d'exécution total ne permettrait pas les comparaisons.

4.3.1.4 Monitoring du cache

Pour obtenir les métriques liées au cache, nous avons utilisé SystemTap [79], un outil conçu pour analyser le noyau Linux à l'exécution. Son utilisation s'appuie sur la table des symboles de debug du noyau, et peut, via des sondes (*probe*), intercepter n'importe quel ligne de code du noyau. Pour générer les informations de debugage du noyau, il suffit de compiler ce dernier avec l'option de configuration `CONFIG_DEBUG_INFO` activé.

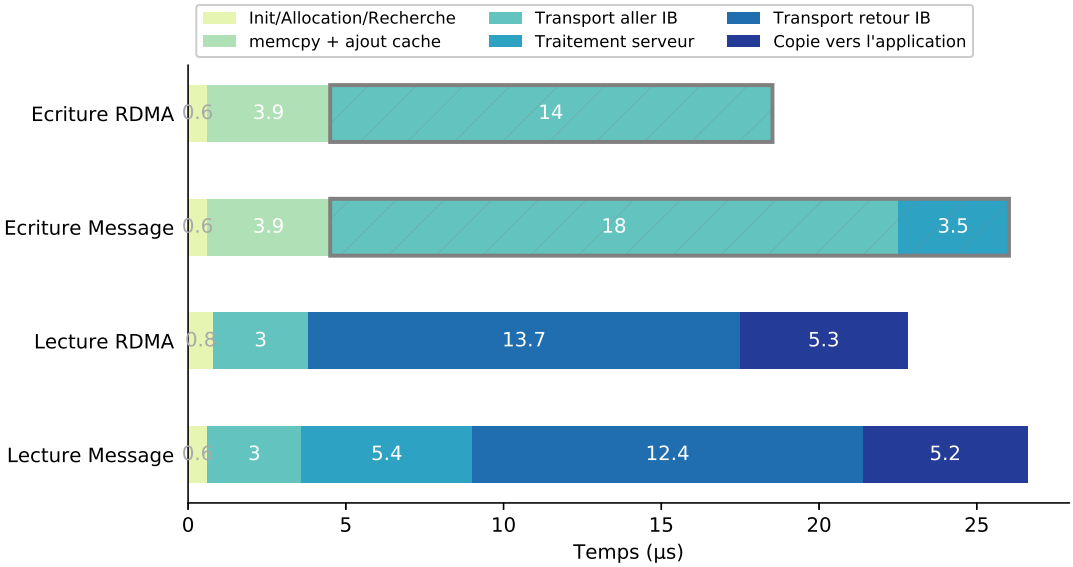


FIGURE 4.11 – **Détail du temps des fonctions d'Infinicache** lors de l'envoi et la réception d'une page de 4096 octets. La partie encadrée en gris correspond à une exécution asynchrone qui ne pénalise donc pas l'application.

4.3.2 Micro Benchmarks

Notre objectif ici est de comprendre le comportement du système lors de l'envoi et de la réception d'une seule et unique page de 4096 octets à la fois. On peut ainsi décomposer le temps passé dans chaque partie du code, et comprendre les points forts et faibles de chaque méthode de communication. Sur la Figure 4.11, nous pouvons comparer le détail des temps d'exécution pour la lecture et l'écriture RDMA (verbs read et write) et la lecture et écriture par messages (verbs send et receive).

Il faut noter en premier lieu que l'écriture de la page à distance se fait de manière asynchrone. En effet, il n'y a aucune attente une fois la requête créée et configurée, et que le contenu de la page ait été copié dans le buffer prévu pour l'envoi. La zone encadrée en gris dans le détail de l'envoi d'une page représente l'envoi asynchrone de la page par le matériel. L'application n'a pas besoin d'attendre l'envoi et le traitement de sa requête par le serveur, que ce soit réalisé par le matériel (verbs write) ou une application (verbs send). Le temps effectif pour le stockage d'une page à distance est ainsi en moyenne de 4.5μs au total pour un envoi (allocation et copie)

La chose importante à analyser ici est la lecture d'une page, car elle est réalisée lors d'un cache hit à distance. L'objectif est donc d'être plus rapide

qu'un accès disque sinon il n'y a aucun intérêt à notre système. C'est bien le cas avec une lecture RDMA qui a une durée totale de 22.8 μs et la lecture par message une durée de 26.6 μs.

Dans la suite des évaluations, nous allons utiliser exclusivement le protocole read/write, car il comporte le principal avantage de ne pas utiliser les processeurs de la machine serveur. Cela permet de n'avoir ainsi aucun impact, en terme de performances, sur la machine serveur.

4.3.3 Passage à l'échelle

Après avoir vu comment une page pouvait effectivement être copiée à distance via RDMA, nous voyons ici comment les requêtes sont enchaînées à la suite. L'objectif est donc d'observer le comportement réel et les problèmes liés à l'allocation de la mémoire, la bufferisation au niveau de la carte, et l'envoi de pages multiples.

4.3.3.1 Réutilisation des buffers

Pour fonctionner, RDMA a besoin de buffers alloués en zone mémoire DMA, et accessoirement fixés en mémoire. Cela veut dire qu'à chaque fois que l'on veut envoyer une page ou en recevoir une, il est nécessaire d'allouer un buffer de 4096 octets, en zone DMA, puis de le libérer.

Il n'est pas possible d'utiliser un seul buffer pour l'envoi de pages car cela contraindrait les envois à être sérialisés (pas en parralèle). Pour passer à l'échelle en utilisant plusieurs buffers d'envoi, nous pouvons adopter deux stratégies :

- Une allocation dynamique des buffers à la volée, c'est-à-dire à chaque fois qu'une page est envoyée en mémoire distante ;
- Utiliser un ensemble (*pool*) de buffers de mémoire pré-alloués. A chaque éviction d'une page du cache local, il suffit de récupérer un pointeur d'un buffer du pool, copier le contenu de la page dedans, l'envoyer, et remettre le buffer dans le pool une fois la complétion de l'écriture reçue sur la carte RDMA. Pour une lecture, on replacera le buffer dans le pool, lorsque le contenu sera copié sur une page locale.

Nous avons évalué les deux stratégies avec l'écriture et la lecture aléatoire d'un fichier de 1 Go avec une taille de cache local de 500 Mo. Le fichier pou-

vant être ainsi présent en totalité, soit dans le cache local, soit dans le cache distant.Le débit mesuré de la lecture aléatoire avec allocation dynamique des buffers est de 99 Mo/s, alors qu'avec un pool de buffers, le débit est de 152 Mo/s. La réutilisation des buffers au travers d'un pool représente donc un réel intérêt pour un meilleur passage à l'échelle.

Aussi, la quantité de buffers pré-alloués est déterminant. Nous avons choisi empiriquement de pré-allouer 128 buffers, soit 500 ko de mémoire. Néanmoins, il est toujours possible, lors d'une grosse éviction d'un fichier, qu'aucun buffer ne soit disponible. Dans ce cas, la page n'est pas envoyée à distance et devra être récupérée sur le disque de manière classique.

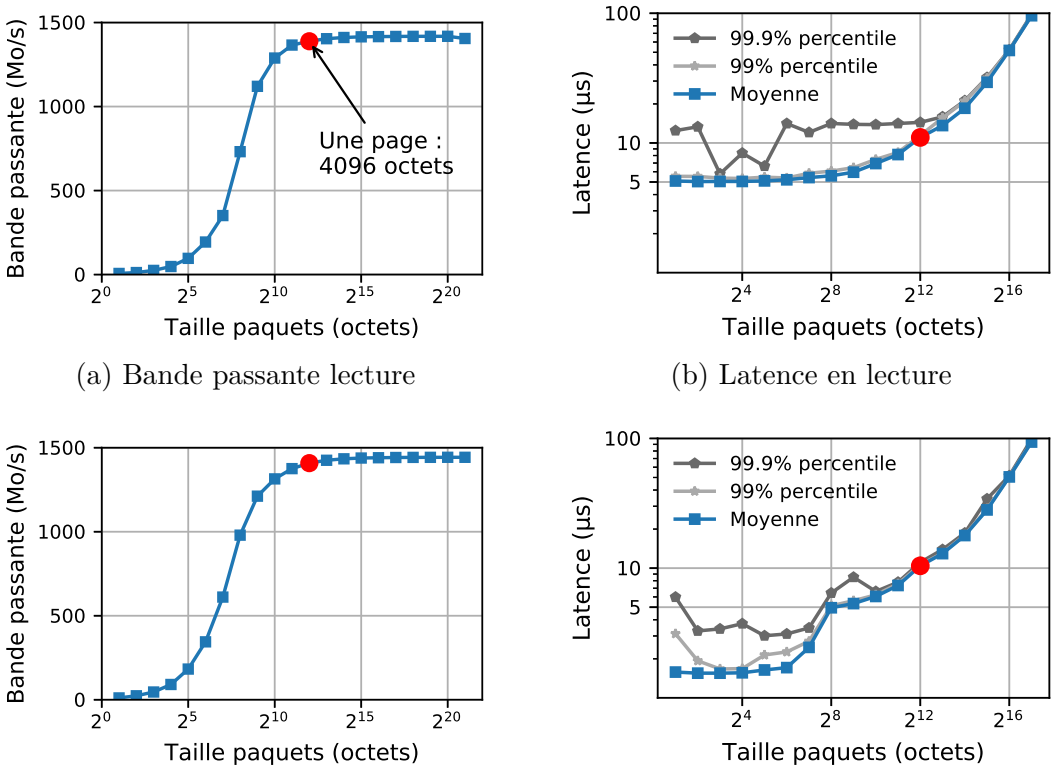
4.3.3.2 Bufferisation des communications

Une technique classique dans la gestion de la mémoire est la bufferisation des requêtes. Il est plus intéressant, par exemple, de demander un grand ensemble de données à un disque en une seule requête, que de les demander en plusieurs petites requêtes.

Il serait alors avantageux de faire la même chose pour Infinicache. Pour l'envoi de pages, donc l'éviction du cache, cela est tout à fait possible. On peut stocker dans un buffer intermédiaire plusieurs pages et les envoyer toutes à distance en un seul appel `RDMA_WRITE`. Malheureusement, la bufferisation des écritures à distance n'a que trop peu d'intérêt car elle se fait de manière asynchrone. En effet, comme on a pu le voir sur la Figure 4.11, l'envoi n'est pas bloquant pour un processus grâce à une copie, et donc sa durée a peu d'importance.

Pour la lecture, cela est plus compliqué. En effet, le code d'Infinicache est exécuté au moment de la requête d'une seule page dans le cache (via la méthode `pagecache_get_page`). La lecture étant synchrone, il est obligatoire d'aller lire à distance la page en question car le processus lecteur est bloqué et en attente. Néanmoins, il peut être intéressant de rapatrier dans le cache un certain nombre de pages consécutives simultanément, pour les futures lectures. On parle de prefetch et ceci est discuté plus tard (§ 4.3.4.3).

Dans le cas des cartes réseaux Infiniband, il est possible d'évaluer le débit (*bandwidth*) et la latence du réseau à l'aide des outils `ib_write_bw` [11] et `ib_write_lat` [12] du package Perftest [14]. Ces résultats sont compilés en



(a) Bande passante lecture

(b) Latence en lecture

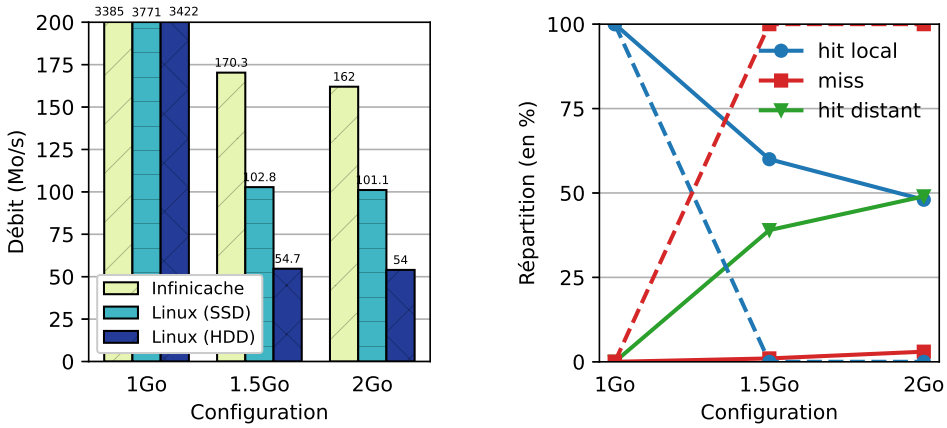
(c) Bande passante écriture

(d) Latence en écriture

FIGURE 4.12 – **Comparatif des lectures et écritures RDMA** en fonction de la taille de la requête. L'axe des abscisses est en log_2. L'axe des ordonnées pour la latence est en *log*. Résultats obtenus à l'aide des outils du package Perftest.

Figure 4.12. Ainsi, pour la lecture d'une page de 4096 octets, le débit moyen est de 1388 Mo/s et la latence de lecture moyenne est de 11.03 μs. Pour la lecture de 16 pages à la fois, c'est-à-dire 2^{16} (= 65536) octets, le débit moyen est de 1416 Mo/s et la latence est de 51.67 μs.

On peut remarquer qu'il est intéressant, du point de vue de la latence, de faire une seule requêtes de 16 pages plutôt que 16 requêtes de taille 4096 octets (51 μs contre 16 fois 11 μs). Les débits des cartes réseaux n'entrent ici pas en ligne de compte car ils sont constants. En contrepartie, les pages doivent être stockées de manière consécutive dans la cluster memory pour pouvoir être lues en une seule requête de 64 ko. La latence d'une requête d'une page reste évidemment inférieure à la latence d'une requête de taille de 16 pages. Ainsi, il semble plus adapté de faire une lecture synchrone (bloquante) d'une seule page lorsqu'une application le demande explicitement, et de lancer en asynchrone une lecture de plusieurs pages pour faire du pré-chargement (prefetch).



(a) Performance d'iozone (b) Detail de l'efficacité du cache

FIGURE 4.13 – **Évaluations avec Iozone séquentiel**. La machine virtuelle effectue deux écritures, puis une lecture de manière séquentielle. Sur la figure (b), le taux de hit et miss est celui d'Infinicache. La commande utilisée est `./iozone -i 0 -i 1 -s size`. Le débit mesuré est celui de la lecture séquentielle.

4.3.4 Benchmarks

Nous utilisons le benchmark Iozone [3] dans plusieurs de ses versions. Pour permettre de comparer, nous avons désactivé le prefetch de l'hôte sur disque car Infinicache ne l'utilise pas et ne l'implémente pas. Nous évaluons l'impact du prefetch en Section 4.3.4.3.

4.3.4.1 Iozone séquentiel

Iozone est un benchmark permettant de faire des lectures et des écritures de tous types, sur des fichiers de taille choisie. Dans cette section, nous mesurons le débit d'une lecture séquentielle d'un fichier de grande taille (1 Go, 1,5 Go et 2 Go). Le benchmark se déroule comme suit : Iozone écrit deux fois séquentiellement le fichier, puis fait une lecture séquentielle du fichier. La taille du cache de fichiers de la VM est toujours de 1 Go. Les résultats de la lecture séquentielle sont réunis en Figure 4.13.

Pour une lecture séquentielle, on remarque que pour la configuration 1 Go les données tiennent entièrement dans le cache, comme on peut le voir sur la Figure 4.13b avec un taux de 100% de hit sur le cache local. Le débit mesuré

est d'environ 3500 Mo/s et correspond au débit de lecture de la mémoire et ne dépend pas du disque ou d'Infinicache.

Pour les configurations suivantes (1,5 et 2 Go), les données ne tiennent pas en totalité dans le cache local. Malgré cela, les performances varient peu selon la taille du fichier, par exemple de 54,7 et 54 Mo/s pour une lecture sur HDD d'un fichier de 1,5 et 2 Go respectivement. En effet, le cache local n'a aucune utilité dans ce cas, car les pages ont été évincées par la LRU avant leur lecture. La taille du cache local étant trop petite pour contenir toutes les données, ces dernières sont évincées avant même d'avoir été utilisées. Nous avons donc un taux de hit qui est de 0% pour Linux sur l'ensemble de ces configurations. Il n'est ainsi pas représenté en Figure 4.13b (il s'agit des résultats d'Infinicache).

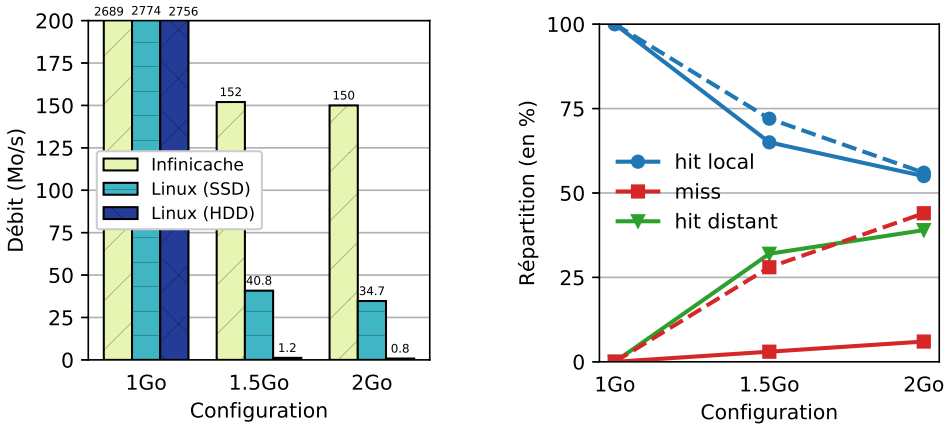
Infinicache obtient des performances supérieures sur les configurations 1,5 et 2 Go. En effet, l'écriture du fichier se fait en première partie du benchmark et ainsi toutes les données du fichier sont présentes soit dans le cache local, soit dans le cache distant, c'est-à-dire la cluster memory. Il n'y a alors quasiment aucun accès disque, à part pour les cas où les pages n'ont pas pu être envoyées à cause de la non disponibilité de buffers d'envoi, qui sont environ de 1% pour 1,5 Go et 3% pour 2 Go.

4.3.4.2 Iozone aléatoire

Les résultats des performances et le détails des ratios hit/miss d'une lecture aléatoire, sont compilés en Figure 4.14. La lecture aléatoire intervient après deux écritures séquentielles de la même manière que lors de la lecture séquentielle précédente.

Comme précédemment, en configuration 1 Go, les données sont entièrement contenues dans le cache local. Le débit mesuré est ici aux alentours de 2700 Mo/s pour tous les types de support. Le taux de cache hit est bien de 100% pour cette configuration, comme on peut le voir en Figure 4.14b.

Dès lors que les données ne sont plus contenues dans le cache (configuration 1,5 et 2 Go), les performances chutent drastiquement par rapport à une lecture séquentielle, surtout pour la version avec un disque dur HDD. Par exemple, une lecture aléatoire d'un fichier de 2 Go, le débit mesuré est de 0,8 Mo/s pour un HDD, de 34,7 Mo/s pour un SSD et de 150 Mo/s pour Infinicache. Les performances varient également entre la lecture de 1,5 et 2 Go car le taux de



(a) Performance d'iozone (b) Detail de l'efficacité du cache

FIGURE 4.14 – **Évaluations avec Iozone aléatoire**. La machine virtuelle effectue une écriture séquentielle d'une fichier, puis lit de manière aléatoire. En (b), les traits pleins représentent le taux de hit et miss de Infinicache, en pointillés ceux de Linux, qui sont les mêmes pour la version HDD et SSD. La commande utilisée est `./iozone -i 0 -i 2 -s size`. Les performances sont celles de la lecture aléatoire.

hit et donc miss du cache local varie.

Pour mettre cela en perspective, nous pouvons comparer avec des valeurs théoriques. Les temps d'accès à une page unique de 4096 octets sur un disque HDD est théoriquement de l'ordre de 5 ms et de 80 ns pour l'accès à une page sur la mémoire vive. D'après la Figure 4.14b, le taux de hit dans le cas de Linux (en pointillé) pour un fichier de 2 Go est de 56% et donc 44% de miss, i.e. des lectures sur disque. Un fichier de 2 Go représente 524 822 pages de 4096 octets. Nous avons donc environ 293 000 pages lues en mémoire et 231 000 pages lues sur disque. Ainsi, théoriquement nous avons une exécution de la lecture aléatoire de 1 178 secondes, soit un débit de 911 ko/s ou 0,9 Mo/s, ce qui correspond aux valeurs mesurées.

Le côté aléatoire des lectures a un impact conséquent sur les performances pour un disque dur HDD. Cela s'explique par les déplacements des têtes de lectures du disque, qu'il n'a pas lors de lectures ou d'écritures séquentielles. En effet, son débit de lecture séquentielle de 2 Go est de 54 Mo/s (cf. Fig. 4.13a), soit dans ce cas-là un temps d'accès moyen de 80 μs par page (contre un débit de 0,8 Mo/s et une temps d'accès par page de 5 ms en aléatoire).

Ce handicap n'existe pas pour un disque SSD mais les performances sont

tout de même divisées par deux ou trois. Les performances d'Infinicache restent comparables à une lecture séquentielle et sont très compétitives.

Dans le cas d'Infinicache, pour une lecture de 2 Go, le débit de la lecture aléatoire est de 150 Mo/s et possède un taux de hit à distance sur la cluster memory de 39%. Nous pouvons ainsi calculer le temps de latence moyen d'accès à une page à distance, en se basant sur un accès théorique de 80 ns pour de la mémoire locale. Ainsi, nous accédons à une page à distance en 40 μs. Cela est légèrement supérieur aux évaluations du micro-benchmark (cf. § 4.3.2), sûrement dû à la structure de données employée (un rb tree), avec la recherche et l'ajout dans cette structure qui grandit beaucoup avec le benchmark.

4.3.4.3 Prefetch du cache

Les évaluations précédentes ont été effectuées sans l'aide du mécanisme de prefetch du cache Linux. Le but du prefetch est de détecter un accès séquentiel à des données et ainsi de les pré-charger en mémoire de manière asynchrone. Les accès suivants bénéficient d'une latence très faible car la donnée sera directement disponible dans le cache.

En reproduisant les mêmes expérimentations avec la lecture séquentielle mais avec le prefetch activé et en reprenant les résultats de la Figure 4.13 (sans le prefetch activé), nous pouvons comparer les deux. Les résultats sont compilés en Table 4.2.

Nous effectuons de même les mesures avec la lecture aléatoire mais avec le prefetch activé et nous reprenons les résultats de la Figure 4.14 (sans le prefetch activé). Les résultats sont compilés en Table 4.3.

La première observation sur la Table 4.3 est que les performances avec et sans prefetch sont relativement identiques. On se rend bien compte que le noyau n'arrive pas à deviner les futurs accès aux pages et ainsi n'en anticipe aucun. Le prefetch n'a donc aucune utilité pour une lecture aléatoire sur disque. Infinicache a donc de meilleures performances dans tous les cas en lecture aléatoire.

En revanche, pour une lecture séquentielle (Table 4.2), le prefetch est très performant et nous observons une amélioration des débits de lecture de 2 à 3 fois avec le prefetch activé. Le principe du prefetch dans Linux est de définir une taille de fenêtre que le processus en arrière-plan va charger. La dernière page de

	SSD		HDD		IC/SSD		IC/HDD	
Config/Prefetch	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
1 Go	3765	3771	3582	3422	N/A	3385	N/A	3462
1,5 Go	276,6	102,8	141,5	54,7	N/A	170,3	N/A	165
2 Go	276	101,1	124	54	N/A	162	N/A	157

TABLE 4.2 – **Comparatif du débit d'une lecture séquentielle** avec Iozone, avec ou sans le prefetch. Les débits sont données en Mo/s

	SSD		HDD		IC/SSD		IC/HDD	
Config/Prefetch	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
1 Go	2789	2774	2758	2756	N/A	2650	N/A	2721
1,5 Go	41	40,8	1,2	1,2	N/A	152	N/A	146,6
2 Go	34,3	34,7	0,7	0,8	N/A	150	N/A	145

TABLE 4.3 – **Comparatif du débit d'une lecture aléatoire** avec Iozone, avec ou sans le prefetch. Les débits sont données en Mo/s

cette fenêtre de chargement est marquée comme `PageTail` par le noyau. Ainsi, lorsque le noyau lit une telle page, le processus décale la fenêtre et charge de nouvelles pages depuis le disque. Il semble que la taille de la fenêtre soit particulièrement bien calibrée pour le SSD, qui arrive à avoir des débits de lectures quasi identiques pour toutes les configurations de mémoire, 276,6 et 276 Mo/s pour 1,5 et 2 Go respectivement, alors que le HDD a un débit de 141,5 et 124 Mo/s pour 1,5 et 2 Go.

Finalement, il serait ainsi très intéressant pour Infinicache d'avoir le même comportement pour améliorer les performances. Cela change quelque peu le fonctionnement du chargement des pages, et il est nécessaire de créer un thread kernel pour charger les pages, de manière asynchrone, dans le cache local depuis le cache distant. Il s'agit d'une perspective à court terme d'évolution de la solution Infinicache.

4.4 Synthèse

Dans ce chapitre, nous avons présenté notre contribution Infinicache, un système permettant d'exploiter la mémoire libre du datacenter, en proposant une extension du cache de fichiers.

Dans un premier temps, nous avons mis en lumière (§ 4.1.1) les problèmes liés à la consolidation de machines virtuelles dans les centres de données de type IaaS, qui entraînent un taux d'utilisation de la mémoire globale non optimal du centre. Notre contribution s'attelle à exploiter ces trous de mémoire. Pour cela, il est nécessaire d'utiliser un réseau à haute vitesse, ici Infiniband (§ 4.1.2). Une particularité d'Infiniband est d'être un réseau RDMA, c'est-à-dire avec la capacité de lire et écrire sur de la mémoire à distance directement sans l'intervention du processeur distant.

Notre solution Infinicache (§ 4.2) exploite la mémoire disponible des machines distantes à l'aide du réseau RDMA. Pour cela, Infinicache est composé de deux modules, un client intégré directement dans le cache de la machine virtuelle qui va gérer le cache distant, et un serveur qui fournit de la mémoire que la machine a à sa disposition. Nous avons proposé deux implémentations du client et serveur, send/receive s'appuyant sur une communication par message et qui pourrait être utilisée avec un type de réseau autre qu'Infiniband, et read/write se basant sur les opérations particulières de lecture et écriture de mémoire à distance. C'est cette dernière qui a été utilisée pour l'évaluation.

Enfin, nous avons évalué Infinicache (§ 4.3). D'abord, nous avons évalué les temps de lecture et d'écriture d'une seule page sur de la mémoire à distance (§ 4.3.2), puis le passage à l'échelle de la solution (§ 4.3.3). Enfin, nous avons utilisé le benchmark Iozone pour évaluer Infinicache (§ 4.3.4). Iozone a été utilisé pour faire des lectures séquentielles et aléatoires de fichiers de grande taille. Les performances d'Infinicache sont très compétitives, en particulier lors de lectures aléatoires. Néanmoins, elles pourraient être améliorées à l'aide du prefetch du cache distant.

Chapitre 5

Etat de l'art

Contenu

Nous avons décrit deux contributions à la gestion des caches dans les systèmes de fichiers dans les infrastructures virtualisées. La première visait à éviter la duplication des pages dans les caches et la seconde visait l'extension des caches par de la mémoire distante. Dans ce chapitre, nous comparons les travaux de cette thèse avec l'état de l'art pour ces deux contributions dans les deux sections suivantes (§ 5.1 et § 5.2).

5.1 Duplication de cache

Cacol s'attaque au problème de duplication des pages du cache dans un environnement virtualisé, à l'aide d'une politique d'éviction du cache au niveau de l'hôte. Ce problème n'est pas nouveau et nous allons voir dans un premier temps l'évolution des politiques de gestion de cache (§ 5.1.1), et ensuite les problèmes relatifs à la virtualisation (§ 5.1.2).

5.1.1 Politiques d'éviction du cache

Les politiques de gestion de cache et de la mémoire en général, ont été étudiées depuis longtemps et sont une des fondations des systèmes d'exploitation. L'efficacité d'un algorithme de remplacement du cache est critique pour obtenir des performances satisfaisantes et stables sur les systèmes d'entrés/sorties. La plus connue et répandue, grâce à sa simplicité, est la politique LRU [41] (*Least recently used*) qui consiste à évincer les pages les moins récemment utilisées, mais cette dernière peut s'avérer très inefficace dans le cas de certaines charges de travail ou motifs d'accès aux disques.

Par exemple, pour un accès cyclique à un fichier qui est juste plus grand que le cache, la politique LRU va toujours supprimer les pages qui vont être accédées dans le futur proche car il s'agit des pages les moins récemment accédées. Une solution simple à ce problème est de recourir à l'algorithme opposé, le MRU (*Most recently used*). Ce dernier évince les pages les plus récemment accédées du cache. Bien qu'améliorant les performances dans ce contexte particulier, cette politique s'adapte mal sur les workloads classiques d'un système d'exploitation, qui réutilise fréquemment les mêmes pages. La politique EELRU [143] (*Early Eviction LRU*) est proposée pour répondre également à ce problème tout en conservant les avantages de la LRU sur le long terme. Cette politique se base

sur les dates d'accès comme LRU mais aussi sur des liens de dépendance entre les pages qui sont mises dans le cache.

Un autre exemple d'inefficacité de la LRU est lors de la lecture subite (*burst*) d'un gros bloc d'un fichier accédé rarement, venant ainsi remplacer les pages fréquemment accédées par des données inutiles, gaspillant ainsi de la mémoire. La politique 2Q [85] (*two queues*) sépare les pages du cache en deux, les pages chaudes qui sont lues fréquemment, et les pages froides qui ne sont accédées que rarement. Ainsi, lors de la lecture en burst, seules les pages froides sont remplacées, laissant les pages chaudes, celles fréquemment accédées, dans le cache. La politique LRU-K [114] généralise ce comportement en proposant une séparation en K listes. La politique LRU-2 est donc sensiblement la même que la 2Q, et représente le meilleur compromis pour des workloads typiques d'un système d'exploitation.

Partant de ce constat, des algorithmes se basant non pas sur la temporalité mais sur la fréquence d'accès des pages ont fait leur apparition. La politique LFU [129] (*Least frequently used*) évince de son cache en priorité les pages les moins fréquemment utilisées répondant à certains défauts du LRU comme la sensibilité aux burst. De même, cette politique ne convient pas forcément à toutes les charges de travail et motifs d'accès. La politique LFU comportant de nombreux défauts, elle est souvent combinée avec la politique LRU. La politique LRFU [93, 92] propose un compromis entre LRU et LFU, en favorisant plus ou moins l'une ou l'autre.

Des travaux ont exploité la présence de régularités sur des accès séquentiels, de boucle, ou des métriques plus complexes pour proposer des politiques telles qu'UBM [89] (*Unified Buffer Management*), LIRS [83] (*Low Inter-reference Recency set*), LIP [127] (*LRU Insertion Policy*) ou encore ARC [101] (*Adaptive Replacement Cache*). Ces systèmes visent à capturer un motif d'accès aux fichiers. L'essence de ces politiques est d'essayer de prédire le futur et ainsi influencer ce que l'on garde dans le cache pour optimiser son efficacité.

La plupart de ces politiques de remplacement de cache sont efficaces et peuvent fonctionner aussi bien pour du remplacement dans les lignes de cache processeur, de cache mémoire ou même de cache distant. Malheureusement, le passage de l'algorithme théorique à une implémentation pratique est souvent rendu compliqué par les comportements techniques de gestion de la mémoire. Une implémentation qui est particulièrement adaptée à la gestion de la mémoire est la solution CLOCK [44], développée il y a plus de 50 ans. C'est une

approximation de la politique LRU qui ne replace pas les pages accédées en tête de liste lors d'une lecture, mais simplement la marque à l'aide d'un drapeau. Les pages dans la liste sont vérifiées de manière périodique et sont évincées du cache si leur drapeau n'est pas levé. Ainsi, cette implémentation est bien adaptée à la gestion de la mémoire car elle a un très faible coût à l'exécution, il n'y aucune modification de liste, de déplacement mémoire, simplement la mise en place d'un drapeau lors d'un accès. Tous les systèmes d'exploitation actuels utilisent une implémentation CLOCK ou dérivée de celle-ci, que ce soit Unix et Linux avec les principes d'une politique LRU-2 mais une implémentation en CLOCK, ou Windows [63]. Des améliorations ont été proposées avec la politique CLOCK-Pro [82] qui tient également compte des motifs d'accès aux fichiers, sans pour autant être exploitées aujourd'hui.

Dans Cacol, nous exploitons dans le cache de l'hôte une politique d'éviction de cache implémentée en CLOCK, en se basant sur la fréquence d'accès, proche de LRFU.

5.1.2 Politiques dans les environnements virtualisés

Les politiques présentées jusqu'à présent avaient pour objectif d'améliorer les performances de la gestion mémoire dans un système d'exploitation classique. La virtualisation modifie la problématique. Dans ce cas, nous obtenons deux niveaux de cache de fichiers, le premier niveau dans la VM, et le second dans l'hyperviseur. Une politique classique, par exemple LRU, convient très bien pour le cache de la VM, mais pas pour le cache de l'hyperviseur. Le cache de l'hyperviseur agit comme un cache de seconde chance [168].

La dualité des caches hôte et invité amène le problème de la duplication des pages entre ces deux caches. De nombreux systèmes ont été proposés pour répondre à ce problème. Un autre problème est la prédictibilité, en effet, désormais il y a plusieurs VM utilisant le cache hôte et donc elles peuvent interférer entre elles.

5.1.2.1 Duplication des pages du cache

Les premiers algorithmes proposés pour empêcher la duplication des pages sont Multiqueue [169] et Demote [161]. Ce dernier implémente un cache exclusif,

c'est-à-dire que les données se retrouvent exclusivement soit dans le cache invité, soit dans le cache hôte. Pour arriver à ce résultat, Demote nécessite une nouvelle commande dans le standard de bus SCSI (*Small Computer System Interface*), pour permettre de notifier l'hôte d'une "rétrogradation", d'où le nom *demote*. Dans ce cas, cela veut dire qu'une page est évincée du cache invité et envoyée au cache hôte.

Cacol est différent car il ne modifie pas le système d'exploitation invité et n'intègre donc pas de commandes supplémentaires dans ce dernier. Cacol voit l'OS invité comme une boîte noire. C'est pour cette raison qu'il n'est pas possible dans notre cas d'obtenir un cache strictement exclusif. En effet, comme il n'existe aucun moyen de savoir si une page est présente ou pas dans le cache invité, l'hôte doit faire des suppositions et peut donc être amené à conserver parfois des pages dupliquées. L'objectif de la politique Cacol est de minimiser ces duplications de pages autant que possible.

Singleton [140] est un système développé dans KVM et s'attaque au problème de duplication des pages des caches. Singleton s'appuie sur KSM [19] (*Kernel Samepage Merging*) pour identifier les pages dupliquées depuis l'hôte. L'idée principale est de scanner la mémoire RAM à intervalle régulier (de l'ordre de 30 à 60 secondes dans l'article), de garder une valeur hachée (un *hash*) de chaque page dans une table de hachage, et lorsque l'on détecte un hash identique, on en déduit que la page est dupliquée. A ce moment-là, Singleton décide de ne conserver qu'une seule des deux pages, et marque la page en mode Copy-On-Write. Ce mode a pour effet de re-dupliquer la page lors d'une écriture, car les pages ne sont alors plus identiques. Singleton affirme obtenir une réduction de la taille du cache hôte d'un ordre de magnitude et doubler le ratio de cache hit dans l'hôte par rapport à une politique par défaut. Ce système est intéressant par son approche mais présente un overhead important de part la mémoire supplémentaire utilisée pour stocker les hashs mais aussi pour les calculer, jusqu'à 20% d'utilisation CPU. Enfin, contrairement à Cacol, Singleton scanne et lit la mémoire de la machine virtuelle, en tant qu'utilisateur privilégié de la machine. Cacol, à l'inverse, ne se permet aucun accès à la mémoire de la machine virtuelle.

Sky [20] est une extension d'un système de surveillance de VM. Ce dernier rassemble des indices tels que la taille des fichiers, les meta-data, le contenu des fichiers en interceptant les appels systèmes de la machine virtuelle. Cela permet à Sky d'adapter sa politique d'éviction dans le cache hôte en fonction de ces indices. Il présente une grande intrusivité vis-à-vis de la VM en interceptant

les appels systèmes effectués. Cacol agit, quant à lui, en boîte noire vis-à-vis de la VM.

5.1.2.2 Prédictibilité des performances

Une partie des contributions de Cacol est d'éviter les interférences entre VM au niveau du cache hôte.

Per-VM page cache [141] propose de partitionner le cache de l'hôte en plusieurs, un pour chaque machine virtuelle s'exécutant sur la machine physique. L'objectif est de réduire les interférences entre VM dans le cache hôte, et de permettre de définir une politique pour chaque VM. Ce travail se rapproche de Cacol sur l'objectif d'obtenir une équité entre VM. Per-VM permet à l'administrateur de définir ses propres politiques et limites de mémoire de chaque VM mais ne mutualise pas les ressources mémoires entre VM, car chaque VM a une portion du cache définie.

Moirai [144] est un système prenant en compte le profil de charge des VM, qui fournit des outils à l'administrateur du cloud pour gérer le cache à l'échelle du data-center. L'administrateur peut alors définir une politique d'éviction différente pour les VM, l'hôte et le système de fichiers ou disques partagés en réseau. Comme Per-VM, Moirai permet à l'administrateur de configurer chaque couche de cache. Une telle politique permettrait d'implémenter la régulation de la mémoire comme Cacol.

5.2 Mémoire à distance

L'idée principale d'Infinicache est d'exploiter la mémoire d'une autre machine pour étendre le page cache de sa propre machine à l'aide d'un réseau RDMA.

Nous étudions dans un premier temps l'impact de l'avènement des réseaux rapides (§ 5.2.1) et les systèmes qu'ils ont amenés. Ces réseaux ont permis la création de systèmes de mémoires partagées et distribuées (§ 5.2.2). Ils ont aussi débouché sur l'utilisation de mémoire distante comme support de swap (§ 5.2.3). Finalement, la plupart de ces systèmes exploitent une hiérarchie entre les différents types de mémoires disponibles que nous analysons en § 5.2.4.

5.2.1 Avènement des réseaux rapides

Les connexions réseaux ont été longuement étudiées, et de nombreux réseaux et protocoles réseaux ont été proposés au fil du temps. Le réseau le plus utilisé aujourd'hui est Ethernet [103] créé en 1973. Le réseau a su s'adapter en augmentant les vitesses de transmission passant au fil des ans de 3 Mb/s pour le design original, à 10 Mb/s, à 100 Mb/s, puis à 1 Gb/s devenant ainsi le Gigabit Ethernet [137] en 1998. Aujourd'hui, le réseau Ethernet a une vitesse de transmission pouvant aller jusqu'à 400 Gb/s [159].

D'autres réseaux sont apparus comme le réseau ATM [60] (*asynchronous transfer mode*) en 1994, proposant un protocole de communication à taille de paquets fixes avec un vitesse de transmission mesurée de 150 Mb/s et pouvant a priori être encore plus importante. En 1995, Myrinet [28] est le premier à proposer un débit supérieur à 1 Gb/s. Il se différencie d'Ethernet en apportant une meilleure tolérance aux erreurs, un contrôle de flux, du contrôle d'erreur, et de la surveillance d'état de chaque connexion physique, principalement gérée par les commutateurs réseau.

En 1999, Infiniband [122] propose un nouveau type de réseau avec plusieurs largeurs de liens et un débit maximal de 6 Gb/s pour les liens 12X. Aujourd'hui, la technologie a évolué et propose de nombreuses déclinaisons de liens, le plus puissant étant le HDR (*High Data Rate*) à 600 Gb/s avec 12 liens. En 2020, Infiniband annonce sortir un nouveau lien nommé NDR (*Next Data Rate*) à 1,2 Tb/s [149]. En plus d'une avancée dans le débit des cartes et une latence encore plus réduite, Infiniband apporte RDMA [123], la lecture et écriture de mémoire à distance sans intervention du processeur distant.

A chaque fois qu'une avancée technologique sur les réseaux à haute vitesse a eu lieu, de nouveaux systèmes sont sortis dans le domaine des mémoires partagées et distribuées. En effet, un réseau de machines ressemble à une machine multi-processeurs, les liens réseaux devenant les bus mémoire. Cela a donné naissance aux systèmes à image unique (SSI, *Single System Image*), c'est-à-dire un seul OS pour un cluster de machines avec une mémoire gérée globalement. De façon moins extrême, les systèmes de mémoire partagée amènent le même concept sans pour autant n'avoir qu'un seul OS.

5.2.2 Systèmes de mémoire partagée et distribuée

L'idée de globaliser la mémoire et donc d'utiliser de la mémoire à distance n'est pas nouvelle, un des pionniers dans ce domaine est Li [94] en proposant une mémoire partagée et distribuée. Ce dernier n'essaie pas d'agrandir le cache mais d'unifier l'ensemble de la mémoire d'un cluster. L'objectif est d'agglomérer toutes les activités de la mémoire, que ce soit les fichiers mappés, les accès aux disques, ou la mémoire anonyme, et de ne voir la mémoire que comme une seule entité au niveau du cluster.

Ce travail a créé une branche de recherche sur les DSM (*Distributed Shared Memory*) [36, 95, 111, 84, 47] qui exposent un espace d'adressage global et partagé aux applications utilisateurs sur un réseau de machines, mais aussi une autre branche sur les systèmes à image unique (SSI), dont l'idée étant de n'avoir qu'un seul système d'exploitation à échelle du cluster et donc une seule mémoire.

Les avancées sur les réseaux ont permis l'apparition des systèmes à image unique. En 1993, MOSIX [21] propose un tel système implémenté avec Myrinet et est la base des futurs systèmes à image unique en définissant toutes les caractéristiques de ce type de système, comme par exemple une transparence du réseau ou un contrôle décentralisé. En 1998, Kerrighed [107], un projet de l'INRIA propose une implémentation dans Linux d'un tel système. Pink [158] est un prototype en 2003 d'un système à image unique de 2048 processeurs, basé sur Myrinet. Ce genre de systèmes sont très utilisés dans le monde du calcul haute performance (HPC).

Moins contraigant qu'un SSI, les DSM ont longtemps souffert d'une forte latence de communication pour maintenir la cohérence entre les différentes mémoires distribuées entre les machines. Pour remédier au problème de cohérence, la communauté HPC a utilisé le modèle PGAS (*Partitioned Global Address Space*) [35, 38, 48, 165] qui définit des langages de programmation pour fournir une abstraction similaire à de la mémoire partagée mais avec de hautes performances et de la scalabilité. Ce modèle nécessite ainsi une complète réécriture des applications utilisateur dans un langage supportant le PGAS, comme l'UPC (*Unified Parallel C*) [43], Titanium [71, 15] un dialecte scientifique de Java, ou CAF (*Co-Array Fortran*) [112], pour pouvoir utiliser de façon explicite la mémoire à distante.

Avec l'arrivée de RDMA et de sa très faible latence, nous avons pu observer

un renouveau d'intérêt sur les DSM. La majeure partie de ces travaux [87, 106, 108, 116, 126] s'oriente vers une utilisation de la mémoire en mode base de données clé-valeur, comme par exemple avec le très célèbre Memcached [61], ou ses dérivés [109, 56]. Par exemple, FaRM [54] est un système de Microsoft, qui montre la mémoire du cluster comme une mémoire partagée et exploite RDMA pour améliorer d'un ordre de magnitude la latence et le débit par rapport aux systèmes DSM utilisant TCP/IP. Sa particularité est de proposer une communication RDMA sans verrou sur la mémoire.

Toutes ces solutions présentées s'adressent généralement à des applications parallèles. En particulier dans un système PGAS, il est nécessaire de réécrire les applications utilisateurs pour permettre l'utilisation de la mémoire distante partagée. Notre implémentation d'Inifinicache s'adresse à l'ensemble des applications car toutes utilisent le cache de fichiers.

5.2.3 Swap-out distant

Nous venons de voir l'utilisation d'une mémoire distribuée impliquant une gestion globale de la mémoire dans un centre de données et ses améliorations à travers le temps, notamment grâce à RDMA. Une autre idée, plus simple à mettre en œuvre, est de stocker les pages de mémoire anonyme (swap-out), non pas sur un disque swap local mais sur un disque distant, grâce à un réseau plus rapide qu'un accès disque.

GMS [57] (*Global Memory Management*), qui avec l'arrivée d'ATM, propose en 1995 un gestionnaire de mémoire à l'échelle du cluster. Contrairement à des travaux antérieurs [42, 58, 34, 27], GMS présente une implémentation réelle directement dans un système d'exploitation OSF/1. La mémoire anonyme peut être envoyée à distance en se servant des réseaux haute vitesse de l'époque [18], ATM en l'occurence.

Cette idée a été exploitée dans de nombreux projets [17, 39, 55, 62, 99], plus récemment Infiniswap [67] exploite RDMA pour proposer un système présentant les meilleures caractéristiques en terme de performances. Il a particulièrement inspiré mes travaux sur Infinicache. Ses caractéristiques principales sont de se présenter sous la forme d'un bloc-device, d'être décentralisé dans sa gestion de la mémoire du centre de données, et de ne pas utiliser le processeur de la machine distante grâce à RDMA, au contraire de travaux similaires comme HPBD [96] et Mellanox nbdX [5].

La principale différence avec Infinicache est que ce dernier a pour objectif de stocker des pages de fichiers, sur une mémoire distante, et non des pages mémoire anonymes. Ainsi, leur positionnement dans le noyau est complètement différent, Infiniswap étant un block device recevant des block I/O (BIO) de la part du système d'exploitation, alors qu'Inifinicache se situe directement dans le cache. Il peut alors exploiter les spécificités du cache comme la possibilité d'évincer des pages ou de mettre en place des politiques de préchargement, ouvrant ainsi de nombreuses perspectives.

5.2.4 Hiérarchie mémoire

On a assisté à une multiplication des couches mémoires provenant de l'apparition de supports aux caractéristiques différentes (taille, coût, vitesse). Ainsi, les systèmes de cache ont toujours exploité cette disparité entre les différentes latences et débits des couches mémoires. Un HDD est moins rapide qu'un SSD, qui est moins rapide que de la mémoire distante, qui est moins rapide que de la mémoire locale.

Les évolutions techniques bousculent ces hiérarchies. Prenons l'exemple des systèmes de fichiers distribués (DFS, cf. § 2.3.2) qui utilisent un ensemble de machines pour créer un unique système de fichiers partagés accessible via le réseau. Les différentes parties communiquent en utilisant des protocoles spécifiques pour stocker les données et méta-données. L'arrivée de RDMA a relancé les recherches dans ce domaine. Une première amélioration simple a été de remplacer le protocole réseau par un protocole RDMA [32, 46, 147, 163, 37, 146], puis de prendre en compte le changement de coût relatif de la mémoire distante [31, 77]

Puis d'autres bouleversements se sont produits avec l'arrivée des NVM (*Non-Volatile Memory*) [33, 74], une nouvelle technologie mémoire persistante plus rapide que RDMA, amenant une nouvelle couche dans ce mille-feuille qu'est la mémoire. Par exemple, il existe NVFS [78] une version optimisée de HDFS, qui exploite RDMA et NVM. Les références dans le domaine sont HotPot [139], Octopus [98] et son successeur Orion [164]. Leur objectif est de coupler l'utilisation de RDMA avec un système de fichiers, qui est ici distribué et d'exploiter les disparités de vitesse entre NVM et RDMA.

Infinicache se situe dans la même lignée que ces travaux, car il tire profit de cette hiérarchie entre les différentes couches de mémoire. Dans notre cas, nous

exploitons la différence entre bande passante RDMA et vitesse de lecture sur un disque local. De même que les bouleversements qui ont impacté les DFS, les mêmes évolutions technologiques peuvent également impacter Infinicache.

Chapitre 6

Conclusions et perspectives

Contenu

6.1 Conclusions

Le Cloud computing est en plein essor depuis plusieurs années, grâce à de fortes demandes en calculs informatiques. Le cloud se base sur une externalisation des services informatiques des entreprises dans des centres de données, gérés par des fournisseurs de cloud. Le fournisseur propose ainsi une mutualisation des ressources informatiques et apporte une réduction des coûts humains et matériels grâce à l'économie d'échelle. La virtualisation se pose comme la technologie de base du cloud en partageant une machine physique du centre de données en plusieurs machines virtuelles à l'aide d'un hyperviseur. Les machines virtuelles invitées partagent les ressources de la machine physique, l'hôte, et particulièrement la mémoire. La mémoire est une ressource limitée dans les centres de données virtualisés et doit être partagée entre les VM. La mémoire permet entre autre d'accélérer les coûteux accès aux disques de stockage locaux ou en réseau, en conservant les pages mémoire fréquemment accédées ou en pré-chargeant des données dans le cache de fichiers pour prévenir des accès aux disques. Nous nous sommes intéressés, dans le cadre de cette thèse, aux optimisations du cache de fichiers dans un environnement virtualisé.

Nous avons identifié deux axes d'amélioration du cache de fichiers dans un environnement virtualisé ; la première est d'optimiser l'utilisation du cache en réduisant la duplication des pages dans les caches de l'hôte et de l'invité, la seconde est d'étendre la capacité du cache de fichiers grâce à de la mémoire à distante disponible dans le centre de données.

Notre première contribution est Cacol, une politique d'éviction du cache de l'hôte pour réduire la duplication des pages entre les invités et l'hôte et ainsi éviter du gaspillage de la mémoire. Nous avons, dans un premier temps, estimé l'étendue du problème de la duplication dans les caches en observant la quantité de mémoire gaspillée sur certaines workloads et l'impact sur les performances des applications. Une solution naïve, qui est de supprimer le cache de l'hôte et donc la duplication, n'est pas envisageable au vu des performances mesurées. Nous proposons alors Cacol, une politique adaptée à la nature du cache de l'hôte d'être de seconde chance. Cacol identifie les pages succeptibles de convenir au cache hôte en se basant sur la fréquence et la temporalité d'accès des entrées et sorties de la VM, et garantit une équité entre VM pour empêcher la monopolisation de la mémoire par une VM trop intensive en I/O. Nous avons montré que Cacol pouvait dans le meilleur des cas réduire de 87,5% la quantité de cache de l'hôte utilisée par des VM sur des charges de travail intensives

en I/O. Nous avons également observé une augmentation des performances normalisées des benchmarks s'exécutant seul de 11% et jusqu'à 21% dans le cas de plusieurs benchmarks colocalisés sur la même machine physique. Enfin, nous avons pu évaluer l'efficacité de la politique d'équité et les impacts bénéfiques sur les VM.

Notre seconde contribution est Infinicache, un système implémenté dans la VM et qui propose une extension du cache de fichiers en exploitant la mémoire disponible du centre de données. Nous supposons que la mémoire inutilisée par les VM a pu déjà être exploitée par des VM de la même machine physique. Le réseau à haute vitesse Infiniband est utilisé pour bénificier d'une latence très faible par rapport à des accès sur le disque local, afin que notre système puisse tirer profit de cette disparité de vitesse. Pour profiter au mieux du réseau RDMA, nous utilisons les verbs read et write qui fournissent la possibilité de lire et écrire sur de la mémoire à distance sans l'intervention du processeur distant et donc sans aucun impact sur les performances de la machine distante. Dans son implémentation, Infinicache se décompose en deux modules, le serveur fournissant de la mémoire inutilisée d'une machine, et le client se chargeant de gérer et de maintenir la cohérence du cache distant. L'intégration du client se fait par une modification minimale du noyau. Nos évaluations montrent tout d'abord que la communication RDMA permet d'obtenir des temps d'accès plus faibles qu'avec un disque local. Nous montrons ensuite qu'Infinicache est capable d'améliorer les performances des applications par extension du cache de fichiers, dans le cas d'accès aléatoires comme dans le cas d'accès séquentiels (dans le cas où le prefetch n'est pas activé).

6.2 Perspectives

Au cours de la réalisation de cette thèse, nous avons identifié plusieurs axes potentiels d'amélioration des contributions. Nous pouvons catégoriser ces axes d'amélioration en deux, d'abord des améliorations réalisables à court terme (§ 6.2.1) et celles à plus long terme (§ 6.2.2).

6.2.1 Améliorations à court terme

Evaluation à l'échelle d'un datacenter. Nous avons évalué les travaux de cette thèse à l'aide de benchmarks pour mesurer les performances, l'intrusivité et les bénéfices que l'on peut en obtenir sur une machine physique ou sur un réseau de machines. Il serait intéressant d'utiliser des traces de datacenters pour observer le comportement des différents systèmes à l'échelle d'un datacenter et évaluer les bénéfices possibles à une plus grande échelle.

Politique globale de gestion mémoire dans le cluster. Infinicache permet une extension du cache de fichiers pour les VM. Les décisions de répartions des ressources disponibles entre les VM se font de manière décentralisée, c'est-à-dire que chaque VM décide indépendamment de ses besoins et donc sur quelles machines elle récupère des ressources. De même, une machine physique va décider, indépendamment de l'état global de la mémoire du centre de données, à quelle VM fournir ses ressources. La première approche que nous avons choisie et implémentée est de simplement donner la mémoire au premier qui la demande. Il serait intéressant d'expérimenter avec différentes politiques de gestion globale de la mémoire disponible.

Prefetch du page distant. Comme nous avons pu le voir à la fin du chapitre sur Infinicache (§ 4.3.4.3), il serait très intéressant, pour notre système d'extension de cache, de proposer un module de pré-chargement asynchrone des pages distantes. Les débits et latences mesurés pour les disques et réseaux nous laissent penser que ce module de préchargement permettrait à Infinicache de se comparer favorablement à Linux (HDD ou SSD) même dans le cas d'accès séquentiels avec prefetch.

6.2.2 Améliorations à long terme

Une possibilité d'amélioration à long terme d'Infinicache est, qu'au lieu d'utiliser la notion de cache distant dans l'invité, d'utiliser cette même notion dans le cache de l'hôte. De plus, cette extension de cache distant est complémentaire avec Cacol qui évite de gaspiller de la mémoire du cache de l'hôte. Ceci permet d'allouer moins de mémoire à l'hôte pour en donner plus aux VM. Nous pourrions également évaluer la complémentarité d'Infinicache et Cacol dans leur inplémentation actuelle.

L'état de l'art a permis de mettre en perspective la hiérarchie mémoire qui se crée avec les différents types de support existants. Ainsi, il serait intéressant de travailler avec de nouvelles technologies, proposant de nouvelles hiérarchies mémoire, tels que les NVM ou des cartes réseau de dernière génération.

Enfin, il serait intéressant d'étudier l'application de nos contributions (Cacol et Infinicache) dans des systèmes de fichiers distribués de type SAN ou NAS, comme par exemple GPFS ou NFS.

Bibliographie

[1] Kallsyms. https://jrgraphix.net/man/K/kallsyms, 2002. En ligne ; consulté le 2020-02-02.

[2] JORF n°0129 du 6 juin 2010 page 10453 texte n°42. https://www.legifrance.gouv.fr/jo_pdf.do?id=JORFTEXT000022309303, 2010. En ligne ; consulté le 2020-02-02.

[3] Iozone benchmark. http://www.iozone.org, 2016. En ligne ; consulté le 2020-02-02.

[4] Powerspy2. https://www.alciom.com/nos-metiers/produits/powerspy2/, 2016. En ligne ; consulté le 2020-02-02.

[5] Accelio based network block device. https://github.com/accelio/NBDX, 2019. En ligne ; consulté le 2020-02-02.

[6] Amadeus. http://www.amadeus.com, 2019. En ligne ; consulté le 2020-02-02.

[7] Amazon ec2. https://aws.amazon.com/ec2, 2019. En ligne ; consulté le 2020-02-02.

[8] Amazon web services. https://aws.amazon.com/, 2019. En ligne ; consulté le 2020-02-02.

[9] Google app engine. https://cloud.google.com/appengine, 2019. En ligne ; consulté le 2020-02-02.

[10] Grid 5000. https://www.grid5000.fr, 2019. En ligne ; consulté le 2020-02-02.

[11] Infinband write bandwidth tool. https://community.mellanox.com/s/article/ib-write-bw, 2019. En ligne ; consulté le 2020-02-02.

[12] Infinband write latency tool. https://community.mellanox.com/s/article/ib-write_lat, 2019. En ligne ; consulté le 2020-02-02.

[13] Linux ate my ram ! https://www.linuxatemyram.com, 2019. En ligne ; consulté le 2020-02-02.

[14] Performance tests (perftest) package for ofed. https://community.mellanox.com/s/article/perftest-package, 2019. En ligne ; consulté le 2020-02-02.

[15] Titanium home page. http://titanium.cs.berkeley.edu, 2019. En ligne ; consulté le 2020-02-02.

[16] Anderson, D. *USB system architecture*. Addison-Wesley Professional, 1997.

[17] Anderson, E. A., and Neefe, J. M. *An exploration of network RAM*. University of California, Berkeley, Computer Science Division, 1998.

[18] Anderson, T. E., Owicki, S. S., Saxe, J. B., and Thacker, C. P. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems (TOCS) 11*, 4 (1993), 319–352.

[19] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using KSM. In *Proceedings of the linux symposium* (2009), Citeseer, pp. 19–28.

[20] Arulraj, L., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Improving virtualized storage performance with sky. *SIGPLAN Not.* (2017).

[21] Barak, A., Guday, S., and Wheeler, R. G. *The MOSIX distributed operating system : load balancing for UNIX*, vol. 13. Springer, 1993.

[22] Barak, D. Verbs programming tutorial.

[23] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 164–177.

[24] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), vol. 41, p. 46.

[25] Beloglazov, A., Piraghaj, S. F., Alrokayan, M., and Buyya, R. Deploying openstack on centos using the KVM hypervisor and GlusterFS distributed file system. *University of Melbourne* (2012).

[26] Bienia, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[27] Black, D. L., and Sleator, D. D. *Competitive algorithms for replication and migration problems*. Carnegie-Mellon University. Department of Computer Science, 1989.

[28] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W.-K. Myrinet : A gigabit-per-second local area network. *IEEE micro 15*, 1 (1995), 29–36.

[29] Boettiger, C. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review 49*, 1 (2015), 71–79.

[30] Buckland, P. A. Numa system with redundant main memory architecture, Aug. 31 2004. US Patent 6,785,783.

[31] Buyun, D., Pei, F., Xiao, F., Bin, L., and Zhihong, Z. Design and implementation of hdfs over infiniband with rdma. In *International Conference on Wired/Wireless Internet Communication* (2013), Springer, pp. 95–114.

[32] Callaghan, B., Lingutla-Raj, T., Chiu, A., Staubach, P., and Asad, O. NFS over RDMA. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence : experience, lessons, implications* (2003), pp. 196–208.

[33] Cappelletti, P. Non volatile memory evolution and revolution. In *2015 IEEE International Electron Devices Meeting (IEDM)* (2015), IEEE, pp. 10–1.

[34] Carey, M., Franklin, M., and Livny, M. Global memory management in client-server dbms architecture. In *Proceedings of International Conference on Very Large Databases* (1992), pp. 596–609.

[35] Carlson, B., El-Ghazawi, T., Numrich, B., and Yelick, K. Programming in the partitioned global address space model. *Tutorial at Supercomputing* (2003), 44.

[36] Carter, J. B., Bennett, J. K., and Zwaenepoel, W. Implementation and performance of munin. *ACM SIGOPS Operating Systems Review 25*, 5 (1991), 152–164.

[37] Chadalapaka, M., Shah, H., Elzur, U., Thaler, P., and Ko, M. A study of iscsi extensions for rdma (iser). In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence : experience, lessons, implications* (2003), pp. 209–219.

[38] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[39] Chen, H., Luo, Y., Wang, X., Zhang, B., Sun, Y., and Wang, Z. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology* (2008).

[40] Codenamed, A. V. Pacifica. *Technology : Secure Virtual Machine Architecture Reference Manual* (2005), 1–124.

[41] Coffman, E. G., and Denning, P. J. *Operating systems theory*, vol. 973. prentice-Hall Englewood Cliffs, NJ, 1973.

[42] Comer, D. E., and Griffioen, J. A new design for distributed systems : The remote memory model.

[43] Consortium, U., et al. Upc language specifications v1.2. Tech. rep., Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), 2005.

[44] Corbato, F. J. A paging experiment with the multics system. Tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[45] Cornwell, M. Anatomy of a solid-state drive. *Commun. ACM 55*, 12 (2012), 59–63.

[46] Couvert, P. High speed io processor for nvme over fabric (NVMeoF). *Flash Memory Summit* (2016).

[47] Cox, A., Dwarkadas, S., Keleher, P., and Zwaenepoel, W. Treadmarks : Distributed shared memory on standard workstations and operating systems.

[48] Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., Von Eicken, T., and Yelick, K. Parallel programming in split-c. In *Supercomputing'93 : Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (1993), IEEE, pp. 262–273.

[49] D'amato, A., Nagar, R. Y., Nishanov, G., Das, R., and Maeso, G. Cluster shared volumes, Nov. 23 2010. US Patent 7,840,730.

[50] Danilak, R. Method and system of improving disk access time by compression, June 6 2006. US Patent 7,058,769.

[51] Das Kamhout, Greg Bunce, C. P. Implementing on-demand services inside the intel it private cloud.

[52] Di, S., Kondo, D., and Cappello, F. Characterizing cloud applications on a google data center. In *2013 42nd International Conference on Parallel Processing* (2013), IEEE, pp. 468–473.

[53] Ding, X., Jiang, S., Chen, F., Davis, K., and Zhang, X. Diskseen : Exploiting disk layout and access history to enhance i/o prefetch. In *USENIX Annual Technical Conference* (2007), vol. 7, pp. 261–274.

[54] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. FaRM : Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 401–414.

[55] Dwarkadas, S., Hardavellas, N., Kontothanassis, L., Nikhil, R., and Stets, R. Cashmere-VLM : Remote memory paging for software distributed shared memory. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999* (1999), IEEE, pp. 153–159.

[56] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3 : Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 371–384.

[57] Feeley, M. J., Morgan, W. E., Pighin, E., Karlin, A. R., Levy, H. M., and Thekkath, C. A. Implementing global memory management in a workstation cluster. In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (1995), pp. 201–212.

[58] Felten, E. W., and Zahorjan, J. *Issues in the implementation of a remote memory paging system.* University of Washington, Department of Computer Science and Engineering, 1991.

[59] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the clouds : A study of emerging scale-out

workloads on modern hardware. *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012).

[60] Fischer, W., Wallmeier, E., Worster, T., Davis, S. P., and Hayter, A. Data communications using ATM : architectures, protocols, and resource management. *IEEE Communications Magazine 32*, 8 (1994), 24–33.

[61] Fitzpatrick, B. Distributed caching with memcached. *Linux journal 2004*, 124 (2004), 5.

[62] Flouris, M. D., and Markatos, E. P. The network ramdisk : Using remote memory on heterogeneous nows. *Cluster Computing 2*, 4 (1999), 281–293.

[63] Friedman, M. B. Windows nt page replacement policies. In *Int. CMG Conference* (1999), vol. 99, pp. 234–244.

[64] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), pp. 29–43.

[65] Goldberg, R. P. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems* (1973), ACM, pp. 74–112.

[66] Goldberg, R. P. Survey of virtual machine research. *Computer 7*, 6 (1974), 34–45.

[67] Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., and Shin, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017), pp. 649–667.

[68] Guthrie, F., Lowe, S., and Coleman, K. *VMware vSphere design*. John Wiley & Sons, 2013.

[69] Hanke, S., Ottmann, T., and Soisalon-Soininen, E. Relaxed balanced red-black trees. In *Italian Conference on Algorithms and Complexity* (1997), Springer, pp. 193–204.

[70] Hertel, C. R. *Implementing CIFS : The Common Internet File System*. Prentice Hall Professional, 2004.

[71] Hilfinger, P. N., Bonachea, D., Datta, K., Gay, D., Graham, S., Kamil, A., Liblit, B., Pike, G., Su, J., and Yelick, K. Titanium language reference manual. *Computer Science* (2006).

[72] Hoyte, D. Vmtouch - the virtual memory toucher. https://hoytech.com/vmtouch, 2018. En ligne ; consulté le 2020-02-02.

[73] Hu, Y., and Yang, Q. DCD—disk caching disk : A new approach for boosting i/o performance. In *ACM SIGARCH Computer Architecture News* (1996), vol. 24, ACM, pp. 169–178.

[74] Intel. Intel Non-Volatile Memory 3d XPoint. https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo, 2017. En ligne ; consulté le 2020-02-02.

[75] Intel, Microsoft, T. P. H.-P. Advanced configuration and powerinterface (ACPI) specification - revision 6.3. https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf, 2019.

[76] International Organization, S. A. International organization : Serial ata revision 3.0. *Gold Revision, June 2* (2009).

[77] Islam, N. S., Rahman, M. W., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C., and Panda, D. K. High performance rdma-based design of hdfs over infiniband. In *SC'12 : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE, pp. 1–12.

[78] Islam, N. S., Wasi-ur Rahman, M., Lu, X., and Panda, D. K. High performance design for HDFS with byte-addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), pp. 1–14.

[79] Jacob, B., Larson, P., Leitao, B., and Da Silva, S. Systemtap : instrumenting the linux kernel for analyzing performance and functional problems. *IBM Redbook 116* (2008).

[80] Jadeja, Y., and Modi, K. Cloud computing-concepts, architecture and challenges. In *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)* (2012), IEEE, pp. 877–880.

[81] Jaggar, D., and Seal, D. *ARM architecture reference manual*. Prentice Hall, 1996.

[82] Jiang, S., Chen, F., and Zhang, X. Clock-pro : An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track* (2005), pp. 323–336.

[83] Jiang, S., and Zhang, X. Lirs : an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review 30*, 1 (2002), 31–42.

[84] Johnson, K. L., Kaashoek, M. F., and Wallach, D. A. Crl : High-performance all-software distributed shared memory. In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (1995), pp. 213–226.

[85] Johnson, T., Shasha, D., et al. 2q : a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases* (1994), Citeseer, pp. 439–450.

[86] Josep, A. D., Katz, R., Konwinski, A., Gunho, L., Patterson, D., and Rabkin, A. A view of cloud computing. *Communications of the ACM 53*, 4 (2010).

[87] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), pp. 295–306.

[88] Kembel, R. W. *Fibre Channel A Comprehensive Introduction*. Northwest Learning Assoc, 2009.

[89] Kim, J. M., Choi, J., Kim, J., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* (2000), USENIX Association, p. 9.

[90] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. Kvm : the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07* (2007).

[91] Kolivas, C. Kernbench v0.50. http://ck.kolivas.org/apps/kernbench/kernbench-0.50, 2012. En ligne ; consulté le 2020-02-02.

[92] Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C.-S. On the existence of a spectrum of policies that subsumes

the least recently used (lru) and least frequently used (lfu) policies. In *SIGMETRICS* (1999), vol. 99, Citeseer, pp. 1–4.

[93] Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. Lrfu : A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 12 (2001), 1352–1361.

[94] Li, K. *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis.

[95] Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS) 7*, 4 (1989), 321–359.

[96] Liang, S., Noronha, R., and Panda, D. K. Swapping to remote memory over infiniband : An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing* (2005), IEEE, pp. 1–10.

[97] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news 37*, 3 (2009), 267–278.

[98] Lu, Y., Shu, J., Chen, Y., and Li, T. Octopus : an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), pp. 773–785.

[99] Markatos, E. P., and Dramitinos, G. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference* (1996), pp. 177–190.

[100] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem : current status and future plans. In *Proceedings of the Linux symposium* (2007), vol. 2, Citeseer, pp. 21–33.

[101] Megiddo, N., and Modha, D. S. Arc : A self-tuning, low overhead replacement cache. In *FAST* (2003), vol. 3, pp. 115–130.

[102] Mell, P., Grance, T., et al. The NIST definition of cloud computing.

[103] Metcalfe, R. M., and Boggs, D. R. Ethernet : Distributed packet switching for local computer networks. *Communications of the ACM 19*, 7 (1976), 395–404.

[104] Microsoft. exFAT file system specification. https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification, 2019. En ligne ; consulté le 2020-02-02.

[105] Mikhailov, D. NTFS file system. *Internet Article* (2000).

[106] Mitchell, C., Geng, Y., and Li, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 103–114.

[107] Morin, C., Lottiaux, R., Vallée, G., Gallard, P., Utard, G., Badrinath, R., and Rilling, L. Kerrighed : a single system image cluster operating system for high performance computing. In *European Conference on Parallel Processing* (2003), Springer, pp. 1291–1294.

[108] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 291–305.

[109] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 385–398.

[110] Nitu, V., Teabe, B., Fopa, L., Tchana, A., and Hagimont, D. Stopgap : Elastic vms to enhance server consolidation. *Software : Practice and Experience 47*, 11 (2017), 1501–1519.

[111] Nitzberg, B., and Lo, V. Distributed shared memory : A survey of issues and algorithms. *Computer 24*, 8 (1991), 52–60.

[112] Numrich, R. W., and Reid, J. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum* (1998), vol. 17, ACM New York, NY, USA, pp. 1–31.

[113] NVM Express Inc. NVM Express Base Specification revison 1.4.

[114] O'neil, E. J., O'neil, P. E., and Weikum, G. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record 22*, 2 (1993), 297–306.

[115] Oracle Corporation. Virtualbox user manual. *Oracle Corporation.-2004.-C 357* (2011).

[116] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., et al. The case for ramclouds : scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[117] Palit, T., Shen, Y., and Ferdman, M. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (April 2016), pp. 122–132.

[118] Park, J., Wang, Q., Li, J., Lai, C., Zhu, T., and Pu, C. Performance interference of memory thrashing in virtualized cloud environments : A study of consolidated n-tier applications. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)* (June 2016).

[119] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data* (1988), pp. 109–116.

[120] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D. NFS version 3 : Design and implementation. In *USENIX Summer* (1994), Boston, MA, pp. 137–152.

[121] PCI-SIG. Single root i/o virtualization and sharing specification revision 1.1.

[122] Pfister, G. F. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O 42* (2001), 617–632.

[123] Pinkerton, J. The case for rdma. *RDMA Consortium, May 29* (2002), 27.

[124] Popek, G. J., and Kline, C. S. The pdp-11 virtual machine architecture : A case study. In *ACM SIGOPS Operating Systems Review* (1975), vol. 9, ACM, pp. 97–105.

[125] Postel, J., and Reynolds, J. File transfer protocol.

[126] Power, R., and Li, J. Piccolo : Building fast, distributed programs with partitioned tables. In *OSDI* (2010), vol. 10, pp. 1–14.

[127] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News* (2007).

[128] Rimal, B. P., Choi, E., and Lumb, I. A taxonomy and survey of cloud computing systems. In *2009 Fifth International Joint Conference on INC, IMS and IDC* (Aug 2009), pp. 44–51.

[129] Robinson, J. T., and Devarakonda, M. V. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (1990), pp. 134–142.

[130] Rodeh, O., Bacik, J., and Mason, C. Btrfs : The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 1–32.

[131] Rosen, R. Linux containers and the future cloud. *Linux J 240*, 4 (2014), 86–95.

[132] Sacks, D. Demystifying storage networking das, san, nas, nas gateways, fibre channel, and iscsi. *IBM storage networking* (2001), 3–11.

[133] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M., and Zeidner, E. RFC3720 : Internet small computer systems interface (iSCSI), 2004.

[134] Schmidt, F. *The SCSI bus and IDE interface : protocols, applications and programming*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[135] Schmuck, F. B., and Haskin, R. L. GPFS : A shared-disk file system for large computing clusters. In *FAST* (2002), vol. 2.

[136] Schopp, J. H., Fraser, K., and Silbermann, M. J. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium* (2006), vol. 2, pp. 313–319.

[137] Seifert, R. *Gigabit Ethernet : technology and applications for high-speed LANs*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[138] Shah, K., Mitra, A., and Matani, D. An o (1) algorithm for implementing the lfu cache eviction scheme. *no 1* (2010), 1–8.

[139] Shan, Y., Tsai, S.-Y., and Zhang, Y. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), pp. 323–337.

[140] Sharma, P., and Kulkarni, P. Singleton : System-wide page deduplication in virtual environments. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (2012), HPDC '12, ACM.

[141] Sharma, P., Kulkarni, P., and Shenoy, P. Per-VM page cache partitioning for cloud computing platforms. In *2016 8th International Conference on Communication Systems and Networks (COMSNETS)* (2016).

[142] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)* (2010), Ieee, pp. 1–10.

[143] Smaragdakis, Y., Kaplan, S., and Wilson, P. Eelru : simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review 27*, 1 (1999), 122–133.

[144] Stefanovici, I., Thereska, E., O'Shea, G., Schroeder, B., Ballani, H., Karagiannis, T., Rowstron, A., and Talpey, T. Software-defined caching : Managing caches in multi-tenant data centers. In *ACM Symposium on Cloud Computing (SOCC) 2015* (2015), ACM.

[145] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001), pp. 1–14.

[146] Talpey, T., and Kamer, G. High performance file serving with SMB3 and RDMA via SMB direct. In *Storage Developers Conference* (2012).

[147] Tantisiriroj, W., Son, S. W., Patil, S., Lang, S. J., Gibson, G., and Ross, R. B. On the duality of data-intensive file system design : reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–12.

[148] Tarasov, V. Filebench. https://github.com/filebench/filebench, 2019. En ligne ; consulté le 2020-02-02.

[149] Technologies, M. Infiniband roadmap. https://www.infinibandta.org/infiniband-roadmap, 2019. En ligne ; consulté le 2020-02-02.

[150] Technologies, M. Mellanox connectx-2. http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX-2_EN_Cards.pdf, 2019. En ligne ; consulté le 2020-02-02.

[151] Troppens, U., Erkens, R., and Müller, W. *Storage networks explained : basics and application of fibre channel SAN, NAS, iSCSI and InfiniBand*. John Wiley & Sons, 2005.

[152] Tweedie, S. Ext3, journaling filesystem. In *Ottawa Linux Symposium* (2000), pp. 24–29.

[153] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[154] Velte, A., and Velte, T. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[155] Waldspurger, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[156] Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T., and Huang, I. Understanding lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep* (2009).

[157] Wang, R. Y., and Anderson, T. E. xFS : A wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III* (1993), IEEE, pp. 71–78.

[158] Watson, G. R., Sottile, M. J., Minnich, R. G., Choi, S.-E., and Hertdriks, E. Pink : A 1024-node single-system image linux cluster. In *Proceedings. Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region, 2004.* (2004), IEEE, pp. 454–461.

[159] Wei, J., Cheng, Q., Penty, R. V., White, I. H., and Cunningham, D. G. 400 gigabit ethernet using advanced modulation formats : performance, complexity, and power dissipation. *IEEE Communications Magazine 53*, 2 (2015), 182–189.

[160] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. Ceph : A scalable, high-performance distributed file

system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

[161] Wong, T. M., and Wilkes, J. My cache or yours ? making storage more exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference* (2002), ATEC '02, USENIX Association.

[162] Worthington, B. L., Ganger, G. R., and Patt, Y. N. Scheduling algorithms for modern disk drives. In *ACM SIGMETRICS Performance Evaluation Review* (1994), vol. 22, ACM, pp. 241–251.

[163] Wu, J., Wyckoff, P., and Panda, D. Pvfs over infiniband : Design and performance evaluation. In *2003 International Conference on Parallel Processing, 2003. Proceedings.* (2003), IEEE, pp. 125–132.

[164] Yang, J., Izraelevitz, J., and Swanson, S. Orion : a distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019), pp. 221–234.

[165] Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S. L., Hargrove, P., Hilfinger, P., Husbands, P., et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation* (2007), pp. 24–32.

[166] Zhang, X., and Jiang, S. Interferenceremoval : removing interference of disk access for mpi programs through data replication. In *Proceedings of the 24th ACM International Conference on Supercomputing* (2010), ACM, pp. 223–232.

[167] Zhang, Y., and Swanson, S. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)* (2015), IEEE, pp. 1–10.

[168] Zhou, Y., Chen, Z., and Li, K. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems 15*, 6 (2004), 505–519.

[169] Zhou, Y., Philbin, J., and Li, K. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track* (2001), pp. 91–104.

[170] Zongjie, Z. M. Z. Analysis of FAT32 file system [j]. *Computer & Digital Engineering 1* (2005), 014.