Caches collaboratifs noyau adaptés aux environnements virtualisés

Maxime Lorrillere

▶ **To cite this version:**

Maxime Lorrillere. Caches collaboratifs noyau adaptés aux environnements virtualisés. Système d'exploitation [cs.OS]. Université Pierre et Marie Curie - Paris VI, 2016. Français. <NNT : 2016PA066036>. <tel-01273367v2>

HAL Id: tel-01273367

https://hal.inria.fr/tel-01273367v2

Submitted on 8 Sep 2016

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 DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Maxime Lorrillere

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Caches collaboratifs noyau adaptés aux environnements virtualisés

---

Soutenue le 4 février 2016 devant le jury composé de :

Mme Christine Morin	*Rapporteure*	Directrice de recherche, Inria
M. Vivien Quéma	*Rapporteur*	Professeur, Grenoble INP
M. Édouard Bugnion	*Examinateur*	Professeur, École polytechnique fédérale de Lausanne
M. Gilles Muller	*Examinateur*	Directeur de recherche, Inria
M. Étienne Rivière	*Examinateur*	Maître assistant, université de Neuchâtel
M. Sébastien Monnet	*Directeur de thèse*	Maître de conférences (HDR), université Pierre et Marie Curie
M. Pierre Sens	*Directeur de thèse*	Professeur, université Pierre et Marie Curie
M. Julien Sopena	*Encadrant*	Maître de conférences, université Pierre et Marie Curie

Remerciements

Je souhaite remercier les nombreuses personnes qui m'ont apporté leur soutien, encouragements et commentaires tout au long de ma thèse.

Un grand merci à Christine Morin et Vivien Quéma pour avoir accepté de consacrer du temps pour rapporter cette thèse. Je remercie également Édouard Bugnion et Étienne Rivière pour l'attention qu'ils ont porté à mon travail en acceptant de participer à mon jury.

Merci à Gilles Muller, à la fois pour avoir accepté de participer à mon jury, mais aussi pour m'avoir encouragé tout au long de ma thèse et permis d'aller vers de nouveaux horizons.

Je n'aurai pas pu réaliser cette thèse sans les précieux conseils de mes encadrants, Julien, Sébastien et Pierre. Nos discussions et débats ont fini par porter leurs fruits ! Merci de m'avoir permis de m'investir dans ce sujet passionnant. Cette thèse a aussi permis d'animer de nombreux stages qui nous ont fait avancer et découvrir de nouveaux problèmes. Un grand merci à Joel, Ludovic, et Guillaume, j'ai particulièrement apprécié de travailler avec vous.

Je souhaite remercier l'ensemble de l'équipe REGAL qui m'a chaleureusement accueilli. Merci Swan, Olivier, Franck, Marc, Julia, et Pierre-Evariste pour les discussions les discussions que nous avons pu avoir autour d'un café, ou pour les commentaires et relectures que vous m'avez apporté. Je n'oublie pas de remercier Gaël pour tous les commentaires et remarques qui nous ont permis d'améliorer notre travail. Continue d'apporter ta joie et ta bonne humeur à tes thésards (et les autres !), ils en ont tous besoin ! Merci également aux collègues que l'on croise tout au long de la journée dans ton bureau, la salle café !

Merci à tous les ingénieurs, thésards et stagiaires passés par notre (grand) bureau. Il y a toujours régné une atmosphère bon enfant, idéale pour s'échanger des idées et travailler. Merci à Véronique, Sergey, Thomas, Lokesh, Florian, Maxime, Brice, Rudyar, Gauthier, Damien ! Et je n'oublie pas, bien sûr, le bureau voisin. Merci Jonathan, Karine, Hamza, et les nouveaux, Marjorie et Denis !

Bien sûr, c'est difficile de s'investir autant dans son travail sans s'aérer de temps en temps ! Un grand merci aux copains, Olivier, Antoine, Élodie et Renaud, pour tous les moments que nous avons passé ensemble, au bar, au ski, à l'escalade et au tennis !

Enfin, un grand merci à ma famille de m'avoir permis de faire de si longues études et de m'avoir soutenu pendant ces années passées à Paris.

Laure, je n'aurai jamais pu terminer ma thèse sans ton soutien inébranlable et tes encouragements et ton sourire. Merci d'avoir toujours cru en moi, je ne pourrais jamais oublier !

Table des matières

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Organisation du manuscrit . . . 4
1.3 Publications . . . 5

2 Caches répartis dans les systèmes de stockage distribués **7**
2.1 Historique des caches . . . 8
2.2 Propriétés des caches . . . 9
2.2.1 Placement et localisation des données . . . 10
2.2.2 Algorithmes de remplacement de cache . . . 10
2.2.3 Inclusivité des caches . . . 11
2.2.4 Politiques d'écritures . . . 12
2.3 Caches répartis . . . 12
2.3.1 Caches distants . . . 12
2.3.2 Caches collaboratifs . . . 13
2.4 Propriétés des caches répartis . . . 18
2.4.1 Accès concurrents et cohérence des données . . . 19
2.4.2 Tolérance aux fautes . . . 20
2.4.3 Sécurité . . . 21
2.4.4 Généricité . . . 21
2.5 Application aux environnements virtualisés . . . 23
2.5.1 Différents types d'hyperviseurs . . . 23
2.5.2 Techniques de virtualisation . . . 24
2.5.3 Mécanismes de gestion de la mémoire pour les machines virtuelles . . . 25
2.5.4 Réduction de l'empreinte mémoire . . . 27
2.5.5 Caches collaboratifs dans les machines virtuelles : approches existantes . . . 28
2.6 Discussion . . . 29

3 Contexte et prérequis techniques 31
3.1 Principes des systèmes de fichiers 32
3.2 Les différents caches 34
3.2.1 *dentry cache* 34
3.2.2 Cache d'*inode* 35
3.2.3 Page cache 36
3.2.4 Buffer cache 38
3.2.5 Unification du *page cache* et du *buffer cache* 39
3.3 Gestion de la mémoire 39
3.3.1 Récupération de la mémoire et activation des pages 40
3.3.2 *Shadow page cache* : estimation de la taille du *working set* . . . 41
3.3.3 Détection des accès au cache avec l'API *cleancache* 42
3.4 Conclusion 43

4 Puma : mutualisation de la mémoire inutilisée des machines virtuelles 45
4.1 Principes de Puma 46
4.2 Architecture générale de Puma 47
4.2.1 Positionnement dans la pile système 47
4.2.2 Gestion de la mémoire 49
4.2.3 Gestion des pannes et du temps de réponse 51
4.2.4 Gestion des accès séquentiels 51
4.2.5 Stratégies de cache 52
4.3 Implémentation 53
4.3.1 Stockage des métadonnées 54
4.3.2 Stockage des pages 54
4.3.3 Cohérence des caches 55
4.4 Conclusion 55

5 Évaluation de Puma 57
5.1 Protocole expérimental 58
5.1.1 Benchmarks 58
5.1.2 Paramètres 61
5.1.3 Métriques 62
5.2 Lectures aléatoires : de l'intérêt des stratégies de cache 64
5.3 Accès séquentiels : de l'intérêt du filtre 66
5.4 Performances des benchmarks applicatifs 67
5.4.1 BLAST : une charge partiellement séquentielle 67
5.4.2 Postmark : expérimentations en présence d'écritures 68
5.4.3 Bases de données 70
5.5 Analyse de la sensibilité au temps de réponse 71
5.6 Comparaison avec un cache sur un SSD 73

5.7 Étude de cas : le cloud privé 74
5.8 Conclusion 76

6 Gestion dynamique du cache entre machines virtuelles 77
6.1 Méthodologie 78
6.2 Récupération efficace du cache pour de la mémoire anonyme 79
6.3 Récupération efficace de la mémoire pour du cache 79
6.3.1 Accélérer la récupération : limitation de PUMA à la liste inactive 79
6.3.2 Prêter toute sa mémoire : rééquilibrage des listes LRUs 81
6.4 Gestion de la concurrence des accès au cache 82
6.4.1 Détection d'une activité via le *shadow page cache* 84
6.4.2 Détection d'une activité via l'augmentation de la pression mémoire 85
6.4.3 Combinaison des deux approches 86
6.5 Discussion 86

7 Évaluation de l'automatisation de PUMA 89
7.1 Plateforme expérimentale 89
7.1.1 Benchmarks 90
7.2 Récupération automatique du cache pour de la mémoire anonyme . . . 91
7.2.1 Analyse de la récupération de la mémoire 91
7.2.2 Efficacité de la récupération : vitesse des allocations 93
7.2.3 Étude de cas : la consolidation sur un serveur unique 94
7.3 Surcout du mécanisme de partage automatique 97
7.4 Analyse du seuil de désactivation des pages 100
7.5 Conclusion 103

8 Conclusions et perspectives 105
8.1 Synthèse 105
8.2 Perspectives 107

Bibliographie 111

Table des figures

1.1 Architectures *N-tier* préconisées par Amazon 2

2.1 Hiérarchie des mémoires 9
2.2 Caches répartis 13
2.3 Algorithme de cache collaboratif *Direct Client Cooperation* 14
2.4 Algorithme de cache collaboratif *Greedy Forwarding* 15
2.5 Algorithme de cache collaboratif *Centrally Coordinated Caching* 16
2.6 Algorithme *Hint-based collaborative caching* 18
2.7 Les différentes couches d'un système d'exploitation 22
2.8 Les différents types d'hyperviseurs 23
2.9 Virtualisation au niveau système d'exploitation (isolateur) 24
2.10 Memory ballooning 27

3.1 Interactions entre les différentes abstractions du VFS 32
3.2 Caches disques du *VFS* 34
3.3 *dentry cache* 35
3.4 Virtualisation et *page cache* du système d'exploitation 36
3.5 Organisation du page cache 37
3.6 Blocs contigus sur le disque 38
3.7 Blocs non-contigus sur le disque 39
3.8 Page cache unifié [18] 40
3.9 Fonctionnement des listes *LRU* du noyau Linux 41
3.10 *Shadow page cache* 42

4.1 Architecture de Puma : opération `get` 48
4.2 Architecture de Puma : opération `put` 48

5.1 Performance des accès aléatoires 65
5.2 Nombre de `get`/`put` pour le benchmark de lectures aléatoires 65
5.3 Performance des accès séquentiels 66

5.4 Accélération obtenue avec BLAST 68
5.5 Accélération obtenue avec Postmark 68
5.6 Accès au périphérique de stockage avec Postmark 69
5.7 Accélération obtenue avec TPC-C 70
5.8 Accélération obtenue avec TPC-H 71
5.9 PUMA sans contrôle du temps de réponse 72
5.10 PUMA avec contrôle du temps de réponse 72
5.11 Accélération obtenue avec les différents benchmarks et des machines virtuelles (VMs) hébergées sur des nœuds différents 75

6.1 Stockage des pages uniquement dans la liste inactive 80
6.2 Désactivation des pages actives 82
6.3 Comportement de PUMA en présence de workloads concurrents plusieurs nœuds 83
6.4 Désactivation de PUMA lors d'un hit dans les *shadow pages* 84
6.5 Désactivation de PUMA lors d'une augmentation de la pression mémoire 85
6.6 Désactivation de PUMA en combinant les deux approches 86

7.1 Performances de Filebench en nombre d'opérations par seconde dans la VM_1 91
7.2 Utilisation de la mémoire avec une gestion dynamique PUMA et des besoins variables 92
7.3 Latence des allocations de mémoire 93
7.4 Mémoire disponible sur la VM Git avec PUMA et du ballooning automatique 96
7.5 Surcout du mécanisme de partage automatique 98
7.6 Variation du seuil de désactivation des pages actives – lectures aléatoires 101
7.7 Variation du nombre de pages actives sur la VM_2 102
7.8 Variation du nombre de pages distantes hébergées par la VM_2 102

Liste des tableaux

5.1 Synthèse des benchmarks utilisés pour l'évaluation de PUMA 59
5.2 Comparaison de PUMA avec un cache SSD 74

7.1 Temps de réponse du ballooning automatique. 95

Chapitre 1

Introduction

Sommaire

1.1 **Contributions** 3
1.2 **Organisation du manuscrit** 4
1.3 **Publications** 5

La virtualisation est aujourd'hui omniprésente dans les systèmes distribués à grande échelle. À la base du « cloud computing », on la retrouve aussi dans nombre de clusters privés. L'utilisation de machines virtuelles (VM, de l'anglais *Virtual Machine*) permet en effet aux fournisseurs d'infrastructure d'introduire dynamicité (déploiement à la volée, migration, ...) et sécurité (isolation des applications, sauvegarde du contexte, ...). Cette flexibilité est à la base du succès des IaaS (*Infrastructure-As-A-Service*), secteur dont la croissance est la plus rapide dans le marché du cloud computing estimé à 9 milliards dollars en 2013 [35].

Des applications de plus en plus complexes ont grandement bénéficé de cette virtualisation massive. Ainsi, on observe le déploiement d'architectures *N-tier* toujours plus complexes, et l'on parle aujourd'hui de *constellations* de VMs. Chacune des VMs se voit alors confier un rôle précis et coopère avec les autres pour assurer un service global, comme le montre la figure 1.1 qui résume les différents schémas préconisés par Amazon [4]. Outre leur nombre, on assiste à hyper-spécialisation des VMs. Les machines finissent donc par héberger un ensemble hétérogène de services (serveurs web, bases de données, répartiteurs de charge, routeurs,...) qui engendrent autant de charges différentes. Certaines VMs orientées données génèreront énormément d'entrées/sorties (E/S) tandis que d'autres, plus orientées traitements, solliciteront plus le processeur et la mémoire.

De nombreux travaux ont étudié le placement et l'ordonnancement des VMs sur les

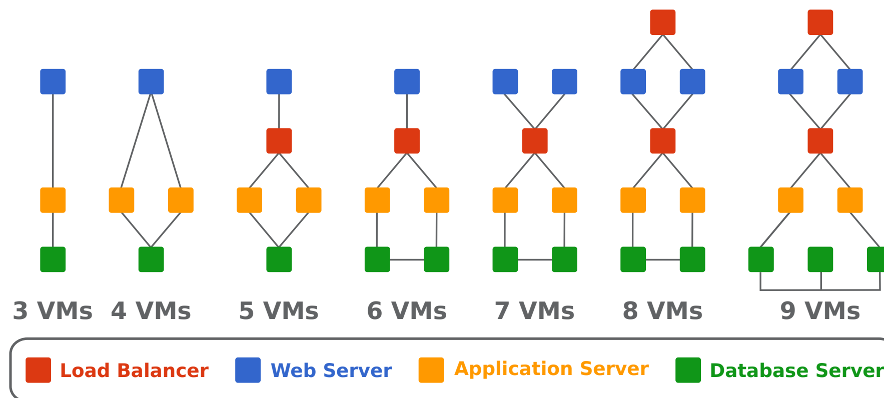


FIGURE 1.1 – Architectures *N-tier* préconisées par Amazon

machines [2, 45, 103]. Cependant, l'utilisation des ressources allouées pose un nouveau défi. En effet, la virtualisation massive génère une fragmentation des ressources physiques disponibles, en particulier de la mémoire. Dans les environnements virtualisés les machines physiques sont partitionnées : leur mémoire est divisée et répartie entre les VMs. Comme il est difficile d'anticiper les besoins en mémoire d'une application, la quantité de mémoire allouée aux VMs est généralement surdimensionnée.

Le problème vient du fait que la mémoire des VMs est généralement définie de manière *statique* : elle est souvent choisie parmi un ensemble de configurations prédéfinies offertes par le fournisseur de cloud [16]. Cette fragmentation prédéfinie de la mémoire des VMs entraîne une sous-utilisation de la mémoire avec une implication forte sur les performances. En effet, certaines VMs peuvent voir leurs performances impactées parce qu'elles n'ont pas assez de mémoire, alors que d'autres pourraient utiliser moins de mémoire sans dégrader leurs performances.

Couplée à l'hyper-spécialisation des VMs, la fragmentation de leur mémoire impacte particulièrement les performances des applications orientées données. En effet, les systèmes d'exploitation mettent généralement leur mémoire inutilisée au profit de ces applications en l'utilisant comme cache : les données déjà lues sont conservées en mémoire pour accélérer leurs futurs accès. Historiquement, différents types d'applications partageaient la même mémoire, une application effectuant beaucoup d'E/S pouvait donc profiter de la faible activité d'une application orientée traitement. Cependant, l'hyper-spécialisation des VMs a changé la donne, les applications sont isolées et il n'est plus possible de faire bénéficier de la mémoire inutilisée d'une VM à une autre.

S'il existe des solutions permettant de réattribuer dynamiquement la quantité de mémoire allouée aux VMs (*memory ballooning* [10, 56, 104]), elles sont difficilement automatisables sans pénaliser les performances des applications [5, 85]. De plus, ces approches sont limitées aux VMs hébergées sur la même machine physique, elles ne permettent donc pas de réutiliser la mémoire inutilisée d'une VM s'exécutant sur une

autre machine. Dans ce contexte, fournir la possibilité de mutualiser la mémoire inutilisée à l'échelle d'un centre de données doit permettre d'améliorer les performances et le niveau de consolidation.

Cette thèse propose donc une nouvelle approche pour mutualiser la mémoire inutilisée entre VMs d'un centre de données. Elle s'inscrit dans le cadre du projet Nu@ge [1], l'un des cinq projets retenus pour réaliser un cloud français indépendant dans le cadre du grand emprunt.

1.1 Contributions

Les principales contributions de cette thèse sont d'une part l'élaboration d'un système de cache réparti transparent et efficace permettant d'exploiter la mémoire inutilisée des VMs, et d'autre part la réalisation de mécanismes permettant de détecter dynamiquement l'(in)activité d'une VM en termes de besoins en mémoire, de façon à automatiser le niveau de contribution au cache réparti des VMs impliquées.

Caches répartis pour les environnements virtualisés. Compte tenu de la diversité des applications déployées dans les clouds, il est important de disposer d'un mécanisme *transparent*, le moins *intrusif* possible, pour mutualiser la mémoire inutilisée des VMs. En effet, les solutions « classiques » reposant sur l'utilisation d'APIs dédiées [34] ou sur des systèmes de fichiers spécifiques [6, 7, 28], impliquent des contraintes fortes pour les applications. Ainsi, la première partie de cette thèse a proposé de nouveaux mécanismes permettant de concevoir un cache réparti le plus *transparent* possible.

Ces mécanismes ont permis la réalisation de Puma [2], un cache réparti mutualisant la mémoire inutilisée des machines virtuelles pour améliorer les performances des applications qui effectuent beaucoup d'E/S [TSI-2015, SYSTOR'15, ComPAS'2014]. L'originalité de Puma réside dans son approche système, au cœur du noyau Linux, ce qui lui permet à la fois d'être transparent pour les applications et de ne pas dépendre d'un système de fichiers spécifique.

Dynamicité des caches répartis. Une des difficultés posée par les caches répartis est de savoir à quel moment un nœud devient inactif, pour que sa mémoire inutilisée puisse être prêtée au cache réparti. De même, lorsqu'un nœud redevient actif, il doit être capable de récupérer efficacement la mémoire qu'il a prêté pour ne pas être pénalisé. Il convient alors de traiter le cas où une application alloue de la mémoire (`malloc`) et le cas où une application effectue beaucoup d'E/S (*page cache*).

1. `http://www.nuage-france.fr`
2. Pour *Pooling Unused memory in virtual MAchines*

- ***Allocations de mémoire.*** PUMA étant directement intégré au sein du *page cache* du noyau Linux, ses données sont naturellement expulsées du cache lors d'une allocation de mémoire. Cependant, il est important d'évincer en priorité les données *hébergées* par un nœud PUMA, plutôt que ses données *locales*. Ainsi, nous proposons des modifications aux stratégies de gestion du cache du noyau Linux afin de les rendre moins prioritaires.
- ***Entrées/sorties.*** Nous avons instrumenté le noyau Linux et proposé dans [Compas'2015] des métriques qui permettent de définir le niveau d'activité « cache » du système. Ainsi, lorsque le niveau d'activité « cache » détecté est élevé le système refuse de prêter de la mémoire au cache réparti. Inversement, lorsque le niveau d'activité « cache » diminue, le système accepte de prêter de la mémoire *inutilisée*. Implémentés dans PUMA, ces métriques permettent d'automatiser la distribution de la mémoire entre les nœuds du cache réparti en fonction de leur activité.

1.2 Organisation du manuscrit

Cette thèse est organisée de la façon suivante

- **Chapitre 2 : Caches répartis dans les systèmes de stockage distribués**. Ce chapitre présente un état de l'art des caches et des caches répartis dans le contexte des systèmes de stockage distribués et des environnements virtualisés.
- **Chapitre 3 : Contexte et prérequis techniques**. Nous exposons dans ce chapitre les prérequis techniques nécessaires à la compréhension de cette thèse. Nous nous concentrons sur le fonctionnement des systèmes de fichiers du noyau Linux, ainsi qu'aux différents caches dont ils dépendent. Nous abordons également la gestion de la mémoire du noyau Linux, en particulier dans le contexte des caches des systèmes de fichiers.
- **Chapitre 4 : PUMA : mutualisation de la mémoire inutilisée des machines virtuelles**. Ce chapitre présente notre première contribution de cette thèse, PUMA. Nous y décrivons son architecture et ses mécanismes qui nous ont permis de le rendre efficace et compatible avec les applications et les systèmes de fichiers existants. *Ces travaux ont été publiés dans [TSI-2015, SYSTOR'15, ComPAS'2014].*
- **Chapitre 5 : Évaluation de PUMA**. Ce chapitre présente une évaluation exhaustive des performances de PUMA et des différents mécanismes que nous y avons introduit. Nous décrivons plusieurs analyses de sensibilité (types d'accès, temps de réponse), et nous montrons à travers plusieurs benchmarks et applications (BLAST, Postmark, TPC-C et TPC-H) que PUMA permet d'améliorer les performances des applications qui effectuent beaucoup d'E/S. *Cette évaluation fait partie de nos travaux publiés dans [TSI-2015, SYSTOR'15, ComPAS'2014].*
- **Chapitre 6 : Gestion dynamique du cache entre machines virtuelles**. La

seconde contribution de cette thèse est présentée dans ce chapitre, et s'intéresse à l'automatisation du partage de la mémoire dans les caches répartis. Nous étudions en particulier différents mécanismes permettant à PUMA d'ajuster dynamiquement la quantité mémoire qu'un nœud offre au cache réparti en fonction de ses besoins. *Ces travaux ont été publiés dans [Compas'2015].*
- **Chapitre 7 : Évaluation de l'automatisation de PUMA**. Ce chapitre décrit une évaluation de l'automatisation de PUMA et montre les limites de notre approche.
- **Chapitre 8 : Conclusions et perspectives**. Ce chapitre conclut ce manuscrit et présente les perspectives ouvertes par cette thèse.

1.3 Publications

[SYSTOR'15] Maxime LORRILLERE, Julien SOPENA, Sébastien MONNET et Pierre SENS. « Puma : Pooling Unused Memory in Virtual Machines for I/O Intensive Applications ». In : *Proceedings of the 8th ACM International Systems and Storage Conference.* SYSTOR '15. Haifa, Israel : ACM, 2015, p. 1–11.

[TSI-2015] Maxime LORRILLERE, Julien SOPENA, Sébastien MONNET et Pierre SENS. « Conception et évaluation d'un système de cache réparti adapté aux environnements virtualisés ». In : *Technique et Science Informatiques* 34.1–2 (2015), p. 101–123.

[Compas'2015] Maxime LORRILLERE, Joel POUDROUX, Julien SOPENA et Sébastien MONNET. « Gestion dynamique du cache entre machines virtuelles ». In : *Conférence d'Informatique en Parallélisme, Architecture et Système.* Compas'2015. Lille, France, juin 2015, p. 1–10.

[ComPAS'2014] Maxime LORRILLERE, Julien SOPENA, Sébastien MONNET et Pierre SENS. « PUMA : Un cache distant pour mutualiser la mémoire inutilisée des machines virtuelles ». Français. In : *ComPAS'2014.* Neuchâtel, Suisse, avr. 2014, 1–12, ***Meilleur article*** *du track système.*

[ComPAS'2013] Maxime LORRILLERE, Julien SOPENA, Sébastien MONNET et Pierre SENS. « Vers un cache réparti adapté au cloud computing ». Français. In : *ComPAS'2013.* Grenoble, France, jan. 2013, p. 1–12.

Chapitre 2

Caches répartis dans les systèmes de stockage distribués

Sommaire

2.1 Historique des caches **8**
2.2 Propriétés des caches **9**
2.2.1 Placement et localisation des données 10
2.2.2 Algorithmes de remplacement de cache 10
2.2.3 Inclusivité des caches 11
2.2.4 Politiques d'écritures 12
2.3 Caches répartis **12**
2.3.1 Caches distants 12
2.3.2 Caches collaboratifs 13
2.4 Propriétés des caches répartis **18**
2.4.1 Accès concurrents et cohérence des données 19
2.4.2 Tolérance aux fautes 20
2.4.3 Sécurité 21
2.4.4 Généricité 21
2.5 Application aux environnements virtualisés **23**
2.5.1 Différents types d'hyperviseurs 23
2.5.2 Techniques de virtualisation 24
2.5.3 Mécanismes de gestion de la mémoire pour les machines virtuelles . 25
2.5.4 Réduction de l'empreinte mémoire 27
2.5.5 Caches collaboratifs dans les machines virtuelles : approches existantes 28
2.6 Discussion **29**

La vitesse d'accès aux données a toujours été l'un des points critiques des systèmes informatiques. Une des approches classiques pour améliorer la performance des accès aux données repose sur l'utilisation des caches, que l'on peut retrouver à la fois au niveau matériel et logiciel. Au niveau matériel on trouve par exemple les caches des processeurs (caches L1, L2, L3, TLB, etc.) ou des périphériques de stockage (mémoire vive intégrée aux contrôleurs). Au niveau logiciel, les caches peuvent être gérés par le système d'exploitation comme le cache d'*i-node* ou le *page cache* (présentés dans le chapitre 3), ou par les applications comme les caches des navigateurs web. Cette thèse s'intéresse à des caches plus complexes qui peuvent être *répartis* entre plusieurs entités, telles que des processus, des machines virtuelles (VMs) ou des machines physiques.

Ce chapitre est une étude bibliographique sur les caches répartis et leur utilisation. Dans un premier temps, nous présentons un historique des caches répartis (section 2.1), puis la section 2.2 détaille les propriétés importantes des caches. Nous analysons dans la section 2.3 les différents types de caches répartis et leurs algorithmes, puis nous présentons dans la section 2.4 les propriétés des caches répartis et les différentes solutions qui ont été proposées pour les respecter. Enfin, la section 2.5 s'intéresse aux particularités des environnements virtualisés et identifie les approches existantes en matière de caches répartis adaptés à la virtualisation et au cloud computing, puis la section 2.6 conclut ce chapitre.

2.1 Historique des caches

Un cache est un espace de stockage utilisé pour conserver des données à forte localité spatiale ou temporelle. La mémoire utilisée pour le cache est plus rapide que l'espace de stockage principal, ce qui permet au processeur de travailler avec des copies des données dans le cache plutôt que dans l'espace de stockage principal, plus lent.

John von Neumann évoque déjà en 1947 l'utilisation d'une hiérarchie de mémoires pour palier le cout [1] extrême des mémoires rapides [22]. L'idée d'utiliser la hiérarchie des mémoires en exploitant la localité temporelle a été introduite en 1965 par Maurice Wilkes sous le nom de *slave memory* [108], et consistait à ajouter au processeur un espace de stockage rapide qui lui permet de conserver « sous la main » les dernières instructions qu'il a exécutées. Le terme *cache* est apparu quelques années plus tard, en 1968, avec le calculateur *IBM System/360-85* [64] qui fut l'un des premiers calculateurs

1. Le lecteur pourrait, comme mes encadrants, être surpris de l'absence d'accent sur certaines lettres. En effet, cette thèse utilise certaines rectifications orthographiques du français portées par la réforme de 1990 [62]. En particulier, vous ne trouverez pas d'accent circonflexe sur la lette « u » lorsque celui-ci n'est pas nécessaire, notamment pour les mots *cout* et *surcout*.

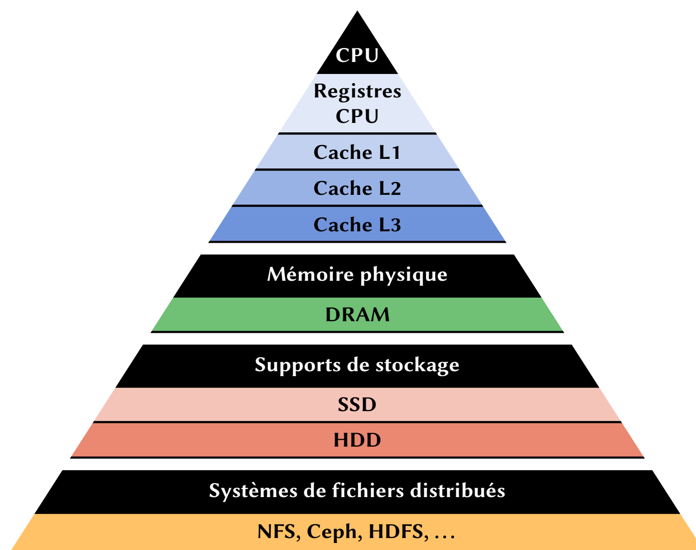


FIGURE 2.1 – Hiérarchie des mémoires

à intégrer un cache au sein de son processeur. De nos jours, les processeurs intègrent plusieurs niveaux de cache de capacité croissante.

Si les premiers caches ciblaient principalement l'accélération des accès à la mémoire, le principe a rapidement été étendu aux périphériques, et notamment aux disques. Parallèlement, la généralisation des systèmes de fichiers distribués a conduit à l'introduction de caches pour diminuer la charge des serveurs de fichiers et mieux passer à l'échelle : en ajoutant aux machines clientes un cache (local) pour les données du système de fichiers distribué, on réduit les accès aux serveurs. Les caches pour les environnements distribués apparaissent en particulier avec le système de fichiers distribué *Andrew File System* [75, 88]. De nos jours, on utilise des caches locaux sur des SSDs (*Solid State Drive*) pour accélérer les accès aux disques durs classiques (HDDs) [8, 54, 79]. Ces différents niveaux dans la hiérarchie des mémoires sont représentés dans la figure 2.1.

Ce n'est qu'au début des années 90 que seront introduits les caches répartis entre plusieurs nœuds. Ainsi, partant du constat que la latence d'accès à un autre nœud dans un réseau local (*LAN*) est plus faible que la latence d'accès à un disque dur, Douglas Comer et James Griffioen proposent d'utiliser la mémoire en réseau fournie par des *serveurs* pour étendre la mémoire locale de nœuds *clients* [26].

2.2 Propriétés des caches

Les caches sont des systèmes complexes caractérisés par un ensemble de propriétés, nous présentons les principales dans cette section. Nous discutons notamment de la

localisation des données dans le cache, des différentes stratégies de caches, de la gestion des écritures ainsi que du choix des données à évincer du cache.

2.2.1 Placement et localisation des données

La méthode de placement d'une donnée dans un cache est une caractéristique importante qui peut avoir une grande influence sur les performances du cache. Une méthode simple consiste à appliquer une fonction de hachage à l'identifiant de la donnée pour obtenir le numéro d'une « case » du cache dans laquelle sera stockée la donnée. Par exemple, dans le contexte des caches CPU, on parlera de cache à *correspondance directe*, où chaque ligne de mémoire ne peut aller que dans une seule ligne de cache prédéfinie. Cette méthode permet une localisation rapide d'une donnée, au prix d'un *taux de miss* plus élevée : deux données *chaudes* peuvent être placées dans la même case et s'évincer mutuellement. Lorsqu'une donnée peut aller dans plusieurs emplacements de caches distincts, on parlera de cache *associatif* à N voies pour N emplacements distincts [42]. Le choix du nombre de voies est alors un compromis entre le cout de la recherche et le taux de miss : plus le nombre de voies est élevé, plus le taux de miss sera faible, mais la localisation des données sera plus couteuse. Ce type d'approche peut également s'appliquer pour des caches locaux sur disque de taille fixe, où la fonction de hachage permet de calculer l'emplacement de la donnée [101]. Avec des caches répartis, une table de hachage distribuée peut être utilisée pour déterminer le ou les nœud(s) du cache réparti où sera envoyée la donnée [31, 33, 49].

Lorsque l'on souhaite pouvoir placer les données n'importe où dans le cache, il est nécessaire de conserver certaines *métadonnées* pour permettre de retrouver ces données. Ces métadonnées décrivent la correspondance entre l'identifiant de la donnée et le numéro de la case du cache dans laquelle elle a été placée. Dans le cas d'un cache réparti, ces métadonnées peuvent être *locales*, *distribuées* ou confiées à un nœud responsable de centraliser les accès aux métadonnées (*manager*) [28, 31, 87].

2.2.2 Algorithmes de remplacement de cache

Lorsque le cache est plein, il est nécessaire de choisir une donnée *victime* à *évincer* du cache pour prendre sa place. Le choix de cette donnée est confié à l'algorithme de remplacement de cache. Idéalement, celui-ci devrait choisir la donnée qui ne sera plus utilisée pendant la plus grande période de temps [13]. Comme il n'est pas possible de prévoir l'avenir, la plupart des algorithmes de remplacement de cache reposent sur le principe de *localité* [32] :

- localité *spatiale* : si une donnée est référencée à un instant particulier, il y a de fortes chances que les données voisines soient accédées prochainement ;

— localité *temporelle* : une donnée référencée à un instant particulier a de fortes chances d'être accédée à nouveau dans un futur proche.

De nombreux algorithmes de remplacement de cache ont été imaginés pour résoudre différents problèmes (types d'accès, taille de cache, latence, etc.). Les plus classiques comme LRU (*Least Recently Used*) ou FIFO (*First-in, First-out*) disposent de multiples variantes permettant d'améliorer de taux de succès (*hit*) du cache dans certains cas ou de limiter le cout d'exécution de l'algorithme. Par exemple, *Second-chance* est une variante de FIFO dans laquelle la donnée la plus ancienne est maintenue dans la liste FIFO si elle a été utilisée pendant sa durée de vie dans la liste. *CLOCK* est une amélioration de cet algorithme qui repose sur une liste circulaire, ce qui permet d'éviter d'avoir à déplacer des données dans la liste.

Les algorithmes de remplacement de cache peuvent également être combinés pour former de nouveaux algorithmes. Par exemple, 2Q (*Two Queue*) [51] repose sur une liste FIFO et une liste LRU. Lors de l'entrée d'une donnée dans le cache, elle est placée dans la liste FIFO. Si elle est référencée alors qu'elle est déjà dans cette liste, elle est considérée comme étant *chaude* et est *promue* dans la LRU. L'algorithme utilisé par le noyau Linux, que nous détaillons dans la section 3.3, ressemble à l'algorithme 2Q.

2.2.3 Inclusivité des caches

Dans les systèmes de cache à plusieurs niveaux, que l'on retrouve par exemple dans les processeurs (caches L1, L2) ou dans les caches pour les systèmes de fichiers distribués (cache en mémoire et cache sur le disque *local*) se pose la question de la redondance des données présentes dans les caches. Par exemple, dans certains processeurs les données présentes dans le cache L1 sont également présentes dans le cache L2, c'est ce qu'on appelle un cache *strictement inclusif* : chaque niveau de cache ***inférieur*** (cache L1) est un sous-ensemble du cache de niveau ***supérieur*** (cache L2). À l'inverse, un cache *exclusif* est un cache dans lequel une donnée ne peut se trouver que dans un seul niveau de cache.

L'avantage d'un cache exclusif est qu'il offre plus d'espace de cache qu'un cache inclusif puisque la taille du cache est la somme de tous les niveaux de cache. Les caches strictement inclusifs ont en revanche des opérations beaucoup plus simples, par exemple l'accès à une donnée présente dans un cache de niveau supérieur a simplement besoin d'être copiée dans le cache de niveau inférieur, alors que dans le cas d'un cache exclusif il est également nécessaire de supprimer la version du niveau supérieur.

Une stratégie hybride, appelée *non-inclusive* [114], consiste à relâcher la contrainte d'inclusivité de la stratégie strictement inclusive. Ainsi, une donnée peut être évincée d'un cache de plus haut niveau pour faire de la place à une nouvelle donnée, tout en conservant les copies présentes dans les caches de niveau inférieur, qui peuvent être

évincées s'il est nécessaire de récupérer de la place. Avec cette stratégie, la quantité de cache réellement disponible est proche d'une stratégie exclusive [50].

2.2.4 Politiques d'écritures

Lorsque des données d'un cache sont modifiées, elles doivent être écrites sur le périphérique de stockage d'origine. Le moment où ces données sont écrites dépend de la *politique d'écriture*. Le choix peut avoir un impact déterminant sur les performances des écritures. On distingue principalement deux politiques d'écritures, *write-through* et *write-back* [42].

Write-through. Les données sont écrites de façon synchrone sur le périphérique de stockage d'origine, ce qui a l'avantage de garantir que la version présente dans le cache est identique à celle présente sur le périphérique d'origine. Cependant, cette politique a le défaut d'être lente, puisque les accès en écriture au cache vont à la vitesse du périphérique d'origine.

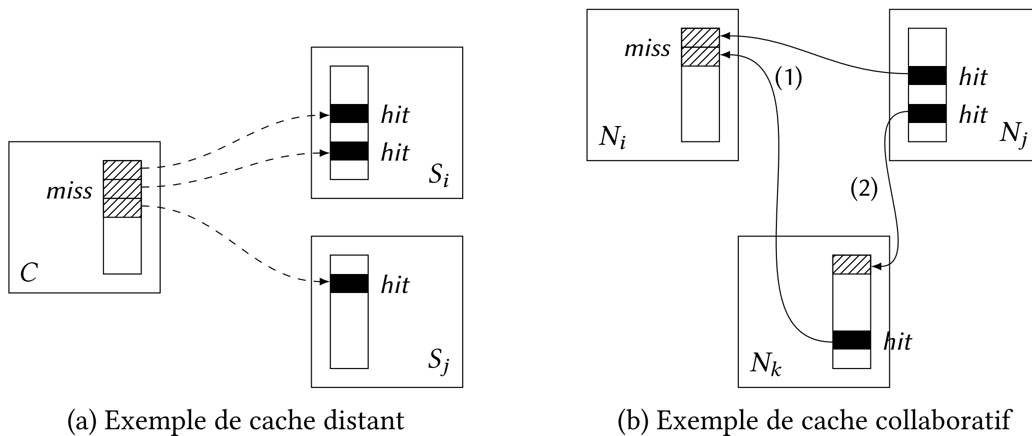
Write-back. L'écriture des données modifiées sur le périphérique d'origine est différée, jusqu'à l'expiration d'une certaine durée ou jusqu'à ce que la donnée soit évincée du cache. L'avantage de cette politique est sa performance, puisque les écritures vont à la vitesse du cache. De plus, les données modifiées plusieurs fois ou modifiées puis supprimées nécessitent beaucoup moins d'entrées/sorties (E/S). Cependant, en cas de panne il est possible de perdre les données qui n'ont pas été écrites.

2.3 Caches répartis

Cette section présente les deux grandes catégories de caches répartis : les *caches distants* (section 2.3.1) pour lesquels des nœuds agissent exclusivement comme *client* ou *serveur* de cache, et les *caches collaboratifs* (section 2.3.2) où le statut des nœuds peut varier en fonction des besoins et de leur charge.

2.3.1 Caches distants

Introduits par Comer et Griffionen [26], les caches distants sont des caches répartis dans lesquels on distingue les serveurs de cache, qui fournissent de la mémoire comme support de stockage pour le cache, et les clients, qui utilisent ce cache et qui ne communiquent pas entre eux. Un exemple d'une telle architecture est présenté dans la figure



(a) Exemple de cache distant (b) Exemple de cache collaboratif

FIGURE 2.2 – Caches répartis

2.2a, dans laquelle le client C accède à des données du cache des serveurs S_i et S_j pour résoudre des *miss* dans son cache local.

Ce type d'architecture a été utilisé par exemple par les auteurs de *CaaS* [39], un système de cache élastique développé pour proposer du cache à la demande à des serveurs de calcul dans le cadre du cloud. Dans cette architecture, des serveurs de mémoire mettent à disposition des serveurs de calcul des *chunks* de mémoire qui sont accédés en réseau pour faire du cache. Ce type d'architecture est utilisé indirectement par les systèmes de fichiers répartis client/serveur comme NFS [90], qui reposent en partie sur le cache des serveurs.

2.3.2 Caches collaboratifs

Les caches collaboratifs sont une évolution des caches distants dans laquelle des nœuds peuvent interagir pour accéder aux données du cache qui sont réparties dans leurs mémoires. Leff et al. [60, 61] font la différence entre les architectures de cache réparti *asymétriques* et *symétriques* : les premières sont des architectures client/serveur, équivalentes des caches distants présentés dans la section précédente, et dans les secondes tous les nœuds peuvent potentiellement répondre à un *miss* d'un autre nœud.

La figure 2.2b illustre le fonctionnement d'un cache collaboratif composé de 3 nœuds (N_i, N_j et N_k). Dans cet exemple, le nœud N_i obtient des blocs de données directement depuis les nœuds N_j et N_k (1). Dans le même temps, le nœud N_k obtient un bloc depuis le nœud N_j (2).

Les travaux de Dahlin et al. [31] ont permis d'identifier plusieurs algorithmes de caches collaboratifs, dont l'un de ces algorithmes, *N-Chance Forwarding*, a été amélioré par la suite par Sarkar et Hartman [87] afin de le rendre plus décentralisé. Dans cette section, nous proposons d'analyser les différents algorithmes qui ont été proposés par

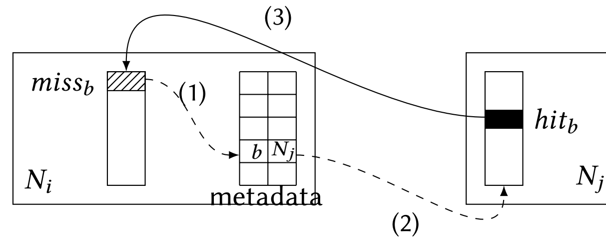


FIGURE 2.3 – Algorithme de cache collaboratif *Direct Client Cooperation*

ces travaux.

2.3.2.1 *Direct Client Cooperation*

L'idée de l'algorithme *Direct Client Cooperation* [31] est d'utiliser directement la mémoire d'autres nœuds comme espace de stockage supplémentaire pour le cache, et ressemble donc à une variante des caches distants décrits dans la section 2.3.1 où les nœuds n'ont pas de fonction dédiée.

Dans cet algorithme, un nœud transfère un bloc qu'il doit évincer de son cache à un autre nœud, inactif, ce qui lui permet d'avoir à disposition un cache « privé » réparti sur d'autres nœuds et qui agit comme une extension de son cache local. Ces autres nœuds peuvent évincer de leur cache les blocs qu'ils hébergent lorsqu'ils ont besoin de place. Lorsqu'un nœud ne possède pas un bloc dans son cache local ou privé, il le demande directement au serveur de stockage et ne profite donc pas des potentiels autres nœuds qui ont déjà mis cette donnée dans leur cache (local ou privé). Cet algorithme est illustré par la figure 2.3. Dans cet exemple, le nœud N_i fait un *miss* sur le bloc b et vérifie dans ces métadonnées (1) si b a été précédemment placé dans un autre nœud. Il contacte ensuite le nœud N_j pour obtenir une copie du bloc b qu'il héberge dans le cache privé de N_i (2), puis le bloc b est transmis à N_i (3).

L'inconvénient de cet algorithme est qu'il ne permet pas aux nœuds de mettre en commun les blocs qu'ils pourraient partager (déduplication) puisque les espaces de cache *privés* sont gérés indépendamment les uns des autres. Ainsi, un nœud qui n'a jamais lu un bloc ira nécessairement le lire depuis le disque, même si un autre nœud possède ce bloc dans son cache.

2.3.2.2 *Greedy Forwarding*

Cet algorithme, également proposé par Dahlin et al. [12, 31], permet de traiter le cache de tous les nœuds du système comme une ressource globale qui peut être utili-

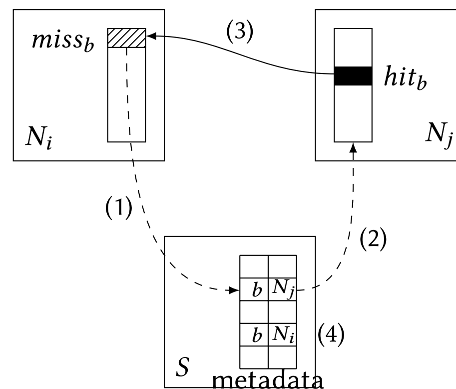


FIGURE 2.4 – Algorithme de cache collaboratif *Greedy Forwarding*

sée pour satisfaire un *miss*. Initialement, l'algorithme fonctionne exactement comme s'il n'y avait pas de cache collaboratif : lorsqu'un nœud ne trouve pas un bloc dans son cache local, il le demande au serveur de stockage. Le serveur de stockage possède des métadonnées sur l'emplacement des blocs au sein du cache collaboratif. Ainsi, lorsqu'il reçoit une demande de lecture, il la transmet au nœud qui possède le bloc correspondant. Dans le cas où aucun nœud n'ayant le bloc n'existe, le serveur le lit depuis le disque, le transmet au nœud qui en a fait la demande puis met à jour ses métadonnées.

La figure 2.4 présente un exemple d'exécution de l'algorithme *Greedy Forwarding*. Le nœud N_i ne dispose pas du bloc b dans son cache et envoie une requête au serveur S (1). Le serveur S transmet la requête au nœud N_j parce qu'il sait qu'il a précédemment donné b à N_j (2), puis N_j transmet le bloc b à N_i (3). À la fin de l'opération, le serveur sait que N_i et N_j possèdent le bloc b (4).

Un avantage de cet algorithme est qu'il n'est pas contraignant pour les nœuds : l'absence de contrôle du serveur leur permet de gérer leur cache local comme ils l'entendent tout en permettant aux autres d'en profiter. En revanche, l'absence de coordination directe entre eux augmente le nombre de copies des blocs et diminue donc l'espace utile disponible, ce qui réduit le taux de *hit* global.

2.3.2.3 *Centrally Coordinated Caching*

Dahlin et al. [31] ont également proposé l'algorithme *Centrally Coordinated Caching* afin d'ajouter de la coordination à l'algorithme *Greedy Forwarding*. L'idée est de réduire le nombre de copies des blocs et donc d'augmenter le taux de *hit* global. Dans cet algorithme, chaque nœud partitionne son cache en une partition gérée localement par le nœud et en une partition gérée globalement par le serveur de stockage. Si un nœud ne trouve pas un bloc dans son cache local, il envoie une requête au serveur de stockage. Le serveur de stockage peut alors (*i*) transmettre directement le bloc, s'il le possède dans son cache local ; (*ii*) transmettre la requête au nœud qui possède le bloc dans

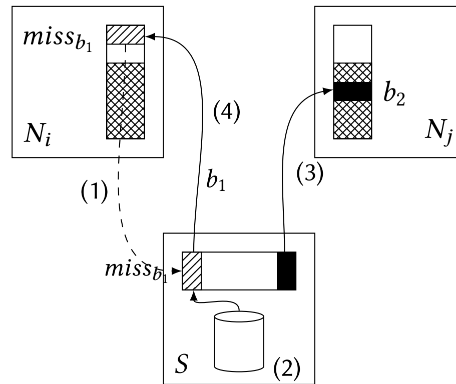


FIGURE 2.5 – Algorithme de cache collaboratif *Centrally Coordinated Caching*

sa partition globale, s'il existe ; (*iii*) lire le bloc depuis le disque, le stocker dans son cache local et le transmettre au nœud. Lorsque le serveur doit évincer des blocs de son cache, il les envoie dans la partition de mémoire gérée globalement d'un nœud du cache collaboratif.

Un exemple d'exécution de cet algorithme est présenté dans la figure 2.5. Dans cet exemple, le nœud N_i fait un *miss* sur le bloc b_1, il le demande donc au serveur S (1). Le serveur S récupère le bloc b_1 depuis le disque et l'ajoute dans son cache local (2) mais doit d'abord évincer le bloc b_2 pour libérer de la place, il décide donc de le placer dans la partition globale du nœud N_j (3). Il peut ensuite transférer le bloc b_1 au nœud N_i (4). Un futur accès d'un nœud au bloc b_2 se déroulera de la même façon qu'avec l'algorithme *Greedy Forwarding*.

Un des avantages de cet algorithme est qu'il permet d'obtenir un taux de *hit* important dans le cache global puisque sa gestion par le serveur permet d'éviter les doublons. Cependant, le taux de *hit* local des nœuds s'en retrouve pénalisé puisque ceux-ci ne maîtrisent plus le choix des données puisque c'est le serveur qui le leur impose. Ils sont donc amputés d'une partie de leur cache local.

2.3.2.4 *Hash-Distributed Caching*

Cet algorithme est une variante de l'algorithme *Centrally Coordinated Caching*, également proposée par Dahlin et al. [31], dans laquelle les métadonnées permettant de localiser les nœuds repose sur une table de hachage distribuée. Ainsi, lors d'un *miss*, la table de hachage distribuée indique le nœud où le bloc doit être présent. Si le bloc n'est pas dans le cache global, le nœud qui héberge la partition transfère la requête au serveur qui pourra alors lire le bloc depuis le disque et le transmettre au nœud qui l'a demandé. *Hash-Distributed Caching* offre des performances équivalentes à l'algorithme *Centrally Coordinated Caching* mais a l'avantage de ne pas charger le serveur pour localiser les blocs dans le cache global.

2.3.2.5 *N-Chance Forwarding*

L'algorithme *N-Chance* [31] est similaire à l'algorithme *Greedy Forwarding* et permet d'améliorer le taux de *hits* dans le cache global en évinçant prioritairement les blocs qui disposent de plusieurs copies dans le cache global, tout en permettant aux nœuds actifs d'utiliser efficacement leur cache local. Cet algorithme est notamment utilisé par le système de fichiers xFS [6].

Dans cet algorithme, avant d'expulser un bloc du cache global on lui laisse n chances. Pour cela, lorsqu'un nœud doit remplacer un bloc de son cache, il vérifie si sa copie du bloc est un *singleton*, c'est-à-dire l'unique copie de ce bloc présente dans un cache. Si c'est un singleton, il place un compteur de recirculation à n, transfère le bloc à un autre nœud choisi aléatoirement pour qu'il soit placé dans son cache et envoie un message au serveur pour l'informer du nouvel emplacement du bloc. Lorsqu'un bloc avec un compteur de recirculation doit être remplacé, le compteur est décrémenté et le bloc est transféré à un autre nœud choisi aléatoirement, sauf lorsque le compteur de recirculation devient nul : le bloc est alors supprimé du cache. Lorsqu'un nœud accède à un singleton, le compteur de recirculation est réinitialisé. Cet algorithme est une généralisation de l'algorithme *Greedy Forwarding* où $n = 0$.

2.3.2.6 *Hint-based collaborative caching*

Sarkar et Hartman ont proposé une amélioration de l'algorithme *N-Chance Forwarding* [87] permettant de réduire la dépendance aux serveurs de métadonnées (*managers*) : pour localiser les blocs, les nœuds utilisent des informations locales, possiblement incorrectes, plutôt que les informations globales correctes fournies par les *managers*.

Par exemple, lors de l'ouverture d'un fichier par un nœud, le manager lui envoie les informations de localisation de chaque bloc du fichier dans les caches d'autres nœuds. Lorsque ce nœud effectue un *miss* dans son cache local sur un bloc, il consulte les informations de localisation pour savoir quel nœud possède le bloc dans son cache, évitant ainsi le manager.

Le nœud qui reçoit la requête transmet le bloc au demandeur s'il le possède, sinon il consulte ses propres informations de localisation et transfère la requête à un autre nœud qui a précédemment possédé ce bloc. Si tel n'est pas le cas, la requête est retransmise récursivement jusqu'à ce qu'un nœud ait le bloc dans son cache ou qu'il n'ait pas d'information de localisation pour ce bloc.

Lorsqu'un nœud doit remplacer un bloc de son cache, il doit choisir le bloc qui est le moins utile au cache global. Pour cela, on distingue les *master-copies* qui sont les copies de bloc récupérées depuis le serveur des autres copies qui sont supprimées prioritairement. Chaque nœud associe un âge au bloc le plus ancien de sa *LRU* locale

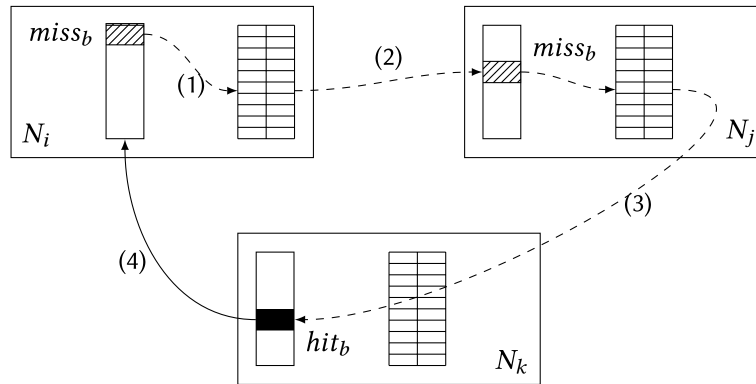


FIGURE 2.6 – Algorithme *Hint-based collaborative caching*

et maintient une structure contenant les âges des plus anciens blocs des autres nœuds. Lorsqu'un nœud doit remplacer une *master-copy* de son cache, il la transfère au nœud qui possède le bloc le plus ancien.

Un exemple de *hit* dans le cache global est présenté dans la figure 2.6 : le nœud N_i ne possède pas le bloc b dans son cache, il consulte alors sa table de *hints* pour savoir quel est le dernier nœud connu qui possède b (1). Il transmet ensuite sa requête au nœud N_j, qui ne possède plus le bloc b (2). La table de *hints* de N_j lui indique que N_k possède le bloc b, il transmet donc la requête à N_k (3) qui possède effectivement le bloc b. N_k peut ensuite envoyer directement à N_i le bloc b (4).

Dans des travaux visant à comparer l'intérêt d'un cache collaboratif à un cache orienté contenu spécialisé, Cuenca-Acuna et NGuyen [30] ont proposé des améliorations à l'algorithme proposé par Sarkar et Hartman. En effet, lors des mesures de performances, les auteurs ont remarqué que les *master-copies* peuvent se retrouver supprimées du cache collaboratif alors que des copies sont encore présentes : les *master-copies* des blocs les moins utilisés sont remplacées et les copies (non-*master*) de blocs utilisés sont conservées. Les auteurs ont alors proposé de modifier la politique de remplacement de la LRU *globale* en remplaçant prioritairement les blocs qui ne sont pas des *master-copies*, même si des copies non-*master* sont plus récentes, ce qui permet de conserver les *master-copies* dans le cache global. À terme, cette politique de remplacement fait que le cache global ne contient que des *master-copies*.

2.4 Propriétés des caches répartis

Cette section présente les propriétés des caches répartis. Nous commençons par discuter de la cohérence des données dans la section 2.4.1, puis de la tolérance aux fautes dans la section 2.4.2. Nous présenterons ensuite les problèmes d'intégrité et de confidentialité que posent les caches répartis dans la section 2.4.3, puis nous terminons

cette section par la généricité des caches répartis (section 2.4.4).

2.4.1 Accès concurrents et cohérence des données

L'introduction d'un niveau de cache supplémentaire partagé entre plusieurs nœuds d'un système distribué complexifie la gestion des accès concurrents aux données pour assurer leur cohérence. Si les accès en écritures sont de plus en plus présents, en particulier dans les clouds [15], plusieurs études basées sur des analyses de traces d'entrée/sortie de systèmes de fichiers distribués ont montré que les accès partagés étaient peu nombreux et que les accès partagés en écriture le sont encore moins [9, 63]. Ainsi, la gestion de la cohérence des données des systèmes de stockage répartis reposent le plus souvent sur des approches centralisées.

Une approche souvent utilisée est celle utilisant des *baux* de Gray et Cheriton [37], qui permettent de gérer ces accès concurrents. Le principe est de confier la gestion des droits d'accès à un fichier (lecture, écriture, ...) à un serveur, qui peut ensuite donner un bail à un client. Lorsque le bail expire, le client doit obtenir un nouveau bail auprès du serveur pour effectuer ses opérations. Lorsqu'un client demande des droits contradictoires avec un ou plusieurs baux en cours (conflits lecteur(s)/écrivain), le serveur peut révoquer les baux pour donner un bail au nouveau client.

Beaucoup de systèmes de fichiers distribués tels que *NFS* [81, 90], *Ceph* [106] et *Sprite* [78] reposent sur ce genre de mécanismes pour gérer et optimiser les accès partagés. Par exemple, dans *Ceph*, un serveur de métadonnées concède des droits à des clients pour mettre en cache les blocs d'un fichier. Lorsqu'un client génère un conflit lecteur/écrivain, le serveur de métadonnées révoque les droits de mise en cache des clients et les force à effectuer des entrées/sorties synchrones, ce qui permet au système de fichiers distribué de respecter le modèle de cohérence spécifié par *POSIX* [48] :

> “ Chaque lecture (*read*) consécutive à une écriture (*write*), même avec des processus différents, doit renvoyer les données modifiées par l'écriture. ”

Plusieurs travaux visant à améliorer les performances de *NFS* ont également dû tenir compte de la cohérence des données. C'est le cas par exemple de *NFS-cc* [112] et de *NFS-CD* [11]. *NFS-cc* est une version du système de fichiers distribué *NFS* modifiée par l'ajout d'un cache collaboratif. Pour que *NFS-cc* conserve le modèle de cohérence d'origine de *NFS* (*close-to-open* [81, 90]), le serveur retransmet les requêtes de blocs avec les attributs des fichiers au client qui possède le bloc afin que celui-ci puisse vérifier si sa copie du bloc est à jour. De façon similaire, *NFS-CD* permet de « déléguer » le rôle de serveur pour un fichier donné à un client à qui seront transférées toutes les requêtes concernant ce fichier, ce qui permet de diminuer la charge du serveur. Dans cette architecture, appelée *cluster-delegation*, le client qui agit comme serveur a pour rôle de sérialiser les conflits, exactement comme un serveur *NFS* classique.

Une autre approche permettant de contourner le problème des accès partagés, proposée par Cortes et al. [28], consiste à ne pas utiliser de cache local. Les blocs de données passent directement du cache global (coopératif) vers l'espace d'adressage de l'utilisateur, évitant ainsi une recopie dans le cache local : puisqu'il n'y a plus de réplication dans le cache local, il n'y a plus de problème de cohérence (au sens *POSIX*). Il est cependant nécessaire de ne pas avoir de réplication dans le cache global.

2.4.2 Tolérance aux fautes

Comme tout système distribué, les caches répartis peuvent subir des pannes qu'il est important de tolérer pour éviter des pertes de données et garantir des propriétés de vivacité. Nous nous intéressons plus particulièrement aux pannes franches, dans lesquelles un nœud s'arrête de fonctionner et ne répond plus, et éventuellement aux pannes franches avec reprise, dans lesquelles un nœud qui s'est arrêté de fonctionner recommence à participer au cache réparti. Le cas des fautes byzantines est indirectement et partiellement géré via les mécanismes de contrôle d'intégrité qui peuvent être mis en place.

Caches en lecture seule. Les caches en lecture seule simplifient la gestion de la cohérence puisque les données du cache et les données du périphérique de stockage principal sont synchronisées. Cependant, il est important de détecter les pannes des nœuds du système de cache réparti afin de garantir certaines propriétés comme la vivacité. Pour cela, des mécanismes de monitoring peuvent être mis en place pour surveiller la disponibilité des nœuds du système et éviter d'attendre indéfiniment une donnée d'un nœud du cache réparti.

Caches en écriture. Les caches répartis plus complexes qui permettent notamment les écritures différées doivent nécessairement disposer de mécanismes de tolérance aux pannes pour éviter les pertes de données. Gray et Cheriton proposent l'utilisation de *leases* [37] qui permettent de garantir la cohérence des données et donc éviter la perte de mise à jour en donnant aux clients des baux d'une durée limitée. Une fois le bail expiré, le client doit synchroniser sa donnée sur le support de stockage principal. Les *leases* permettent de tolérer les pannes (non-byzantines), à condition de maîtriser la dérive des horloges du système.

La réplication est également une approche qui a été utilisée dans le cadre des systèmes de caches répartis. C'est notamment le cas de NFS-CD [11, 12], où les auteurs proposent de répliquer les écritures de façon synchrone dans la mémoire de N nœuds pour tolérer au plus $N-1$ fautes. Dans NFS-CD toutes les écritures sont, à terme, écrites sur un support stable, partagé entre les nœuds.

Dans le système de fichiers réparti PAFS [28, 29], des buffers sont distribués sur plusieurs nœuds et les auteurs proposent l'utilisation de mécanismes de calcul de parité similaire au *RAID 5* [80], et de serveurs de parité qui sont responsables de la maintenance des informations de parité et de la reconstruction de blocs en cas de faute.

2.4.3 Sécurité

Un cache réparti peut permettre de déplacer des données vers le cache d'une machine cible avant de les supprimer du cache local. Beaucoup de systèmes de caches coopératifs supposent que l'environnement dans lequel ils sont déployés est sûr [33, 87, 112] : les machines peuvent se faire confiance. Or, si les nœuds du cache réparti ne sont pas tous de confiance, par exemple dans le contexte d'un cloud public, il devient nécessaire d'inclure dans le système de cache réparti des mécanismes permettant d'assurer l'intégrité et la confidentialité des données confiées à des machines tierces.

Garantir l'***intégrité*** des données consiste à pouvoir affirmer que les données ont ou n'ont pas été altérées. Une approche simple dans le cas de caches distants consiste à conserver un *hash* des blocs qui sont transmis à d'autres nœuds, ce qui permet de vérifier l'intégrité du bloc lorsqu'on le récupère. Cependant, ce type d'approche peut devenir couteux, en particulier si on veut autoriser la modification des données par certains nœuds : il faudra dans ce cas recalculer le hash. Shrira et Yoder [91] ont proposé une solution basée sur du hachage incrémental [14], permettant de garantir l'intégrité des données exportées chez d'autres nœuds tout en leur permettant la modification partielle de celles-ci sans qu'il soit nécessaire de recalculer le hash sur toute la donnée.

Garantir la ***confidentialité*** consiste à s'assurer qu'une donnée ne puisse pas être lue par une entité (un nœud) non autorisée. Une méthode couramment utilisée consiste à utiliser le chiffrement, soit en chiffrant les communications entre deux nœuds de confiance, soit en chiffrant les données que l'on souhaite placer dans le cache réparti. Un exemple de système de cache réparti garantissant à la fois des contraintes d'intégrité et de confidentialité est proposé avec le système de fichiers distribué Shark [7]. Dans ce système, le serveur de fichiers distribue aux clients autorisés des *jetons* permettant à un client de prouver aux autres qu'il a le droit de lire un fichier particulier. Un client utilise un jeton pour calculer l'index d'un bloc et consulter un annuaire distribué pour savoir quels nœuds possèdent ce bloc. Le jeton est ensuite utilisé pour sécuriser le canal de communication entre les nœuds pendant les échanges en utilisant un chiffrement symétrique tel qu'*AES*.

2.4.4 Généricité

Une des caractéristiques des caches répartis qui nous intéresse particulièrement concerne la généricité. En effet, s'ils peuvent s'implanter à plusieurs niveaux au sein

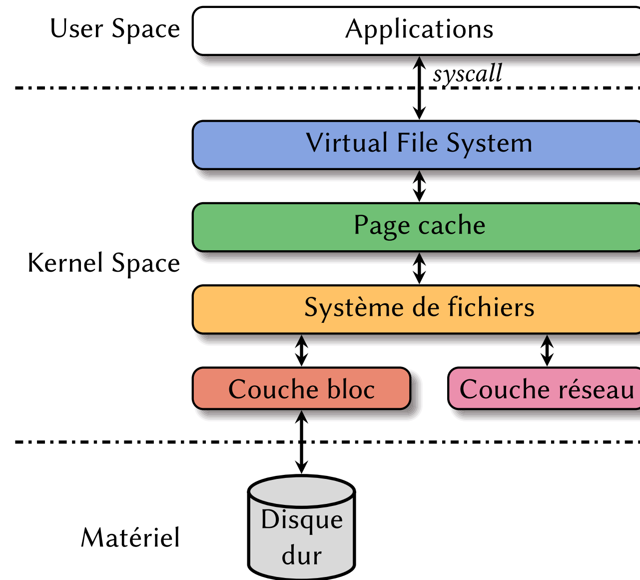


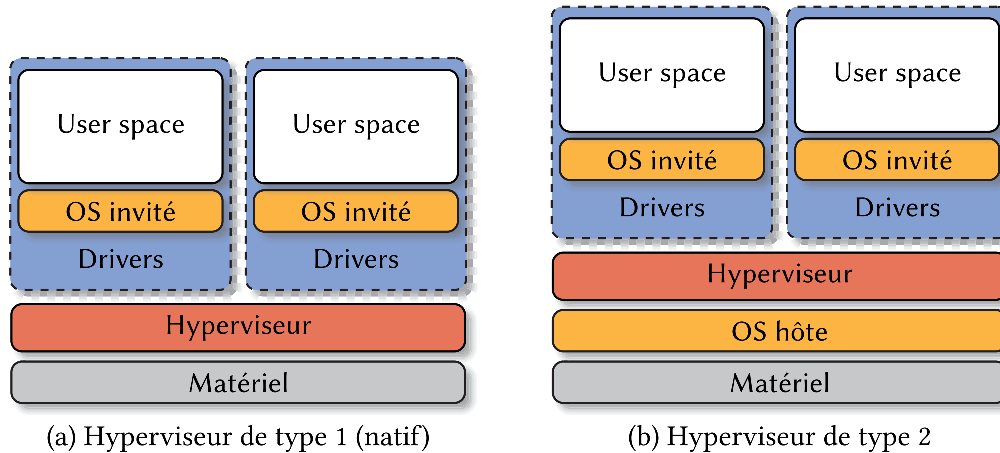
FIGURE 2.7 – Les différentes couches d'un système d'exploitation

d'un système d'exploitation (voir figure 2.7), les différentes approches impliquent certaines contraintes sur le type d'applications qui pourront s'en servir.

Certains caches répartis se situent au ***niveau applicatif***, dans l'espace utilisateur. C'est le cas par exemple de *Memcached* [34], un cache réparti principalement utilisé pour conserver en cache le résultat de requêtes à des bases de données. Le défaut de ce type d'approche est que les applications qui l'utilisent doivent être conçues pour, ou spécialement adaptées pour utiliser l'API offerte par le service de cache. C'est le cas par exemple du serveur de base de données PostgreSQL [94] pour lequel une extension a été développée pour fournir une interface à Memcached (*pgmemcache*).

Les approches intégrées au système d'exploitation ont l'avantage d'être plus modulaires et peuvent être utilisées avec les applications sans qu'il ne soit nécessaire de les adapter. Il est par exemple possible de proposer un cache réparti au ***niveau du système de fichiers*** [6, 7, 28], ce qui permet de profiter des fonctionnalités offertes par les couches inférieures du système (buffering, prefetching, ...). Cependant, de multiples systèmes de fichiers existent, chacun proposant des fonctionnalités variées (chiffrement, compression, *copy-on-write*, etc.), voire sont spécialisés pour des supports de stockage utilisant des technologies particulières comme la mémoire flash [59, 111] ou le *Shingled Magnetic Recording* (SMR) [58, 95].

Une alternative consiste à se rapprocher du périphérique de stockage en se plaçant au ***niveau de la couche bloc*** [39], mais ce type d'approche présente un autre inconvénient : certains systèmes de fichiers, notamment les systèmes de fichiers distribués tel que NFS, ne reposent pas sur la présence d'un périphérique bloc, ce qui rend ce type de cache inapproprié.



(a) Hyperviseur de type 1 (natif)

(b) Hyperviseur de type 2

FIGURE 2.8 – Les différents types d'hyperviseurs

Comme nous le verrons dans le chapitre 4, une autre solution, que nous avons privilégiée au travers de cette thèse, consiste à se placer à un ***niveau intermédiaire***, directement dans le *page cache* du système d'exploitation (voir section 3.2.3, ce qui permet de rendre compatible le cache réparti avec n'importe quel système de fichiers.

2.5 Application aux environnements virtualisés

L'avènement du cloud computing a permis de démocratiser l'usage des techniques de *virtualisation*. Leurs avantages sont multiples :

- Consolidation des ressources : plusieurs petits serveurs physiques peuvent être consolidés au sein d'un seul serveur physique de plus grande capacité.
- Isolation des applications : des applications qui autrefois partageaient le même système d'exploitation peuvent désormais être isolées les unes des autres dans des machines virtuelles (VMs), et disposer chacune de son propre système d'exploitation.
- Provisioning : de nouvelles VMs peuvent être facilement déployées.
- Continuité de service : les VMs peuvent être déplacées d'une machine physique à une autre, par exemple en prévision d'une maintenance nécessitant un redémarrage.

2.5.1 Différents types d'hyperviseurs

La plupart des méthodes de virtualisation reposent sur un *hyperviseur*, parfois appelé *moniteur de machines virtuelles* (*VMM*, pour *virtual machine monitor*), qui correspond au système d'exploitation installé directement sur la machine physique (*hôte*) et

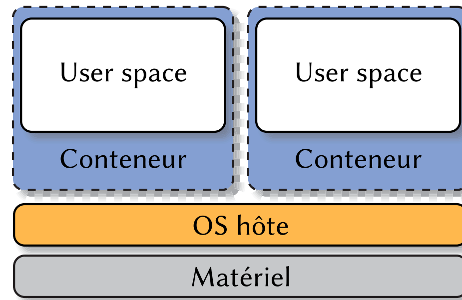


FIGURE 2.9 – Virtualisation au niveau système d'exploitation (isolateur)

qui a pour rôle d'exécuter des VMs. Les VMs exécutent le système d'exploitation *invité*. Les hyperviseurs peuvent être regroupés en 3 catégories :

Type 1. Les hyperviseurs de type 1 (figure 2.8a), appelés parfois *natifs* ou *bare metal* s'exécutent directement sur le matériel de la machine hôte et exécutent les VMs dans des processus. VMware ESXi [104], Xen [10], Microsoft Hyper-V [102] et KVM [56] sont des exemples de tels hyperviseurs.

Type 2. Les hyperviseurs de type 2 (figure 2.8b) s'exécutent au dessus du système d'exploitation existant (*hôte*) de la même façon que les autres applications. VMware Workstation [21] et VirtualBox [105] sont des exemples de tels hyperviseurs.

Type 3. À l'inverse des hyperviseurs de type 1 et 2, la virtualisation au niveau système d'exploitation [74, 93] (figure 2.9), qu'on appelle plutôt *conteneurs*, ne repose pas sur des mécanismes matériels ou sur des modifications du système d'exploitation invité. Cette approche repose sur des mécanismes d'isolation fournis par le système d'exploitation hôte similaires au `chroot`, comme le cloisonnement des différents espaces de noms du noyau (*cgroups*), permettant d'isoler les ressources des différents conteneurs. Ainsi, si la littérature ne définit pas d'hyperviseur de type 3, le rôle joué par le système d'exploitation hôte est ici proche de celui joué par les hyperviseurs. LXC (Linux Containers), Docker [71] et Linux-VServer [93] sont des exemples d'hyperviseurs au niveau système d'exploitation.

2.5.2 Techniques de virtualisation

Plusieurs méthodes de virtualisation existent, notamment la virtualisation complète (*full virtualization*), la *paravirtualisation* et la virtualisation assistée par le matériel. Le choix de l'une de ces méthodes est un compromis entre les performances de la VM et le niveau d'isolation et de transparence exigé.

Virtualisation complète. La virtualisation complète permet d'exécuter un système d'exploitation *invité* sans aucune modification, au prix d'une importante dégradation

des performances. En effet, les opérations privilégiées exécutées par le système d'exploitation invité doivent être *trappées* par l'hyperviseur puis émulées (translation de code) pour pouvoir être exécutée dans des conditions sûres avant de lui rendre la main. Cependant, cela permet d'exécuter n'importe quel système d'exploitation, ce qui explique que beaucoup d'hyperviseurs supportent ce mode de virtualisation.

Paravirtualisation. La paravirtualisation est une technique de virtualisation qui a été introduite afin de palier le problème de performances de la virtualisation complète. Celle-ci présente des interfaces spécifiques aux VMs pour réduire le surcout de la virtualisation. Les systèmes d'exploitation invités sont désormais « conscients » qu'ils s'exécutent au sein d'une VM et utilisent ces interfaces pour éviter à l'hôte d'émuler certaines opérations privilégiées et limiter ainsi le surcout de la virtualisation. Le support de la paravirtualisation peut être fourni par l'intermédiaire d'APIs telles que paravirt_ops, spécialisée dans les opérations privilégiées (CPU, MMU, etc.) ou *virtio* [84], spécialisée dans la paravirtualisation des E/S.

Virtualisation assistée par le matériel. La virtualisation assistée par le matériel permet de palier les défauts de la paravirtualisation, qui peut nécessiter de lourdes modifications des systèmes d'exploitation invités. Pour cela, des extensions matérielles ont été ajoutées aux processeurs, telles qu'Intel VT-d ou AMD-V [1], et permettent d'exécuter des VMs dans un nouveau mode privilégié pour éviter d'avoir à émuler certains mécanismes, tels que la *shadow page table.* De nos jours, beaucoup d'hyperviseurs supportent ce mode de virtualisation [10, 56, 102, 104]. Couplé avec de la paravirtualisation, on parlera de virtualisation *hybride* [76].

2.5.3 Mécanismes de gestion de la mémoire pour les machines virtuelles

Le nombre de VMs pouvant être hébergées sur un même hôte est souvent limité par la quantité de mémoire disponible. Cependant, plusieurs mécanismes permettent aux hôtes d'allouer plus de mémoire aux VMs que ce qu'ils possèdent (*memory overcommitment*). Le ***swap*** est une approche existante dans d'autres contextes qui a été améliorée spécifiquement pour répondre aux besoins des hyperviseurs [5]. Une autre solution est le ***memory ballooning*** [104], qui permet de redimensionner dynamiquement la quantité de mémoire attribuée à une VM.

Swap. Une approche classique permettant allouer plus de mémoire que l'on en possède consiste à placer des pages des processus dans un espace d'échange appelé *swap*, généralement placé sur un disque dur. Lorsque ces pages sont écrites dans le swap

(*swap out*), elles peuvent être réattribuées à d'autres processus. Lorsque le processus tente d'accéder à une page *swappée*, le matériel déclenche une *faute de page* qui peut être traitée par le système d'exploitation, lequel devra alors charger la page depuis le swap (*swap in*).

Le principe est le même avec des VMs, celles-ci étant vues comme des processus par l'hyperviseur. Cependant, les VMs gèrent elles même leur espace mémoire, elles disposent notamment d'un espace de *cache* permettant d'accélérer les E/S. La difficulté ici est que l'hôte n'a pas connaissance de l'utilisation qui est faite de cette mémoire : une page de cache du système d'exploitation invité pourrait alors être déchargée dans le swap, alors qu'elle est déjà présente sur le disque (en tant que bloc d'un fichier), ce qui dégraderait les performances. C'est ce qu'on appelle le *gap sémantique* [24].

Pour limiter les effets du gap sémantique, Amit *et al.* ont proposé VSwapper [5], un gestionnaire de swap utilisé par l'hyperviseur KVM qui lui permet notamment de détecter qu'une page physique (de l'invité) contient un bloc disque non modifié, et qu'il n'est donc pas nécessaire la swapper. La particularité de leur approche est qu'elle ne nécessite aucune modification au sein du système d'exploitation invité, l'hyperviseur est capable de « deviner » l'usage qui est fait des pages en monitorant les accès au périphérique bloc virtuel de la VM.

De façon similaire, Jones et al. [52] ont proposé des techniques qui permettent de deviner les pages du page cache du système d'exploitation invité en utilisant un monitoring de l'activité disque de la VM.

Memory ballooning. Pour changer dynamiquement la quantité de mémoire allouée à une VM, Waldspurger et al. [104] ont proposé un mécanisme appelé *ballooning*. Leur approche consiste en l'ajout d'un pilote dans la VM chargé d'allouer de la mémoire (*gonflage* du ballon). La mémoire allouée est *épinglée* (*memory pinning*) et rendue à l'hyperviseur qui peut la réattribuer à une autre VM (figure 2.10), laquelle pourra *dégonfler* son ballon pour augmenter sa quantité de mémoire. Cette solution a été depuis adaptée à d'autres hyperviseurs comme Xen [10] ou KVM [56, 89].

Si le *ballooning* permet de redimensionner *dynamiquement* la mémoire, il est rarement *automatique* et est généralement contrôlé manuellement. Cependant, il peut être automatisé depuis l'hyperviseur en utilisant des techniques d'échantillonnage des pages utilisées [104] ou d'approximations de la LRU [116], ou à l'aide d'un contrôleur externe à l'hyperviseur tel que MoM (*Memory Overcommitment Manager*) [65]. Dans KVM, l'automatisation du *ballooning* [23] repose sur des informations sur la pression mémoire en provenance des VMs.

L'approche à base de ballons présente deux défauts majeurs principalement dus au gap sémantique entre les VMs et l'hyperviseur :

— il est difficile de tenir compte des applications intensives en E/S, qui allouent

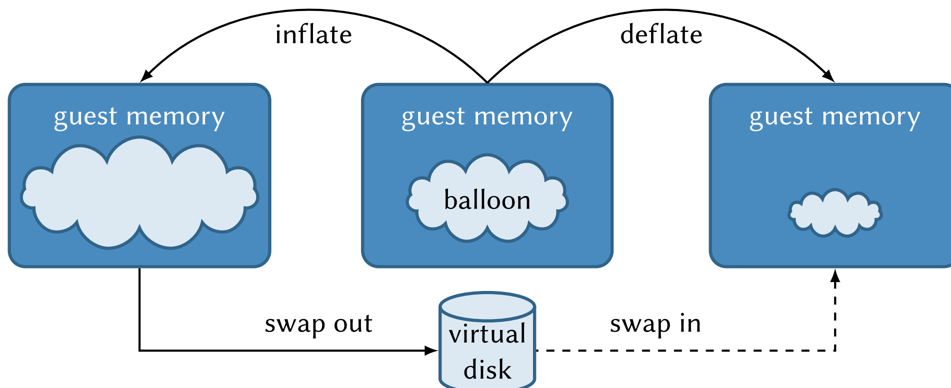


FIGURE 2.10 – Le gonflement du ballon augmente la pression mémoire de l'invité et le contraint à swapper. Le dégonflement du ballon permet de réduire la pression mémoire et donc de réduire le swap [5, 104].

peu de mémoire mais nécessitent beaucoup de cache pour atteindre des performances acceptables ;

— une fois que la mémoire a été réattribuée, il est difficile, voire impossible, de la récupérer sans *swapper* [46, 85, 104], ce qui dégrade les performances globales des applications.

2.5.4 Réduction de l'empreinte mémoire

De façon orthogonale, des techniques permettant de réduire l'empreinte mémoire des VMs telles que la ***déduplication*** [73] ou la ***compression*** [100] sont souvent utilisées pour attribuer plus de mémoire aux VMs que n'en possède la machine physique.

Une des méthodes de déduplication est le partage transparent de pages [20] : l'hyperviseur scanne périodiquement la mémoire de chaque VM, et lorsque des pages identiques sont trouvées, elles sont partagées de sorte que les VMs accèdent aux mêmes pages physiques. L'hyperviseur peut ensuite détecter une modification de la page par l'une des VMs (faute de page) et la dupliquer si nécessaire (*copy-on-write*). L'inconvénient majeur est que scanner la mémoire consomme du temps processeur et de la bande passante mémoire. Pour réduire le cout de ce scan, il est possible de paravirtualiser le mécanisme (*i.e.*, la VM fournit des indications à l'hyperviseur) de façon à limiter les pages scannées à celles qui offrent le plus de chances d'être partagées, en se concentrant par exemple sur les pages du page cache [57]. C'est ce type d'approche qui a été utilisé par Satori [73] et KSM++ [72] pour limiter le surcout de la déduplication.

La réduction de l'empreinte mémoire des VMs peut aussi passer par des techniques de compression [100], souvent associées à la déduplication [38].

2.5.5 Caches collaboratifs dans les machines virtuelles : approches existantes

À notre connaissance, peu de solutions de caches spécialisés pour les environnements virtuels ont été proposés. Dans cette section, nous proposons d'en étudier deux : XHive [55] et Mortar [46].

XHive. XHive [55] est un cache collaboratif pour l'hyperviseur Xen qui repose sur l'analyse des accès aux périphériques blocs virtuels des VMs. Les auteurs de XHive utilisent un algorithme de cache collaboratif similaire à *N-Chance*, présenté dans la section 2.3.2.5.

La difficulté est ici, pour l'hyperviseur, de savoir quand une VM est *inactive* de façon à lui envoyer les blocs *singletons.* Ce problème est similaire à celui posé par le *memory ballooning*, en effet nous avons vu dans la section précédente que cette approche pose deux problèmes : d'une part l'hyperviseur ne sait pas à quoi sert la mémoire qu'il prend à une VM ; et d'autre part une fois que la mémoire a été donnée à une autre VM il est difficile de la récupérer. Ceci est d'autant plus dommageable dans le cadre d'un cache collaboratif, notamment parce que « prêter » de la mémoire au cache ne devrait pas impacter les performances.

Les auteurs de XHive proposent d'adopter une approche *contributive*, dans laquelle l'hyperviseur stocke temporairement les *singletons* évincés par les VMs dans une mémoire tampon. Les VMs peuvent ensuite, volontairement, collecter les blocs dans leur mémoire pour libérer le tampon de l'hyperviseur.

Si leur approche est intéressante, elle se retrouve fortement limitée par la taille de ce tampon, la mémoire disponible pour l'hyperviseur Xen n'est en effet que de 10 Mo. Ce tampon peut rapidement saturer lors d'une charge intensive en E/S, rendant le mécanisme de cache complètement inefficace. De plus, XHive nécessite de lourdes modifications à la fois à l'hyperviseur, qui monitore les accès aux périphériques de stockage et coordonne les accès au cache, et aux systèmes d'exploitation invités, chargés de collecter les données du cache. Ces choix limitent fortement l'utilisation de cette approche à des VMs adaptées à un hyperviseur spécifique. De plus, l'approche centralisée sur l'hyperviseur limite l'usage d'une telle solution à un seul nœud, ce qui ne permet pas de consolider l'utilisation de la mémoire à l'échelle d'un cluster ou d'un centre de données.

Mortar. Hwang et al. [46] se sont intéressés au problème du surdimensionnement (*overprovisionning*), où l'hyperviseur possède plus de mémoire qu'il n'en alloue aux VMs. La difficulté ici est d'attribuer cette mémoire libre aux VMs sous forme de cache pour qu'elle puisse être utile et facilement récupérée. Dans ce contexte, une approche

reposant sur du *memory ballooning* n'est pas viable, car une fois donnée aux VMs il est difficile de récupérer la mémoire puisque l'hyperviseur ne sait pas quel en sera l'usage.

Les auteurs ont proposé Mortar, un framework permettant à l'hyperviseur d'utiliser sa mémoire libre pour fournir un cache *clé-valeur* aux VMs. Appliqué par exemple à Memcached [34], ce cache peut être utilisé par des applications existantes (qui utilisent Memcached) pour qu'elles puissent « placer » des objets dans cette mémoire gérée par l'hyperviseur. Les données stockées ainsi sont en lecture seule et peuvent donc être réclamées facilement, ce qui permet de récupérer rapidement la mémoire s'il faut la réattribuer lors d'un pic de charge ou lors du démarrage d'une nouvelle VM.

Si Mortar règle le problème de surdimensionnement des machines physiques, Birke et al. [16] ont montré que dans un contexte de *cloud privé* d'entreprises, où les propriétaires des VMs disposent d'un usage exclusif des machines physiques, le niveau de surdimensionnement est plutôt faible : la quantité de mémoire allouée par l'hyperviseur pour les VMs était proche de la quantité de mémoire physique disponible.

2.6 Discussion

La fragmentation de la mémoire dans les environnements virtualisés a lourdement impacté les performances des applications qui effectuent beaucoup d'E/S. En effet, leurs performances dépendent en partie de la quantité de mémoire inutilisée disponible, qui est alors utilisée par le système d'exploitation pour du cache (*page cache*). En fragmentant la mémoire, chaque VM dispose de moins de cache : celles qui n'en n'ont pas besoin en « gaspillent », alors que d'autres, orientées données, en auraient besoin de plus.

Une approche classique pour mutualiser de la mémoire est d'utiliser un système de cache réparti. Cependant, la plupart des solutions de caches répartis existantes reposent sur une adaptation des applications [34] ou sur des systèmes de fichiers spécifiques [6, 7, 28]. Or, dans ce type d'environnement la diversité des applications nécessite une solution aussi transparente que possible.

Une autre approche, spécifique aux environnements virtualisés, consiste à redimensionner dynamiquement la mémoire des VMs pour éviter de la gaspiller (*memory ballooning*). Cependant, ses limites sont vite atteintes lors d'un pic d'activité : le *gap sémantique* entre les VMs et l'hyperviseur l'empêche de récupérer efficacement la mémoire qu'il a donné, même si elle est utilisée pour faire du cache.

Nous avons vu dans ce chapitre d'autres approches reposant à la fois sur l'hyperviseur et le système d'exploitation invité [46, 55]. Cependant, elles souffrent de problèmes majeurs :

— en introduisant des modifications à la fois à l'hyperviseur et au système d'exploi-

tation invité, elles limitent fortement le nombre d'environnements dans lesquels elle peut être utilisée ;
- en reposant sur l'hyperviseur, il n'est plus possible d'envisager une mutualisation de la mémoire inutilisée à l'échelle d'un centre de données ;
- ces approches reposent toutes sur la présence de mémoire au sein de l'hyperviseur, ce qui n'est pas envisageable dans des environnements fortement consolidés, où l'hyperviseur ne dispose plus de mémoire libre, tels qu'on les retrouve dans les clouds privés [16].

Pour ces raisons, nous pensons qu'une approche plus classique, reposant sur un réseau performant entre les VMs et les nœuds, permet d'offrir suffisamment de performances pour mutualiser la mémoire à l'échelle d'un centre de données. Cependant, afin d'offrir un maximum de transparence tout en limitant l'intrusivité du mécanisme, une approche plus « bas niveau », située au cœur de la gestion des caches existants du système, nous semble nécessaire.

Chapitre 3

Contexte et prérequis techniques

Sommaire

3.1 Principes des systèmes de fichiers 32
3.2 Les différents caches 34
3.2.1 *dentry cache* 34
3.2.2 Cache d'*inode* 35
3.2.3 Page cache 36
3.2.4 Buffer cache 38
3.2.5 Unification du *page cache* et du *buffer cache* 39
3.3 Gestion de la mémoire 39
3.3.1 Récupération de la mémoire et activation des pages 40
3.3.2 *Shadow page cache* : estimation de la taille du *working set* 41
3.3.3 Détection des accès au cache avec l'API *cleancache* 42
3.4 Conclusion 43

Ce chapitre présente et définit les différents éléments techniques nécessaires à la compréhension de la réalisation de notre cache réparti, présenté dans le chapitre 4. Le noyau Linux dispose de plusieurs sous-systèmes et couches d'abstraction permettant d'étendre ses fonctionnalités. Parmi ceux-ci, nous nous intéressons plus particulièrement au *Virtual File System* (VFS), qui permet l'accès aux fichiers, et au *page cache*, dont le but est d'accélérer les accès aux données. La réalisation d'un cache réparti intégré au noyau passe par la compréhension de ces différentes couches d'abstractions. Outre le VFS, les mécanismes de la gestion de la mémoire ont leur importance, puisque leur action sera déterminante sur l'efficacité du cache réparti. En effet, c'est à ce niveau que le déplacement d'une donnée d'un cache « local » vers un cache « distant » va se décider. Une des difficultés de cette thèse a été de maîtriser ces abstractions et ces sous-systèmes car ils sont complexes et peu documentés.

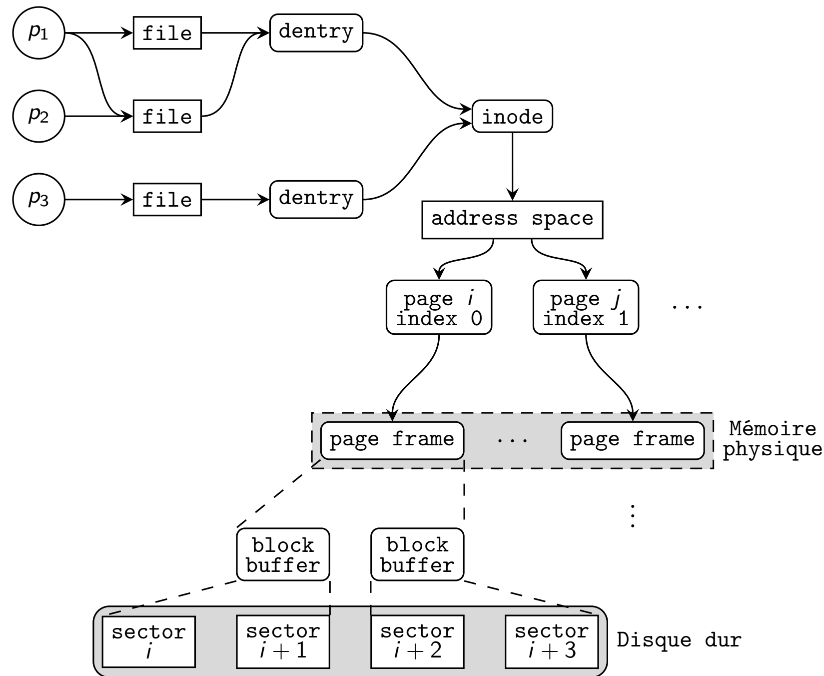


FIGURE 3.1 – Interactions entre les différentes abstractions fournies par le VFS, la mémoire et le périphérique de stockage

Dans un premier temps, ce chapitre présente de façon simplifiée les grands principes des systèmes de fichiers par l'intermédiaire du VFS (section 3.1). Ensuite, la section 3.2 définit les différents caches présents au sein du noyau Linux et permet d'identifier précisément ceux qui nous intéressent. Enfin, la section 3.3 étudie le fonctionnement de la mémoire du noyau Linux, en particulier les algorithmes de récupération de la mémoire et d'estimation de la taille du *working-set* sur lesquels reposent une partie de nos travaux.

3.1 Principes des systèmes de fichiers

L'accès à un fichier par un processus implique l'activation de plusieurs mécanismes complexes du noyau qui ne sont pas directement liés à une opération d'entrée/sortie (E/S) mais dont la maîtrise est cruciale pour ne pas dégrader les performances. La plupart de ces mécanismes font partie ou « traversent » le VFS, qui est la couche d'abstraction fournie par le noyau Linux et qui permet aux différents systèmes de fichiers de coexister. Ces mécanismes peuvent être représentés par les différents objets qu'ils manipulent : fichiers (*file*), *inodes*, chemins (*dentry*) et espace d'adressage. La figure 3.1 illustre les interactions entre des processus et les principaux objets abstraits par le VFS.

Fichiers. La structure `file` est la représentation d'un fichier *ouvert* par un processus (mode d'ouverture, position dans le fichier, etc.). C'est notamment cette structure qui permet à plusieurs processus de manipuler le même fichier indépendamment l'un de l'autre. Il est également possible de partager un même fichier ouvert, par exemple lorsqu'un processus effectue un `fork`, le processus fils hérite des descripteurs de fichiers ouverts du processus père. Dans l'exemple de la figure 3.1, le processus p_2 est un fils du processus p_1 et a hérité des descripteurs de p_1.

Inodes. Un fichier est représenté en mémoire par un `inode` qui permet de décrire les propriétés du fichier (numéro d'*inode*, droits d'accès, taille, emplacement des blocs sur le disque, etc.). À noter que cet objet est la version abstraite fournie par le VFS, qui est différent des *inodes* spécifiques à chaque système de fichiers présents physiquement sur le disque. Les fichiers sont identifiés par leur numéro d'*inode*, mais les utilisateurs utilisent généralement un nom, qui est représenté en mémoire par la structure `dentry` et qui permet d'associer un nom à un *inode*. Dans la figure 3.1, les processus p_1 et p_3 ouvrent trois fois le même fichier :

— p_1 ouvre le fichier deux fois en utilisant le même nom ;
— p_3 ouvre le fichier en utilisant un autre nom : il s'agit d'un *lien physique*.

Pages. En mémoire, les données lues d'un fichier sont stockées dans des pages, dont la taille dépend de l'architecture matérielle. Typiquement, la taille des pages est de 4 Ko sur les architectures x86 et x86-64. Ces pages sont associées à un *inode* par l'intermédiaire d'un espace d'adressage spécifique représenté par la structure `address_space`. Au sein de cette structure, les pages sont identifiées par un *index*, qui correspond à leur emplacement *virtuel* relativement au début du fichier (*offset*). Cette structure stocke les pages dans un ***arbre radix***, dont l'implémentation a l'avantage d'être compacte et offre des accès en lecture *wait-free* (à l'aide de RCUs (read-copy-update)). Chaque *inode* a donc son propre arbre radix. Une illustration de l'arbre radix est présente dans la figure 3.5.

Les structures `page` permettent de représenter les pages de la mémoire physique et leurs propriétés, telles que les droits d'accès, le type de page, les *flags*, etc. Généralement, on utilise le terme de *page frame* lorsque l'on parle de l'espace d'adressage *physique*, c'est-à-dire des pages physiques présentes dans la RAM, et le terme de *page* lorsque l'on parle de l'espace d'adressage *linéaire* (ou *virtuel*) du système.

Blocs. Les pages associées aux inodes contiennent un ou plusieurs *blocs* lus depuis le disque : le bloc est l'unité de base utilisée par les systèmes de fichiers. Un bloc représente un ou plusieurs *secteurs* consécutifs du périphérique de stockage : le secteur est l'unité de base manipulée par les périphériques bloc et représente un ensemble d'octets contigus. La taille des blocs est définie par le système de fichiers et est au minimum

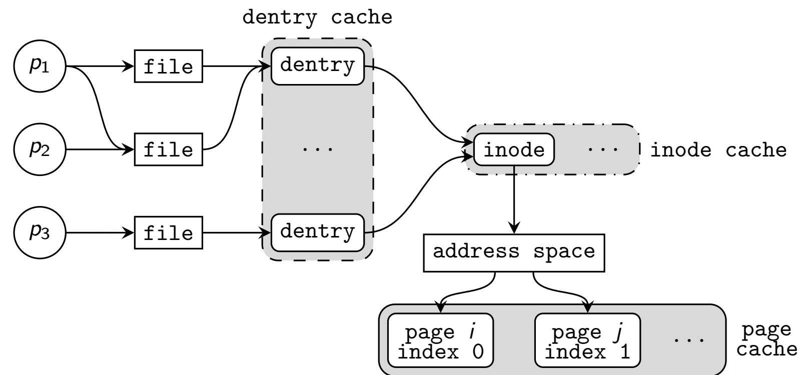


FIGURE 3.2 – Caches disques du *VFS*

égale à la taille d'un secteur (512 octets) et au maximum égale à la taille d'une page. Un bloc lu est stocké dans une page et représenté par un *block buffer*. Une configuration classique est d'utiliser une taille de bloc égale à la taille d'une page, ce qui permet de se passer de ce niveau d'indirection.

3.2 Les différents caches

Le VFS fournit un ensemble de caches spécialisés pour permettre aux systèmes de fichiers de conserver en mémoire des données pour accélérer leurs futures opérations. Ces caches, illustrés dans la figure 3.2, sont présentés dans les sections suivantes. Le *dentry cache* est un cache permettant d'accélérer la correspondance entre le nom d'un fichier et son inode (section 3.2.1). Le cache d'*inode* permet de conserver en mémoire les inodes des fichiers après leur utilisation (section 3.2.2). Le *page cache* conserve en mémoire les données des fichiers qui ont été accédés par l'intermédiaire du système de fichiers (section 3.2.3). Le *buffer cache*, qui n'est pas représenté sur la figure 3.1, est utilisé pour conserver des blocs disque en mémoire lorsqu'un périphérique bloc est accédé en mode bloc (section 3.2.4). Nous expliquons dans la section 3.2.5 pourquoi le *buffer cache* n'est pas représenté dans cette figure.

3.2.1 *dentry cache*

Lors de l'ouverture d'un fichier par un processus, par exemple à l'aide de l'appel système open, le *dentry cache* fournit le chemin du fichier au système de fichiers qui doit retrouver l'*inode* correspondant. Ce chemin contient le nom de chaque répertoire à traverser pour obtenir le fichier. Chaque répertoire successif du chemin, en commençant par la racine, est représenté par un inode qu'il est nécessaire de localiser pour connaitre via son `dentry` l'inode du répertoire suivant. Pour accélérer les prochains

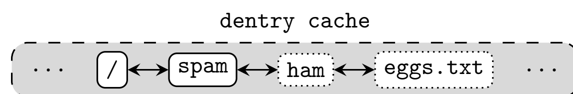


FIGURE 3.3 – Résolution d'un nom de fichier à l'aide du *dentry cache*

accès à ce chemin, le noyau Linux dispose d'un cache appelé *dentry cache*, qui conserve en mémoire les `dentry` déjà résolus. Les `dentry` présents dans ce cache sont maintenus dans une liste LRU qui permet de récupérer automatiquement l'espace occupé par les `dentry` les moins récemment utilisés en cas de pression mémoire.

Lorsque les `dentry` correspondant au chemin ne sont pas dans le *dentry cache*, il est nécessaire d'effectuer une lecture depuis le périphérique pour l'obtenir. La figure 3.3 illustre la résolution du fichier dont le nom est `/spam/ham/eggs.txt`. Ce chemin est composé de 4 *dentry* : `/`, `spam`, `ham` et `eggs.txt`. Le *dentry cache* ne contient que la racine (`/`) et le répertoire `spam`, il est donc nécessaire d'effectuer 2 lectures depuis le périphérique bloc :

1. pour localiser l'inode du répertoire `/spam/ham` qui contient le fichier `eggs.txt` ;
2. pour récupérer l'inode de `eggs.txt`

Le prochain accès au fichier `/spam/ham/eggs.txt` ne nécessitera pas d'accès au périphérique de stockage pour trouver son numéro d'inode, sauf si la pression mémoire exercée sur le système nécessite la récupération des `dentry` qui composent son chemin.

3.2.2 Cache d'*inode*

Les inodes du VFS, que l'on appelle également ***inodes virtuels***, sont des représentations en mémoire des inodes du système de fichiers, qui sont eux appelés ***inodes disque***. Un inode disque permet de décrire les métadonnées d'un fichier, telles que le propriétaire, les droits ou la date de dernière modification, ainsi que les blocs qui le composent. L'inode disque est stocké directement sur le périphérique de stockage, contrairement à l'inode virtuel qui n'est présent qu'en mémoire et inclut, notamment, les informations de l'inode disque.

Les données des fichiers sont stockées sur le disque dans des blocs, dont la taille est définie par le système de fichiers. Classiquement, les inodes disque des systèmes de fichiers sous Unix permettent d'accéder directement aux premiers blocs du fichier, les autres pouvant être accédés via un ou plusieurs blocs d'*indirection*. Ainsi, pour lire un bloc d'un fichier il est nécessaire d'accéder à l'inode et éventuellement à plusieurs blocs d'indirection pour pouvoir le localiser.

Les systèmes de fichiers récents comme Btrfs [83] ou ext4 [69] reposent sur le concept d'***extents***. Un extent est une zone de stockage contigüe réservée pour le fichier.

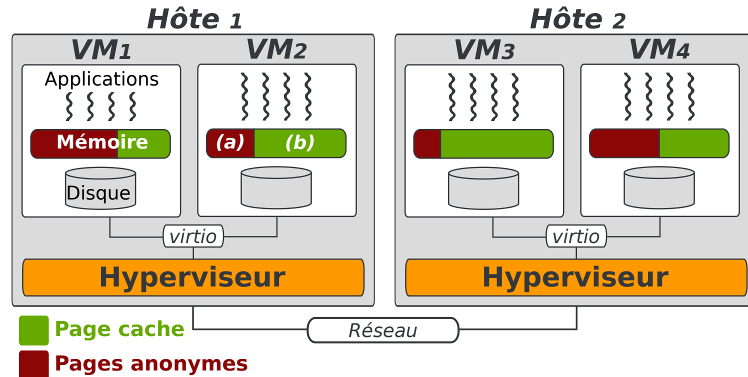


FIGURE 3.4 – Virtualisation et *page cache* du système d'exploitation

Les extents sont de taille variable, ce qui permet à de petits fichiers d'être représentés par 1 seul extent, par exemple de 64 ko, et à de plus gros fichiers d'être représentés par seulement quelques extents, jusqu'à 128 Mo par extent par exemple pour le système de fichiers ext4. L'intérêt des extents par rapport aux blocs d'indirection est qu'ils permettent de réduire le nombre d'indirections nécessaires à la lecture d'un bloc, et donc le nombre d'E/S. De plus, les blocs du fichier au sein d'un extent sont contigus, ce qui permet de limiter la fragmentation du fichier.

De façon similaire au *dentry cache*, le rôle du cache d'inodes est de conserver ces informations en mémoire pour réduire le nombre d'opérations d'E/S liées aux inodes. Au sein de ce cache les inodes sont maintenant dans une liste LRU pour les réclamer en cas de pression mémoire.

3.2.3 Page cache

Lorsque le système est inactif (avec des processus consommant peu de mémoire), l'essentiel de la mémoire est remplie par un cache appelé ***page cache***. L'objectif de ce cache est de conserver les données lues depuis le disque afin d'améliorer les performances des accès aux fichiers. Lorsque l'activité du système augmente, la mémoire se remplit des pages des processus actifs, ce qui réduit la taille du page cache.

Les pages des processus sont appelées ***pages anonymes*** (pile, tas, etc.), à l'inverse des ***pages du page cache*** qui ont une représentation sur le disque (fichiers). Dans la figure 3.4, nous avons représenté les pages anonymes (a) et les pages du page cache (b) pour chaque machine virtuelle (VM). Les pages anonymes font également partie du page cache, où elles disposent de leur propre LRU. Lorsqu'elles doivent être évincées, par exemple en cas de forte pression mémoire, elles sont *swappées* sur le disque. Cependant, pour des questions de simplicité dans la suite de cette thèse nous considérons que le page cache n'est composé que de pages provenant de fichiers. Nous reviendrons sur ce détail dans la section 3.3.1.

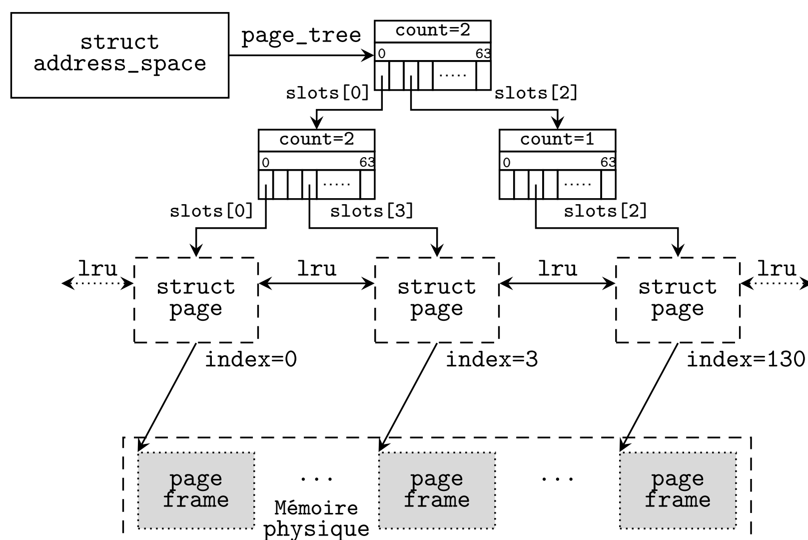


FIGURE 3.5 – Organisation du page cache

Les systèmes de fichiers utilisent le page cache pour limiter les accès au périphérique de stockage et accélérer les accès aux données. Lorsque qu'un processus veut lire une donnée dans un fichier, le système de fichiers regarde si celle-ci n'est pas déjà présente dans le page cache, dans ce cas elle peut être directement copiée dans l'espace d'adressage du processus sans nécessiter l'accès au périphérique de stockage. Lorsque la page n'est pas présente dans le page cache, le système de fichiers y ajoute une nouvelle page, non initialisée, puis prépare les opérations d'E/S nécessaires pour qu'elle soit remplie.

Le page cache est intégré directement dans les inodes du VFS, chaque inode contient une structure ***address_space*** qui représente l'espace d'adressage du fichier et contient un ***arbre radix***, dans lequel sont placées les pages lues. Les pages du page cache sont également liées entre elles dans une liste LRU, indépendamment de la structure `address_space` dans laquelle elles sont, ce qui permet d'implémenter l'algorithme de remplacement de pages global basé sur l'algorithme LRU. Nous détaillons le fonctionnement de cet algorithme dans la section 3.3.1. Cette organisation du page cache est illustrée dans la figure 3.5.

Chaque page du page cache représente une zone contigüe dans l'espace d'adressage du fichier correspondant de la taille d'une page, et est composée d'un ou plusieurs blocs lus depuis le disque. Les pages sont identifiées par un ***index***, commençant à 0, qui indique la position de la page au sein du fichier. Par exemple dans la figure 3.5, la page d'index 0 contient les données de l'inode du premier octet jusqu'à l'octet 4095, la page d'index 3 les données de l'inode de 12 ko à 16 ko, et la page d'index 130 les données de 520 ko à 524 ko. Il est donc facile de retrouver une page dans le page cache lorsque l'on sait à quelle position relativement au début fichier elle se trouve.

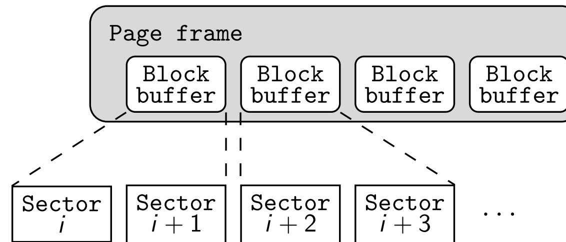


FIGURE 3.6 – Blocs contigus sur le disque

Lorsque les blocs qui composent la page ne sont pas contigus sur le disque, on associe à la page des descripteurs (*buffers*) qui permettent de décrire chaque bloc de la page indépendamment les uns des autres : c'est ce que l'on appelle une *buffer-page*, présentée dans la section suivante.

3.2.4 Buffer cache

Le VFS manipule les données des périphériques par pages, dont chacune est composée d'un ou plusieurs blocs. Le *buffer cache* est un cache qui permet d'accélérer les accès à chacun de ces blocs individuellement. Les systèmes de fichiers accèdent aux données sur le disque avec la granularité du bloc, qui est une suite contiguë de secteurs du périphérique bloc. Lors de la lecture d'un fichier, chaque bloc qui le compose est placé dans un *block buffer* qui est lui-même au sein d'une page du page cache. La figure 3.6 illustre la représentation en mémoire d'une page d'un fichier au sein du page cache : un fichier en mémoire est une suite logique de pages, chacune contenant un ou plusieurs blocs qui correspondent à une suite de secteurs.

Lorsqu'on veut lire un octet d'un fichier, il faut retrouver le numéro du *bloc disque* correspondant au *bloc logique* qui contient la position de l'octet relative au début du fichier. Par exemple, avec des pages de 4 ko et des blocs 1 ko, retrouver le caractère à la position 5000 revient à trouver le numéro de bloc physique correspondant au bloc logique 4, lui-même présent au sein de la page d'index 1.

Lorsque les blocs d'une page sont contigus sur le disque (figure 3.6), il suffit de déterminer le numéro de bloc physique du premier bloc de la page, puis ensuite d'effectuer une E/S de la taille d'une page : c'est le cas notamment si le système de fichiers a pu allouer les blocs des pages de façon contiguë sur le périphérique, ou lorsque la taille d'un bloc est la même que la taille d'une page.

À l'inverse, la figure 3.7 illustre le cas où les blocs d'une page ne sont pas contigus sur le disque : même si on souhaite lire entièrement la page, il est nécessaire de soumettre plusieurs petites E/S au pilote de périphérique, une pour chaque bloc de celle-ci. Pour repérer les différents blocs qui composent la page et pour mémoriser la correspondance entre le numéro de bloc logique et le numéro de bloc physique, on associe à

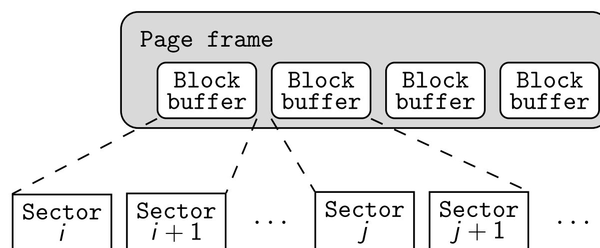


FIGURE 3.7 – Blocs non-contigus sur le disque

chaque bloc une structure `buffer_head` qui contient ces informations.

3.2.5 Unification du *page cache* et du *buffer cache*

Dans les versions du noyau Linux antérieures à la version 2.4.10 (2001), le page cache et le buffer cache étaient indépendants et des mécanismes lourds et complexes étaient présents pour assurer leur cohérence. Aujourd'hui, le buffer cache à proprement parler n'existe plus, les pages du page cache référencent les structures `buffer_head` et inversement. On peut donc définir le buffer cache actuel comme le sous-ensemble de pages du page cache qui ont des `buffer_head` associés. En effet, lorsqu'une page ne contient qu'un seul *buffer*, ce qui est le cas par exemple pour des pages de 4 ko, il n'est pas nécessaire de créer des `buffer_head`. Les structures `buffer_head` sont créées dans les situations suivantes :

- lorsqu'on accède directement à un périphérique bloc, il est alors lu bloc par bloc, par exemple lors de la lecture d'un inode, d'un superbloc ;
- lorsque les blocs qui constituent une page ne sont pas contigus sur le périphérique bloc ;
- lorsqu'une page est incomplète, par exemple en fin de fichier.

Les `buffer_head` sont également chaînés entre eux, ce qui permet de localiser rapidement une page dans le page cache qui correspond à un numéro de bloc physique donné. Les relations entre les `buffer_head` et les pages du page cache sont représentées dans la figure 3.8.

3.3 Gestion de la mémoire

Cette section présente trois mécanismes de gestion de la mémoire du noyau Linux qu'il nous semble important d'étudier afin d'aider à la compréhension de cette thèse. Le premier est l'algorithme de récupération de mémoire du noyau Linux, présenté dans la section 3.3.1, qui a pour rôle d'évincer les pages du page cache pour libérer de la mémoire. Le second est le *shadow page cache* (section 3.3.2), dont l'objectif est d'améliorer la détection des pages « chaudes » du page cache. Enfin, nous présentons dans la

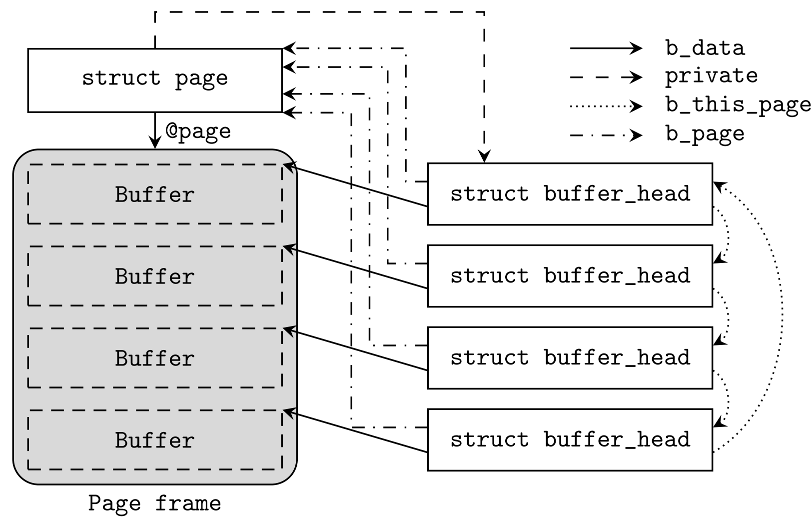


FIGURE 3.8 – Page cache unifié [18]

section 3.3.3 l'API *cleancache*, qui permet de capturer les accès et les évictions du page cache.

3.3.1 Récupération de la mémoire et activation des pages

L'algorithme de récupération de mémoire (PFRA, pour *Page Frame Reclaiming Algorithm*) du noyau Linux a pour rôle de libérer des pages du page cache. Les pages libérées sont appelées pages *victimes*. Cet algorithme se déclenche en cas de pression mémoire, par exemple lorsqu'un processus tente d'allouer de la mémoire, ou lorsque le processus noyau *kswapd* se réveille.

Le noyau Linux privilégie l'éviction des pages les moins utilisées en premier. Pour cela, les pages du page cache sont stockées dans deux listes LRU : les listes *actives* et *inactives*. Lors d'un premier accès à une page, celle-ci est placée en tête de la liste inactive. Lorsqu'une page est accédée une seconde fois, elle est considérée comme « chaude » et est promue dans la liste active, qui a pour objectif de conserver en mémoire les données les plus utilisées. L'algorithme de récupération de la mémoire du noyau Linux est similaire à l'algorithme 2Q que nous avons présenté dans la section 2.2.2.

La figure 3.9 est un exemple du fonctionnement des listes LRU du noyau Linux. Dans cet exemple, lors de la première lecture de la page P5, la page P8, qui est en queue de liste inactive, est évincée pour faire de la place. Lors d'un accès à la page P3, celle-ci est promue dans la liste active car elle était déjà présente dans la liste inactive.

Afin de ne pas se retrouver dans une situation où toutes les pages du page cache seraient actives, le noyau Linux tente de rééquilibrer la taille de ces listes lors de l'exécution de l'algorithme de récupération de mémoire. Pour cela, lorsque la liste active

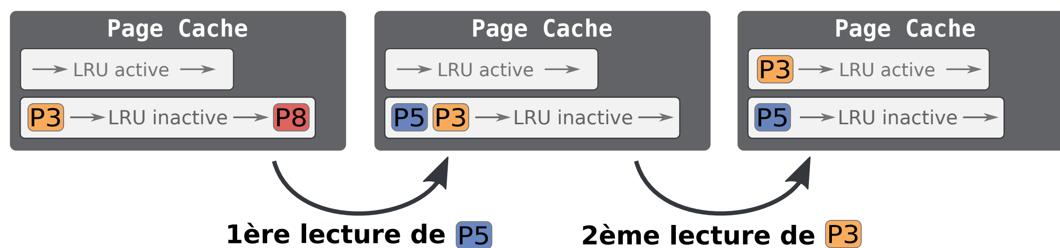


FIGURE 3.9 – Fonctionnement des listes *LRU* du noyau Linux

devient plus grande que la liste inactive, les pages les plus anciennes (*i.e.*, en queue) de la liste active sont *désactivées* jusqu'à ce que la liste inactive redevienne plus grande que la liste active. Ainsi, la liste active ne représente généralement pas plus de 50% de la taille du page cache. En revanche, s'il n'y a pas de pression mémoire, aucune page n'est évincée et un grand nombre de pages inactives peuvent être activées, ce qui explique pourquoi il est possible d'avoir une liste active bien plus grande que la liste inactive.

La plupart du temps, les pages *propres* du page cache sont évincées en priorité parce que c'est peu coûteux : elles ont une version identique sur le disque et peuvent donc être simplement libérées. Si des pages du page cache sont *sales* (c'est-à-dire modifiées), elles doivent d'abord être écrites sur le disque. Le noyau déclenche alors leur écriture et les remet en tête de liste inactive pour les réclamer plus tard, lorsqu'elles seront propres. On retrouve le même mécanisme lorsqu'une E/S est en cours sur une page ou lorsqu'une page est verrouillée (`mlock`). Ces cas particuliers démontrent à quel point l'implémentation d'un algorithme de récupération de la mémoire est complexe.

3.3.2 *Shadow page cache* : estimation de la taille du *working set*

Le découpage en deux listes a pour effet de bord de diminuer le temps que passe une nouvelle page dans le cache avant son éviction. En effet, il est possible qu'une page soit évincée de la liste inactive du page cache juste avant un nouvel accès à celle-ci, qui aurait dû la promouvoir dans la liste active. Ce cas peut se produire, par exemple, lorsque la distance, en termes de nombre de pages accédées entre les deux accès, est supérieure à la taille de la liste inactive, mais inférieure à la totalité de la taille du page cache. Dans ce cas, l'E/S qui en découle aurait pu être évité.

Pour corriger ce problème, le noyau Linux dispose depuis sa version 3.15 (2014) d'un mécanisme permettant de conserver l'historique des pages évincées [27, 107], que l'on appelle le *shadow page cache*. Le principe de ce mécanisme est de maintenir la fréquence des accès aux pages pour savoir si elles sont suffisamment utilisées pour les activer directement, y compris lorsqu'elles ne sont plus dans le cache. Cependant, monitorer la fréquence des accès aux pages est extrêmement coûteux, c'est pourquoi l'approche adoptée est une approximation qui consiste à déterminer la distance moyenne entre

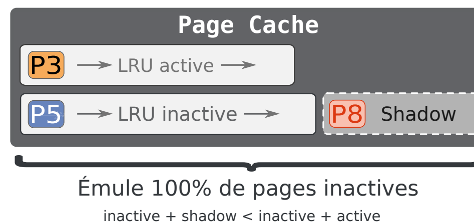


FIGURE 3.10 – *Shadow page cache*, après éviction de la page P8 (figure 3.9)

deux accès à une même page pour savoir si elle aurait dû être activée. L'implémentation de ce calcul dans le noyau Linux consiste à compter le nombre de pages évincées de la liste inactive entre les deux accès à une même page.

Le *shadow page cache* repose sur l'arbre radix utilisé pour stocker les pages du pages cache. En effet, lorsqu'une page est évincée du page cache, l'entrée correspondante de l'arbre radix persiste, puisque celui-ci ne stocke que des pointeurs vers des structures `page` (section 3.1). Ainsi, lors de l'éviction d'une page, le nombre de pages évincées de la liste inactive depuis le démarrage de la machine est stocké directement dans le nœud de l'arbre radix qui référençait la page évincée. Lorsqu'une page évincée est accédée depuis le disque, la valeur stockée dans l'arbre radix est comparée à la valeur courante : si la distance est plus faible que la totalité de page cache (*shadow hit*), le système la considère comme chaude et l'active immédiatement. La figure 3.10 illustre ce mécanisme.

3.3.3 Détection des accès au cache avec l'API *cleancache*

La complexité des mécanismes de gestion de la mémoire et du page cache du noyau Linux rendent difficiles la détection des accès au cache et des évictions des pages de la mémoire. Afin de faciliter la mise en œuvre de ce type de cache, Magenheimer et al. ont proposé l'API *cleancache* [66]. Cette API définit un ensemble de *hooks* placés au sein du noyau Linux qui permettent de détecter les évictions de pages *propres* du page cache (opération `put`) et les accès au page cache (opération `get`). D'autres opérations permettant d'invalider les données du cache (pages, inodes ou systèmes de fichiers entiers) sont fournies pour maintenir le cache dans un état cohérent.

Cette API repose sur le principe de « mémoire transcendante » [68], qui permet au noyau d'accéder à de la mémoire qu'il ne peut pas directement adresser. Pour cela, l'opération `put` associe à une page un identifiant unique, composé par exemple de l'UUID (*Universally Unique Identifier*) du système de fichier, du numéro d'inode et de l'index de la page, et confie la page à l'implémentation de cette API (*backend*). À l'inverse, l'opération `get` demande au *backend* la page correspondante à un identifiant unique.

Dans sa spécification, cleancache est défini comme étant « éphémère » : les données qui y sont placées ont une durée de vie inconnue et peuvent disparaître du cache à n'importe quel moment. C'est pourquoi cette API est conçue pour ne traiter que les pages *propres* du page cache, les pages *sales* étant écrites sur le disque avant d'être confiées à cette API.

3.4 Conclusion

La réalisation d'un cache réparti intégré au cœur du noyau Linux nécessite de maîtriser des mécanismes clés relatifs à la gestion des fichiers et des caches, tels que le VFS, le page cache ou le PFRA. En effet, ces sous-systèmes ne sont pas des boîtes noires dont l'accès n'est possible qu'au travers d'APIs clairement définies, mais plutôt des structures de données (LRU, arbres radix, etc.) et algorithmes (PFRA) « ouverts », qui peuvent être manipulés directement par d'autres sous-systèmes du noyau Linux.

Ce chapitre a présenté une vue d'ensemble de la gestion des fichiers et de la mémoire sous Linux. Nous nous sommes intéressés en particulier à définir les différents caches qu'on peut retrouver au sein d'un système d'exploitation et aux mécanismes de gestion et de récupération de la mémoire sur lesquels reposent nos contributions présentées dans les chapitres suivants.

Chapitre 4

PUMA : un cache distant pour mutualiser la mémoire inutilisée des machines virtuelles

Sommaire

4.1 Principes de PUMA 46

4.2 Architecture générale de PUMA 47

4.2.1 Positionnement dans la pile système 47

4.2.2 Gestion de la mémoire 49

4.2.3 Gestion des pannes et du temps de réponse 51

4.2.4 Gestion des accès séquentiels 51

4.2.5 Stratégies de cache 52

4.3 Implémentation 53

4.3.1 Stockage des métadonnées 54

4.3.2 Stockage des pages 54

4.3.3 Cohérence des caches 55

4.4 Conclusion 55

Ce chapitre présente notre contribution, PUMA [1], un cache réparti permettant de mutualiser la mémoire inutilisée des machines virtuelles pour améliorer les performances des applications qui effectuent un grand nombre d'entrées/sorties (E/S). Comparé aux caches répartis existants, PUMA a l'avantage de fonctionner avec les applications existantes, sans modifications. De plus, PUMA est directement intégré au page cache du noyau Linux, ce qui lui permet de ne pas dépendre d'un système de fichiers

1. Pour *Pooling Unused memory in virtual MAchines*

spécifique. Enfin, comme nous le verrons dans le chapitre 6, cette approche permet à un nœud ayant prêté de la mémoire au cache réparti de la récupérer efficacement.

La section 4.1 présente les objectifs de PUMA puis la section 4.2 décrit son architecture. Nous expliquons notamment les choix de conception qui permettent à PUMA de limiter son empreinte mémoire et de s'adapter au temps de réponse des nœuds. La section 4.3 décrit l'implémentation de PUMA, les structures de données utilisées et les interactions avec le noyau Linux. Nous expliquons également la méthode utilisée pour maintenir les caches de PUMA cohérents. La section 4.4 conclut ce chapitre en résumant les objectifs de PUMA, son architecture et les bénéfices attendus.

4.1 Principes de PUMA

Le principal objectif de PUMA est de mutualiser la mémoire inutilisée des machines virtuelles (VMs) pour le bénéfice d'autres VMs qui exécutent des applications qui effectuent un grand nombre d'E/S. L'une des contraintes imposées par le projet dans lequel s'inscrit cette thèse (nu@age) est de pouvoir faire *migrer* facilement les VMs d'un nœud à un autre, ce qui implique que le mécanisme de cache fourni par PUMA ne doit pas être limité aux VMs colocalisées. Pour cela, nous avons choisi une approche « classique » orientée réseau, où les nœuds PUMA communiquent entre eux via un réseau TCP. Cette approche semble raisonnable compte tenu de l'architecture cible, où les VMs disposent d'un réseau paravirtualisé performant et où les nœuds physiques sont interconnectés via un réseau à hautes performances tel que du 10 GbE ou de l'InfiniBand. Une grande partie des mécanismes présentés dans cette thèse resteraient corrects avec d'autres moyens de communication. Nous discuterons de ce point dans le chapitre 8.

Notre approche repose notamment sur l'API cleancache, présentée dans la section 3.3.3, qui permet d'implémenter un cache ne supportant que des pages propres. Nous avons choisi de conserver cette restriction dans PUMA pour les raisons suivantes :

- Si un processus qui s'exécute sur le nœud qui participe au cache a besoin d'allouer de la mémoire, les pages propres qu'il héberge peuvent être récupérées sans synchronisation.
- Les écritures sont souvent non bloquantes parce que les écritures sur le disque sont généralement différées, les possibilités d'augmentation des performances sont donc faibles. À l'inverse, lire un bloc depuis le disque est une opération bloquante et aussi lente que la latence du disque.
- Gérer des pages sales dans un cache coopératif pose des problèmes de performances parce qu'il faut traiter les problèmes de cohérence en cas de panne. Le cout inhérent d'un tel mécanisme limite l'intérêt d'une telle approche compte tenu du faible gain de performances à espérer.

En choisissant de ne traiter que les pages propres du page cache, la gestion des

pannes devient simple : en cas de panne, une version à jour des pages reste disponible depuis le disque. Cette approche permet également de récupérer rapidement la mémoire prêtée par un nœud, puisqu'il est possible d'évincer les données hébergées sans synchronisation ni risque d'incohérence, ce qui donne un avantage certain à PUMA, similaire à celui offert par XHive [55] décrit dans la section 2.5.5, mais sans les limitations liées à l'hyperviseur.

4.2 Architecture générale de PUMA

Cette section présente l'architecture générale de PUMA et les différents mécanismes que nous avons introduits afin de limiter son empreinte mémoire, d'éviter les dégradations de performances et d'utiliser au mieux le cache disponible.

4.2.1 Positionnement dans la pile système

Une des approches utilisées pour concevoir un cache collaboratif pour les environnements virtualisés consiste à proposer un périphérique bloc *virtuel* au-dessus d'un périphérique bloc *cible*. Cette approche, utilisée par exemple par XHive [55] ou CaaS [39], peut reposer sur des mécanismes génériques offerts par le système d'exploitation hôte, tels que *device-mapper* sous Linux [101]. Elle a l'avantage de ne pas nécessiter une adaptation des applications existantes. Le périphérique bloc virtuel peut ainsi capturer chaque accès au disque, c'est-à-dire chaque défaut de cache depuis le page cache. Il peut ensuite faire une recherche dans le cache collaboratif pour tenter de trouver la donnée correspondante.

Si cette approche semble simple et élégante, elle limite la portée du cache coopératif aux périphériques blocs, ce qui empêche les systèmes de fichiers distribués d'en bénéficier alors qu'ils sont largement utilisés dans le cloud. De plus, en embarquant la logique du cache au sein du périphérique bloc virtuel, on interdit à la VM d'en bénéficier si elle souhaite accéder à des périphériques blocs de façon indépendante de l'hyperviseur, par exemple via un protocole tel qu'iSCSI.

Pour toutes ces raisons, nous avons adopté une approche plus générale, qui capture directement les défauts de cache et les évictions du page cache local. Cette approche repose en partie sur l'API cleancache que nous avons présenté dans la section 3.3.3. Les figures 4.1 et 4.2 illustrent notre implémentation des opérations `get` et `put` de cette API, où VM_1 utilise la mémoire inutilisée de VM_2. Chaque défaut de cache du page cache peut entraîner une opération `get`, tandis que chaque éviction du page cache implique une opération `put`. Dans la suite, `get` et `put` font référence à notre implémentation de l'API offerte par cleancache.

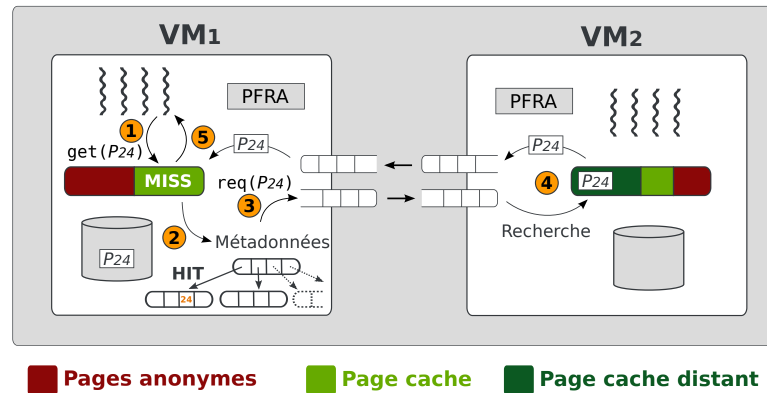


FIGURE 4.1 – Architecture de Puma : l'opération get récupère une page du cache distant

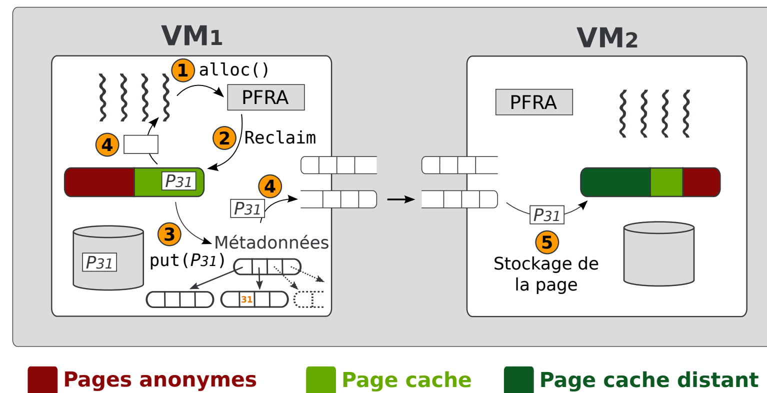


FIGURE 4.2 – Architecture de Puma : l'opération put envoie une page dans le cache distant

Opération get. Lors d'un défaut sur une page du page cache local (étape 1 de la figure 4.1), Puma vérifie dans ses métadonnées si cette page a déjà été envoyée dans le cache réparti (2). Ces métadonnées contiennent les identifiants des pages qui ont été envoyées sur un autre nœud lors d'une opération put et sont utilisées pour éviter d'envoyer des requêtes get inutiles. Lorsque l'identifiant de la page est présent dans les métadonnées, Puma envoie une requête au nœud qui l'héberge (3), qui lui renverra la page (4).

Opération put. Lorsqu'une page victime est choisie par le *Page Frame Reclaiming Algorithm* (PFRA) pour libérer de la mémoire (étapes 1 et 2 de la figure 4.2), une opération put peut être exécutée (3) si Puma décide que cela en vaut la peine (en tenant compte des mécanismes décrits dans les sections 4.2.3 et 4.2.4). Puma copie ensuite le contenu de la page dans un tampon pour que la page puisse être libérée (4). Enfin, la page contenue dans le tampon est envoyée à un autre nœud pour qu'il la stocke (5).

4.2.2 Gestion de la mémoire

Si l'API cleancache permet d'implémenter des opérations détectant les accès et les évictions du page cache, leur implémentation nécessite de tenir compte de la pression mémoire en jeu. En effet, l'opération `put` est appelée directement par le PFRA lorsqu'il tente de libérer de la mémoire. Or cette opération nécessite un minimum de mémoire, par exemple pour allouer les structures de données nécessaires à l'envoi de la page, comme des tampons réseau ou des métadonnées locales. Le traitement de cette opération doit donc éviter toute allocation de mémoire qui pourrait conduire à une nouvelle activation du PFRA, entraînant un nouvel appel à l'opération `put`, et ainsi de suite. En pratique, cela ne mènerait pas à un crash du noyau mais à un échec de l'opération `put`. La page évincée ne serait alors pas envoyée dans le cache distant, ce qui laisserait le système dans un état correct puisque PUMA n'envoie dans le cache distant que des pages propres du page cache. Cependant, il est important de limiter au maximum les échecs lors de l'envoi des pages, d'une part pour éviter le cout de traitement d'une page qui n'est finalement pas envoyée, d'autre part pour tirer le maximum de performances du cache.

Pour tenir compte de la pression mémoire et améliorer les performances globales du cache, PUMA repose sur les mécanismes décrits dans les sections suivantes.

4.2.2.1 Préallocation des messages

L'une des limites des approches existantes reposant sur cette API, telles que *zcache* [67], est qu'en cas de forte pression mémoire, notamment lors d'une d'activité intensive en E/S, le mécanisme se retrouve de fait désactivé parce qu'il devient impossible d'allouer la mémoire nécessaire au placement des données dans le cache réparti. Pourtant, il nous semble essentiel de maximiser les chances de placer les données dans le cache réparti pour les raisons suivantes :

- Les données qui ne sont pas envoyées dans le cache sont potentiellement chaudes : les futurs accès généreront des accès disque et contribueront au ralentissement des performances de l'application.
- Les accès au périphérique de stockage eux-mêmes peuvent générer de la pression mémoire. En effet, une donnée lue depuis le disque doit être placée dans une page, qui doit être allouée. De plus, même si ces accès ne doivent pas être mis en cache (accès séquentiels), ils peuvent impliquer l'éviction de pages chaudes du page cache qui ne pourront pas être placées dans le cache réparti si le mécanisme se désactive.

Avec PUMA, nous nous efforçons d'envoyer les pages dans le cache réparti lors de leur éviction afin d'augmenter les chances d'un *hit*. Ainsi, pour favoriser la réussite des opérations `put`, la mémoire nécessaire pour traiter les requêtes est allouée depuis un *pool* de mémoire pré-allouée. Grâce à cela, nous n'ajoutons pas de pression mémoire au

PFRA. Ainsi, chaque page qui doit être envoyée dans le cache est d'abord copiée dans une page provenant de ce pool, puis envoyée en *zero-copy* pour limiter la mémoire nécessaire pour l'envoi. La page est ensuite libérée lors de la réception de l'acquittement envoyé par le nœud destinataire.

Cependant, si le pool de mémoire est trop petit les pages peuvent ne pas être envoyées dans le cache réparti. Pour limiter cela, nous avons calibré la taille du pool pour que la majorité des évictions puissent être envoyées dans le cache. Combiné au mécanisme de gestion des accès séquentiels présenté dans la section 4.2.4, nous avons observé lors de nos évaluations que moins de 1% des pages n'étaient pas envoyées dans le cache réparti.

4.2.2.2 Agrégation des requêtes de la fenêtre de préchargement

Un des défauts majeurs de l'API cleancache est que sa fonction `get` doit être appelée à chaque lecture d'une page. Par exemple, cela a un impact sur algorithme de préchargement de pages du noyau Linux, qui consiste à lire en avance, et de façon asynchrone, une séquence de pages par anticipation qui pourraient être utilisées par l'application. Dans son cas, un appel bloquant à cette fonction sera effectué pour chaque page de la fenêtre de préchargement. Ce défaut n'a que peu d'importance pour les implémentations existantes de cette API telles que zcache [67], qui fonctionnent principalement localement et sont donc peu impactées par la latence. Cependant, dans le cas de Puma, chaque opération `get` implique 1 *RTT*, qui peut s'élever à plusieurs centaines de microsecondes.

Pour éviter de bloquer les processus à chaque fois qu'une page est lue, nous avons étendu l'API cleancache pour que l'opération `get` accepte qu'un groupe de pages puisse être demandé. Ainsi, nous pouvons rassembler toutes les pages de la fenêtre de préchargement au sein d'une seule opération `get`, moins couteuse. Cette approche nous permet également de profiter de l'algorithme de préchargement de pages déjà présent au sein du noyau.

4.2.2.3 Agrégation des requêtes d'envoi

Le PFRA choisit généralement plusieurs dizaines de pages victimes pour éviter que l'on fasse appel à lui trop souvent. Or la fonction `put` de l'API cleancache, qui a pour objectif de « placer » une page dans le cache, souffre du même défaut que la fonction `get` : celle-ci est appelée chaque fois qu'une page est évincée du cache par le PFRA. La difficulté ici est double :

— Envoyer chaque page indépendamment les unes des autres peut s'avérer extrêmement coûteux en raison des métadonnées associées à chaque message. En effet, il n'est pas nécessaire de « bloquer » le PFRA pendant l'envoi d'une page dans

le cache puisque celle-ci est en voie d'être libérée définitivement de la mémoire, ce qui peut être fait dans un second temps, de façon asynchrone.

— La libération d'une page du page cache par le PFRA s'effectue sous la protection d'un *spinlock* qui interdit toute opération bloquante. Il devient donc difficile d'effectuer des E/S réseau immédiatement pour placer la page dans le cache.

La solution que nous proposons est d'utiliser un tampon de messages dans lequel les pages à envoyer sont simplement placées en attente. Ainsi, nous avons étendu l'API cleancache pour permettre au PFRA d'informer l'implémentation de cette API qu'il a terminé son exécution. Ceci permet de vider le tampon et d'envoyer simultanément plusieurs messages `put`. On arrive ainsi non seulement à réduire l'empreinte mémoire en évitant la multiplication de petits messages, mais aussi à améliorer la latence car les E/S réseau sont effectuées en dehors de la section critique du PFRA.

4.2.3 Gestion des pannes et du temps de réponse

Si le fait de se limiter à un cache en lecture a réduit les problèmes de cohérence liés aux pannes, celles-ci peuvent tout de même dégrader les performances. Ainsi, une panne d'un nœud ou un ralentissement du réseau peut conduire le noyau à rester bloqué indéfiniment ou dégrader les performances.

Puisque PUMA repose sur un faible temps de réponse, il monitore la latence réseau entre les nœuds pour éviter de dégrader les performances. En cas d'augmentation de la latence réseau, un nœud PUMA peut arrêter d'utiliser le cache réparti pour basculer sur des accès disque classiques. Pour cela, les nœuds PUMA échangent périodiquement des messages (*ping*) et calculent la moyenne glissante de la latence *courte* L_{short} (les 15 dernières secondes) et longue L_{long}. La première permet de détecter un pic de latence tandis que la seconde permet de mesurer le temps de réponse moyen.

Nous définissons deux seuils S_{long} et S_{short}. Lorsque l'une de ces moyennes dépasse l'un de ces seuils, le nœud PUMA arrête d'envoyer des messages `put` et `get`. Lorsque le temps de réponse repasse en dessous d'un autre seuil, le nœud PUMA recommence à envoyer des messages `put` ou `get`. Le seuil de désactivation de PUMA et différent du seuil d'activation, ce qui permet de créer un cycle d'hystérésis. Les valeurs de S_{short} et de S_{long} sont choisies de façon expérimentale en fonction du système cible, nous les avons fixées à $S_{short} = [1.5ms, 40ms]$ et $S_{long} = [1ms, 1.5ms]$ lors de notre évaluation (section 5.5).

4.2.4 Gestion des accès séquentiels

Les accès séquentiels sont particulièrement difficiles à accélérer. En effet, les disques durs actuels peuvent offrir une bande passante de plusieurs centaines de Mo/s, ce qui

peut être plus important que la bande passante réseau utilisable par PUMA. De plus, les accès séquentiels sont généralement anticipés (*prefetching*), ce qui implique que la latence des accès disque est amortie. Enfin, certains accès séquentiels, tels que des flux vidéo, tiennent difficilement dans le cache. Les y placer expulserai alors d'autres données, potentiellement plus importantes. Pire, un tel flux peut « s'expulser » lui même du cache s'il est trop grand (*thrashing*), ce qui dégraderait considérablement les performances du système.

Dans cette situation, PUMA pourrait ralentir les performances d'une application qui effectue beaucoup d'accès séquentiels. Pour limiter ce risque, nous avons introduit une option *filtre* à PUMA qui permet de filtrer les accès séquentiels pour que ceux-ci ne soient pas envoyés dans le cache distant.

Lorsque cette option est activée, PUMA détecte les accès séquentiels en provenance du disque et marque les pages correspondantes de sorte que, lorsque celles-ci sont évincées du cache local, PUMA ne les envoie pas dans le cache distant. L'intérêt de l'option *filtre* est multiple.

- **Limitation des envois** : les pages marquées ne sont pas envoyées dans le cache distant, ce qui veut dire que nous évitons le surcout (CPU et réseau) d'une opération `put`.
- **Limitation des accès** : lors d'un second accès à ces pages, elles ne sont pas présentes dans le cache réparti (puisque le filtre l'a empêché), ce qui permet à PUMA de ne pas ralentir l'application en allant chercher ces pages depuis un autre nœud. En revanche, si le second accès n'est pas séquentiel, les pages ne sont pas marquées et pourront être envoyées dans le cache réparti lors de leur éviction.
- **Optimisation de la taille du cache** : ne pas envoyer les pages accédées séquentiellement permet d'économiser de la place dans le cache distant, et donc de permettre à d'autres types d'accès plus lents de tenir dans le cache.

L'option *filtre* est évaluée en détails dans les sections 5.3 et 5.4.1. Notons que dans certaines circonstances, par exemple lorsque la bande passante du disque est faible, cette option peut être désactivée par l'administrateur pour que tous les types d'accès (y compris séquentiels) puissent reposer sur le cache réparti.

4.2.5 Stratégies de cache

Nous avons vu dans la section 2.2.3 que la stratégie de cache adoptée est une propriété déterminante [25, 110, 113]. Dans le cas de PUMA, nous avons choisi d'implémenter deux stratégies de cache réputées, exclusive (stricte) et non-inclusive, afin d'étudier leurs performances (chapitre 5). Cette section rappelle les principes de ces deux stratégies de cache et détaille leur implémentation au sein de PUMA.

4.2.5.1 Exclusive

Un cache exclusif a l'avantage de proposer plus de capacité de cache qu'un cache inclusif [110] : le cache réparti est utilisé *en supplément* du cache local, la capacité totale de cache est donc la somme du cache local et du cache réparti. L'implémentation d'une telle stratégie de cache dans PUMA est relativement simple : lorsqu'un nœud demande une page à un autre nœud du cache, ce dernier la retire de sa mémoire pour garantir la propriété d'exclusivité. Le nœud qui reçoit la page mettra à jour ses métadonnées lors de la réception pour en tenir compte.

La simplicité et l'optimisation de l'utilisation de la mémoire s'accompagnent néanmoins d'un surcout. En effet, un des défauts de cette stratégie est qu'un nœud peut envoyer la même page à plusieurs reprises dans le cache réparti, ce qui peut devenir problématique, d'une part parce que cela implique un cout d'envoi (CPU et réseau), et d'autre part parce que nous avons vu qu'il est difficile de contrôler la pression mémoire (décrite dans la section 4.2.2).

4.2.5.2 Non-inclusive

Si une stratégie de cache *strictement inclusive* a le défaut de limiter considérablement la quantité de cache disponible, une stratégie *non-inclusive* [50, 114] permet de relâcher la propriété d'inclusivité. Une stratégie de cache non-inclusive est donc une stratégie de cache où la propriété d'inclusion n'est pas garantie. Ainsi, la taille totale du cache disponible est plus proche d'une stratégie exclusive, alors qu'avec une stratégie de cache strictement inclusive elle serait du *max* entre la taille du page cache et celle du cache réparti.

L'implémentation de cette stratégie dans PUMA a pour objectif de réduire la charge des nœuds et l'utilisation du réseau. Pour cela, un nœud qui reçoit une requête `get` conserve les pages concernées en mémoire même après les avoir envoyées. Ainsi, les pages chaudes restent dans le cache réparti, ce qui permet d'éviter aux nœuds d'avoir à renvoyer ces pages dans le cache réparti si elles sont de nouveau choisies comme victimes. Cependant, cette stratégie peut entraîner des problèmes de cohérence que nous décrivons dans la section 4.3.3.

4.3 Implémentation

Nous avons implémenté PUMA dans le noyau Linux 3.15.10. L'essentiel de notre implémentation est contenue dans environ 8000 lignes de code isolées au sein d'un module. Des modifications mineures du cœur du noyau Linux ont été nécessaires et

concernent environ 500 lignes de code localisées principalement dans les sous-systèmes VFS et *mm* (*Memory Management*).

4.3.1 Stockage des métadonnées

Chaque nœud PUMA doit conserver pour chaque page envoyée dans le cache réparti des métadonnées permettant de la localiser et de gérer les problèmes de cohérence (décrits dans la section 4.3.3). Ces métadonnées incluent quelques bits (*present*, *busy*) qui permettent de savoir si une page est présente dans le cache ou si elle est en cours d'envoi. D'autres bits peuvent être utilisés pour déterminer le nœud PUMA où la page a été envoyée précédemment.

Afin de limiter le surcout de ces métadonnées, nous avons utilisé un arbre radix, dont l'API est fournie par le noyau Linux et est déjà utilisé pour le stockage des pages du page cache local (voir section 3.1). L'intérêt de cette structure de données réside dans sa compacité. En effet, si dans le cas du page cache celle-ci est utilisée pour effectuer une correspondance entre un index dans le fichier et une structure `page` (pointeur), dans notre cas, nous embarquons directement les métadonnées dont nous avons besoin dans l'espace réservé au pointeur.

Concrètement, pour chaque page (4 ko) stockée dans un autre nœud PUMA, un nœud doit conserver seulement quelques bits embarqués au sein d'un seul entier de 64 bits, ce qui veut dire que nous avons besoin de seulement 2 Mo de mémoire (complexité spatiale amortie) sur un nœud pour gérer 1 Go de données envoyées à d'autres nœuds.

Notons pour finir qu'un nœud a toujours la possibilité de réclamer la mémoire utilisée pour les métadonnées ; dans ce cas il lui suffit d'invalider les pages correspondantes hébergées sur d'autres nœuds.

4.3.2 Stockage des pages

Un nœud PUMA *héberge* des pages du cache réparti dans un arbre radix, de façon similaire au page cache existant du noyau Linux. Les pages sont également placées directement au sein des listes LRU du noyau Linux, ce qui permet de reposer sur le PFRA existant pour réclamer ces pages.

Une évolution de cette implémentation présentée dans le chapitre 6 consiste à ne placer ces pages que dans la liste inactive d'un nœud qui héberge les pages, et de ne jamais les promouvoir dans la liste active : un processus qui nécessite de la mémoire réclamera donc ces pages plus rapidement que les pages locales du page cache.

4.3.3 Cohérence des caches

Les nœuds Puma agissent uniquement sur des pages propres, ce qui facilite la gestion de la cohérence des données, notamment en cas de panne. Cependant, dans le cas d'une stratégie non-inclusive, nous n'avons aucun moyen de savoir si une page présente dans le cache réparti a été modifiée par son propriétaire. En effet, si le noyau Linux est capable d'identifier les pages sales à l'aide du bit *dirty* qui leur est associé, elles sont écrites sur le disque avant leur éviction, et donc avant que l'opération `put` ne soit appelée, ce qui ne les rend pas identifiable par Puma. Ce problème est connu sous le nom du « problème du ABA » [44] : une variable est lue deux fois et a la même valeur (A) ; la conclusion est que l'état du système n'a pas changé, alors qu'entre les deux lectures un processus concurrent a modifié la variable deux fois (B puis A). Ici, nous pouvons observer l'état du bit *dirty*, mais il peut changer d'état plusieurs fois entre deux observations.

Ainsi, un nœud peut héberger une ancienne version d'une page modifiée, il est donc *nécessaire* de la mettre à jour ou de l'invalider. Pour régler ce problème, nous avons ajouté aux bits existants des pages un nouveau bit *dirtied*, qui est positionné à chaque fois qu'une page est modifiée et est remis à zéro par Puma. Nous pouvons ensuite contrôler la présence de ce bit lors de l'éviction d'une page par Puma pour savoir si celle-ci doit être renvoyée dans le cache réparti ou non.

Une seconde cause possible pouvant entraîner des problèmes de cohérence est la mise en tampon des pages avant leur envoi. En effet, une page modifiée puis évincée peut être placée dans le tampon alors qu'un processus tente de la lire (via une opération `get`). Ceci peut conduire à un scénario où la requête pour la page atteint le nœud (qui stocke potentiellement une ancienne version de la page) avant la nouvelle version. Nous résolvons ce problème de concurrence en ajoutant un bit de synchronisation (*busy*) aux métadonnées de Puma. Ainsi, dans cette situation le processus est bloqué en attendant que l'envoi de la page dans le cache réparti soit terminé.

4.4 Conclusion

Ce chapitre a présenté Puma, un système de cache réparti générique capable de mutualiser la mémoire inutilisée dans les environnements virtualisés. Pour rendre ce mécanisme efficace et limiter son surcout, nous l'avons conçu pour ne gérer que les pages *propres* du page cache qui peuvent être réclamées sans aucune synchronisation. Puma repose sur des mécanismes existants du noyau Linux, tels que son *page cache*, ses LRUs et ses APIs du système de fichiers virtuel (VFS), ce qui lui permet d'être agnostique aux périphériques blocs, aux systèmes de fichiers et aux hyperviseurs. De plus, il peut fonctionner à la fois avec des VMs locales, en bénéficiant des techniques de

paravirtualisation des réseaux, et distantes. Le chapitre suivant présente une évaluation exhaustive des performances et des différents mécanismes présentés dans ce chapitre.

Chapitre 5

Évaluation de Puma

Sommaire

5.1 Protocole expérimental **58**
5.1.1 Benchmarks 58
5.1.2 Paramètres 61
5.1.3 Métriques 62
5.2 Lectures aléatoires : de l'intérêt des stratégies de cache **64**
5.3 Accès séquentiels : de l'intérêt du filtre **66**
5.4 Performances des benchmarks applicatifs **67**
5.4.1 BLAST : une charge partiellement séquentielle 67
5.4.2 Postmark : expérimentations en présence d'écritures 68
5.4.3 Bases de données 70
5.5 Analyse de la sensibilité au temps de réponse **71**
5.6 Comparaison avec un cache sur un SSD **73**
5.7 Étude de cas : le cloud privé **74**
5.8 Conclusion **76**

Ce chapitre présente les résultats des expérimentations que nous avons effectuées avec Puma. Nous commençons par présenter la plateforme expérimentale et les benchmarks que nous utilisons, et nous définissons les métriques sur lesquelles repose une partie de notre évaluation (section 5.1). Nous présentons nos évaluations des performances des différentes stratégies de cache dans la section 5.2, puis nous montrons l'intérêt du filtrage des accès séquentiels dans la section 5.3. La section 5.4 montre les gains de performances observés avec différentes applications. La section 5.5 étudie le comportement de Puma lorsque la latence réseau est dégradée. Nous comparons les performances de Puma avec les performances d'un cache SSD dans la section 5.6, puis

nous montrons dans la section 5.7 comment PUMA peut être utilisé dans un contexte de cloud privé avec plusieurs machines physiques.

5.1 Protocole expérimental

Les expérimentations présentées dans ce chapitre ont été effectuées sur les nœuds du cluster *Paranoia* de la plateforme Grid'5000 [17] (site de Rennes). Chacun de ces nœuds est équipé de 2 processeurs Intel Xeon E5-2660v2 (4 cœurs avec hyper-threading), de 128 Go de mémoire et d'une carte Ethernet 10 Gb/s. Ils disposent aussi de 5 disques SAS de 600 Go, que nous avons configurés en RAID 0 afin d'obtenir des performances maximales.

Les benchmarks ont été déployés dans des machines virtuelles (VMs) reposant sur l'hyperviseur KVM [56] avec la version 1.7.50 de QEMU. Nous avons configuré les VMs en suivant les recommandations de fournies par IBM [47]. Ainsi, tous les disques virtuels sont paramétrés pour contourner le *page cache* du système hôte pour éviter le « double cache » (hôte et invité). Nous utilisons également l'ordonnanceur d'entrées/sorties (E/S) *deadline*, qui offre de meilleurs performances que d'autres ordonnanceurs (tel que CFQ, par défaut sous Linux) et des garanties sur le temps de réponse des requêtes d'E/S. Nous avons installé le système de fichiers *ext4* à la fois dans l'hôte et dans les VMs. Les VMs possèdent 2 vCPU et une quantité de mémoire variable. (voir section 5.1.3). Une image disque dédiée est utilisée pour le stockage des données des expériences. Nous utilisons le framework de paravirtualization *VirtIO* [84] pour améliorer l'efficacité des E/S (disque et réseau) et ainsi obtenir des performances proches des standards industriels.

Pour déployer nos expériences, nous avons utilisé la plateforme d'expérimentations MOSBENCH [19] que nous avons modifiée pour permettre l'exécution des expériences dans des VMs. Chaque expérimentation est effectuée sur une VM dédiée après une phase de chauffe suffisamment longue pour que l'ensemble des caches soient entièrement remplis. Les expériences sont reproduites 10 fois, puis nous calculons la moyenne et un intervalle de confiance à 95% en utilisant la distribution t de Student. Nous avons systématiquement observé un faible écart type, c'est pourquoi nous avons décidé de ne pas le représenter sur les figures afin d'améliorer leur lisibilité.

5.1.1 Benchmarks

L'évaluation d'un système tel qu'un cache réparti ou un système de fichiers est une tâche complexe et les biais peuvent être multiples. Par exemple, Traeger et al. ont montré que de nombreuses évaluations de systèmes de fichiers reposent sur un benchmark de compilation [99], du noyau Linux par exemple, alors que cette tâche est surtout

Benchmark	Type d'accès			
	Aléatoires	Séquentiels	Lectures	Écritures
Lectures aléatoires	+++	∅	+++	∅
Lectures séquentielles	∅	+++	+++	∅
BLAST	−	++	+++	∅
Postmark	++	−	−	++
TPC-C	++	−	++	+
TPC-H	+	+	++	−

TABLE 5.1 – Synthèse des benchmarks utilisés pour l'évaluation de PUMA

limitée par la puissance de calcul disponible (CPU). Tarasov et al. ont poursuivis ces travaux, et montrent qu'il n'existe pas de benchmark standard pour évaluer un système de fichiers [96]. Pour ces raisons, nous avons choisi plusieurs benchmarks et applications pour mesurer les différents aspects de PUMA. Une synthèse des benchmarks que nous utilisons est présentée dans le tableau 5.1.

5.1.1.1 Microbenchmarks

Filebench [70] est un générateur de charge d'E/S qui peut être configuré via un langage de modélisation de charge. Il comporte plusieurs types de charges prédéfinies qui peuvent être facilement modifiées. Parmi ces charges, nous utilisons les microbenchmarks de lectures aléatoires (section 5.2) et séquentielles (section 5.3) pour évaluer PUMA. Nous préférons utiliser les autres applications décrites plus loin dans cette section, plus réalistes, plutôt que les autres générateurs fournis avec Filebench. Nous utilisons la version 1.4.9.1 de Filebench.

Lectures aléatoires. Elles sont effectuées par un unique thread à des positions aléatoires au sein d'un fichier. Nous l'avons configuré pour utiliser un fichier de 4 Go (contigu sur le disque), en effectuant des lectures de 4 ko pendant 10 minutes. Les expériences effectuées avec des lectures aléatoires dépendent principalement de la latence des E/S. Filebench mesure le nombre moyen d'E/S complétées par seconde.

Lectures séquentielles. Nous utilisons un fichier de 4 Go qui est lu séquentiellement, en boucle, pendant 10 minutes. Les lectures séquentielles sont peu sensibles à la latence mais dépendent fortement de la bande passante disponible et de l'efficacité de l'algorithme de préchargement du noyau. Nous mesurons le débit moyen obtenu pendant la durée de l'expérience (Mo/s).

5.1.1.2 Applications

Applications scientifiques. BLAST [3] (Basic Local Alignment Search Tool) est un outil utilisé par les chercheurs en bio-informatique pour trouver les régions similaires entre une séquence donnée et une base de données de séquences de nucléotides ou d'acides aminés. BLAST fonctionne essentiellement en parcourant la base de données pour trouver des sous-séquences similaires à la requête. Cet outil génère une grande quantité d'E/S, dont une large partie est séquentielle. Pour nos expérimentations, nous avons utilisé la base de données *patnt* qui a une taille d'environ 3 Go. Il a été montré que 90% des requêtes soumises par les chercheurs en bio-informatique comprenaient entre 300 et 600 caractères [82], nous avons choisi d'extraire aléatoirement 5 séquences de 600 caractères de la base de données, qui nous servent de requêtes pour nos expériences.

Applications intensives en écritures. Postmark [53] est un benchmark qui mesure la performance d'un système de fichiers en générant une charge typique d'applications de type serveur d'e-mails. Postmark génère des opérations à la fois sur les données et les métadonnées des fichiers, et est composé de multiples petites écritures. Il définit une *transaction* comme étant une lecture ou un ajout à un fichier existant suivi de la création ou de la suppression d'un fichier.

Nous utilisons la version 1.51 de Postmark, avec une configuration où plus de 80% des E/S sont des écritures. Rappelons que PUMA est un cache réparti en lecture seule, c'est donc une configuration défavorable pour PUMA puisque ces écritures doivent d'abord être synchronisées sur le périphérique de stockage avant de pouvoir être envoyées dans le cache réparti. Nous avons configuré Postmark pour générer 20 000 transactions sur 25 000 fichiers. Chaque fichier a une taille choisie aléatoirement entre 512 octets et 64 ko, ce qui nous donne un total d'environ 3,5 Go de données. Postmark est configuré pour générer d'une part le même ratio de lectures/ajouts et d'autre part le même ratio de créations/suppressions.

Bases de données. Les benchmarks de traitement transactionnel en ligne (OLTP) mesurent la capacité d'un système à traiter des transactions qui mêlent accès et mises à jour de données en temps réel. Ces systèmes sont utilisés par exemple pour les réservations de billets pour des compagnies aériennes ou les systèmes de paiements bancaires. Pour nos évaluations, nous avons utilisé les benchmarks TPC-C [98] et TPC-H [97] pour évaluer PUMA avec une charge typique de serveurs de base de données. Ces deux benchmarks ont été déployés sur la version 9.3.1 de PostgreSQL avec une implémentation open source de TPC-C [109] et de TPC-H [115].

TPC-H définit un ensemble de 22 transactions complexes, dont la majorité est en lecture seule. Il génère un type de charge que l'on retrouve dans les grandes entreprises

qui analysent une grande quantité de données, typique de l'informatique décisionnelle. Nous l'avons configuré avec un facteur d'échelle de 3, ce qui est équivalent à un jeu de données d'environ 3 Go. TPC-H mesure un débit en termes de nombre de requêtes complétées par heure.

TPC-C simule un environnement de traitement transactionnel en ligne (OLTP) typique d'entreprises de commerce électronique. Environ $2/3$ des E/S générées par ce benchmark sont des lectures, et $1/3$ sont des écritures. Nous l'avons configuré pour qu'il définisse 40 entrepôts, ce qui est équivalent à un jeu de données d'environ 4 Go. TPC-C mesure le nombre de transactions « New-Order » exécutées par minute et leur temps de réponse. Dans notre évaluation, nous avons uniquement retenu le temps de réponse (9e décile) car nous pensons que c'est cette métrique qui permet de mieux mesurer l'expérience ressentie par l'utilisateur de bout en bout.

5.1.2 Paramètres

Cette section présente les différents paramètres, de Puma et de notre plateforme, que nous faisons varier dans nos évaluations.

Taille du cache. Un paramètre important lorsque l'on évalue un cache est sa taille : s'il est suffisamment grand pour que toutes les données puissent y être placées en même temps, on évaluera alors l'efficacité des mécanismes permettant d'y accéder. Inversement, s'il est trop petit on évaluera les différents mécanismes permettant d'optimiser le choix des données à placer dans le cache. Pour faire varier la taille du cache qu'offre Puma, nous utilisons une VM qui offre une quantité de cache différente à chaque expérience.

Ainsi, nos expérimentations utilisent 2 VMs : VM_1, où les benchmarks sont exécutés, qui possède 1 Go de mémoire ; et VM_2, qui offre sa mémoire inutilisée à VM_1 via Puma. Nous exécutons également ces expériences au sein d'un VM seule (*référence*) qui possède, elle, 1 Go de mémoire mais n'utilise pas Puma. C'est cette *référence* qui nous permet de calculer l'accélération ou le ralentissement obtenu avec Puma (voir section 5.1.3.1. Dans ce chapitre, nous nous intéressons à la performance du benchmark qui s'exécute sur VM_1.

Stratégie de cache. Les différentes stratégies de cache implémentées dans Puma (non-inclusive et exclusive) sont présentées dans la section 5.2. Cependant, toutes les expériences sont produites avec ces deux stratégies à des fins de comparaison.

Filtre des accès séquentiels. L'option *filtre* de Puma, permettant de filtrer les accès séquentiels, est analysée en détails dans la section 5.3. Cependant, afin d'étudier l'effet

de cette option dans d'autres contextes, toutes les expériences sont effectuées avec et sans celle-ci afin de comparer les résultats.

Résumé. Dans nos expériences nous évaluons PUMA en faisant varier la quantité de cache offerte par une VM (VM_2). Ainsi, l'axe des abscisses de la plupart des courbes présentées ici représente la taille totale de cache disponible (page cache local + cache réparti). Nous faisons également varier la stratégie de cache et l'activation de l'option *filtre* de PUMA, ainsi les courbes seront présentées comme suit :
- stratégie de cache exclusive, avec filtrage des accès séquentiels ;
- stratégie de cache exclusive, sans filtrage des accès séquentiels ;
- stratégie de cache non-inclusive, avec filtrage des accès séquentiels ;
- stratégie de cache non-inclusive, sans filtrage des accès séquentiels.

5.1.3 Métriques

Nous avons retenu différentes métriques nous permettant d'étudier PUMA sous plusieurs angles. Parmi celles-ci, nous utilisons des métriques *applicatives* pour mesurer les performances des applications, ainsi que des métriques relevant d'informations fournies par PUMA ou par le système, qui nous permettent de comprendre certains scénarios.

5.1.3.1 Métriques applicatives

Dans l'ensemble de nos expériences, basées sur les applications décrites précédemment, nous mesurons les performances en faisant varier la taille d'une VM *inactive* (VM_2) qui offre sa mémoire inutilisée à une autre VM *active* (VM_1) à l'aide de PUMA. Ensuite, nous calculons l'*accélération* obtenue par rapport à une VM seule, sans cache additionnel (*référence*).

Le calcul de l'accélération dépend de la métrique retenue pour chaque application, qui peut être une durée (temps d'exécution, latence, etc.), ou un indice de performance (transactions, ES/s, etc.). Pour une durée T, le calcul de l'accélération A est le suivant :

$$A = \frac{T_{référence}}{T_{active}} \Bigg| si\ T_{active} < T_{référence}$$

Avec cette métrique, une valeur supérieure à 1 indique une amélioration des performances. Par exemple, 1,25 indique une augmentation des performances de +25%, et une valeur de 2 indique une multiplication des performances par 2 (+100%). Une valeur inférieure à 1 indique une dégradation des performances. Cependant, cette métrique nous semble difficilement compréhensible, c'est pourquoi nous préférons calculer le

ralentissement R :

$$R = \frac{T_{active} - T_{référence}}{T_{référence}}\bigg|\, si\ T_{active} > T_{référence}$$

Cette métrique permet d'indiquer à quel point l'application est ralentie. Par exemple, si le ralentissement est égal à 1, cela veut dire que le temps d'exécution a été doublé (+100%), si cette valeur est égale à 0,25, le temps d'exécution a été augmenté de +25%. Dans le reste de l'évaluation, on dira que la performance a été diminuée (÷2, −25%, etc.).

Pour un indice de performances, les calculs de l'accélération et du ralentissement sont les suivants :

$$A = \frac{P_{active}}{P_{référence}}\bigg|\, si\ P_{active} > P_{référence}$$

$$R = \frac{P_{référence} - P_{active}}{P_{active}}\bigg|\, si\ P_{active} < P_{référence}$$

5.1.3.2 Métriques de Puma

Certaines de nos expériences nécessitent d'analyser certaines métriques internes de Puma. Parmi celles-ci, nous étudierons le nombre de `put` et le nombre de `get`.

Nombre de get/io. Le taux de *hit* est une métrique importante pour un cache. Cependant, le mesurer précisément sans impacter les performances est difficile : un surcout non négligeable serait alors ajouté lors de chaque accès au page cache local. Pour nous en approcher, nous mesurons le nombre de pages récupérées depuis le cache réparti divisé par le nombre de lectures effectuées par l'application. Ici, le nombre de lectures correspond au nombre d'appels systèmes `read` effectués. Cette métrique permet d'estimer le taux de *hit* du cache réparti : la valeur get/io augmente proportionnellement au taux de hit.

Nombre de put/io**.** De façon similaire, nous mesurons le nombre de pages envoyées dans le cache distant en fonction du nombre d'appels système `read` effectués. Cette métrique est intéressante pour mesurer l'efficacité des stratégies de cache de Puma : plus cette valeur est élevée, plus il est nécessaire d'envoyer une page (locale) évincée dans le cache réparti. Inversement, une valeur faible traduit le fait qu'une page (locale) évincée n'a pas besoin d'être envoyée dans le cache réparti car déjà présente, ce qui limite l'encombrement réseau généré par Puma.

5.1.3.3 Métriques système : accès aux disques

Lors de nos expériences, nous monitorons un grand nombre de paramètres du système qui nous permettent d'analyser les résultats. Dans cette thèse, nous avons isolé deux paramètres importants que nous utilisons dans la section 5.4.2.

Nombre de lectures. Nous mesurons le nombre de requêtes de lecture soumises au périphérique bloc. Cette valeur correspond aux lectures qui finissent effectivement sur le disque : un accès à une page qui est présente dans le cache local ou réparti n'augmente pas cette valeur.

Nombre d'écritures. Nous mesurons également le nombre de requêtes d'écriture soumises au périphérique bloc. De même, cette valeur correspond aux écritures qui finissent sur le disque : une écriture différée incrémentera cette valeur uniquement lors de l'écriture *effective* des pages correspondantes sur le disque.

5.2 Lectures aléatoires : de l'intérêt des stratégies de cache

Dans cette section, nous proposons d'étudier les effets des différentes stratégies de cache implémentées au sein de PUMA à l'aide du microbenchmark de lectures aléatoires. Les résultats de cette expérience sont présentés dans la figure 5.1. Avec des accès aléatoires, le taux de hit devrait augmenter de façon linéaire avec la quantité de cache disponible. Cependant, on remarque que les performances obtenues avec ce benchmark augmentent de façon exponentielle. Ceci s'explique par le fait que les performances du cache dépendent de la vitesse des accès au périphérique de stockage « lent » (lors d'un défaut de cache) : si le temps de réponse moyen d'un accès au disque est de plusieurs millisecondes, le temps de réponse moyen que nous avons mesuré avec PUMA est de 200μs. Ainsi, un faible taux de défauts de cache est suffisant pour écrouler les performances.

Pour expliquer les différences entre les stratégies de cache exclusives et non-inclusives, nous mesurons le nombre d'opérations `get` et `put` par E/S effectuées par le benchmark de lectures aléatoires. La figure 5.2a montre le nombre d'opérations `get` par E/S effectuées par le benchmark de lectures aléatoires. Comme nous nous y attendions, le nombre de pages récupérées depuis le cache distant augmente linéairement avec la quantité de mémoire disponible pour le cache (et donc, avec le taux de hit). On remarque que les stratégies exclusives effectuent plus d'opérations `get` (et donc plus de *hits distants*) que les stratégies inclusives, et ce jusqu'à ce que la totalité du jeu de

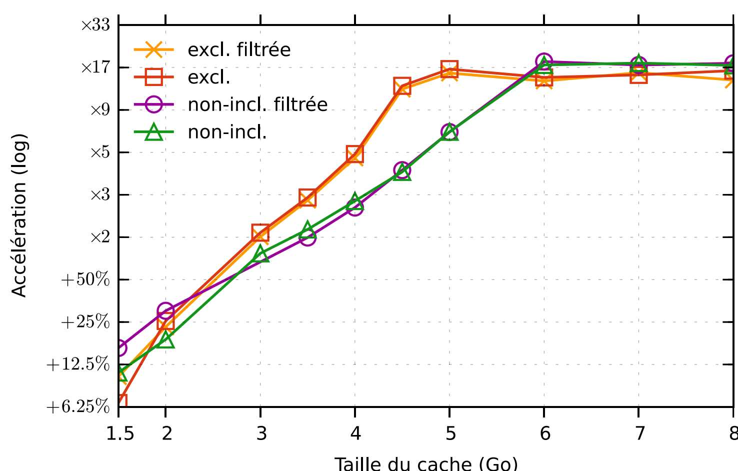
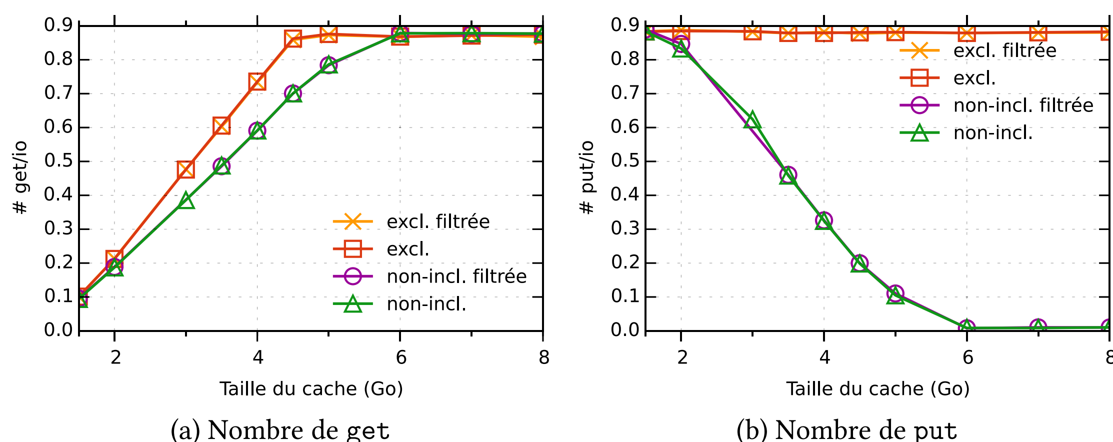


FIGURE 5.1 – Performance des accès aléatoires



(a) Nombre de `get` (b) Nombre de `put`

FIGURE 5.2 – Nombre de `get`/`put` pour le benchmark de lectures aléatoires

données tienne dans le cache. Ce résultat est cohérent avec la mesure de performance présentée dans la figure 5.1, où les stratégies exclusives sont plus efficaces que les stratégies non-inclusives parce qu'elles bénéficient d'un cache plus grand.

La figure 5.2b montre le nombre d'opérations `put` par E/S effectuées par le benchmark de lectures aléatoires. Avec une stratégie de cache non-inclusive, on remarque que le nombre de pages envoyées dans le cache décroît en même temps que la taille du cache augmente. À l'inverse, avec une stratégie de cache exclusive le nombre d'opérations `put` est constant. Ces résultats sont cohérents avec le fonctionnement des stratégies de cache décrites dans la section 4.2.5 : avec une stratégie exclusive, une page lue depuis le cache est retirée, et doit donc être renvoyée lors de son éviction. À l'inverse, avec une stratégie non-inclusive, la page reste dans le cache et n'a pas besoin d'être renvoyée.

Ici, l'intérêt d'une stratégie non-inclusive peut être observé lorsque le jeu de don-

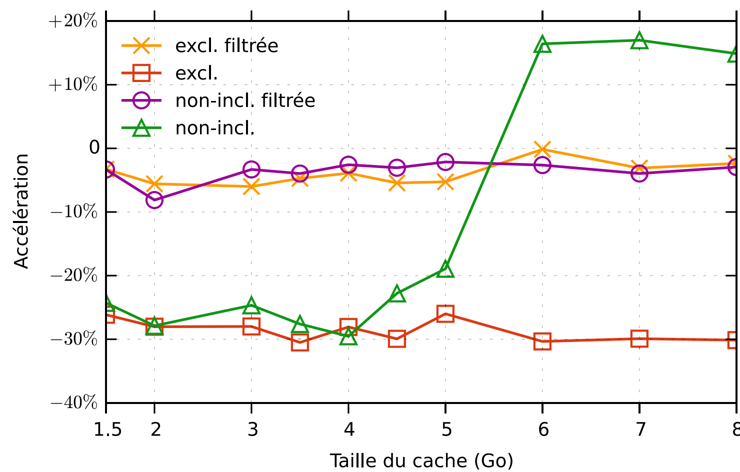


FIGURE 5.3 – Performance des accès séquentiels

nées tient entièrement dans le cache. Sur la figure 5.1, on remarque que, lorsque le fichier utilisé tient entièrement dans le cache, les stratégies de cache non-inclusives sont plus efficaces que les stratégies de cache exclusives, ce qui illustre le surcout de l'opération put de PUMA. Enfin, ces figures montrent que dans le cas de lectures aléatoires le filtre d'accès séquentiels (présenté dans la section 4.2.4) n'a pas d'impact.

5.3 Accès séquentiels : de l'intérêt du filtre

Dans cette section, nous étudions les performances de PUMA en présence d'accès séquentiels, et nous montrons qu'avec l'option *filtre*, décrite dans la section 4.2.4, PUMA est capable de détecter les accès séquentiels pour éviter d'avoir un impact négatif sur les performances.

Avec l'utilisation de disques durs performants, il devient particulièrement difficile d'améliorer les performances d'une charge composée en majorité d'accès séquentiels. En effet, ces types d'accès sont déjà largement optimisés par le système à l'aide d'algorithmes de préchargement (*readahead*), les performances ne sont alors limitées que par la bande passante disponible. Or, dans une configuration utilisant plusieurs disques en parallèle, telle que le RAID 0, cette bande passante se retrouve démultipliée par le nombre de disques utilisés ce qui rend d'autant plus difficile l'amélioration des performances des accès séquentiels.

La figure 5.3 montre l'accélération mesurée avec PUMA et des accès séquentiels. Comme nous nous y attendions, avec les configurations sans *filtre* PUMA dégrade les performances lorsque le jeu de données ne tient pas dans le cache (−25%). En effet, les pages sont systématiquement évincées du cache avant de pouvoir être touchées, il ne reste donc que le cout de PUMA. Cependant, avec le filtre nous détectons les accès

séquentiels et le nombre de pages envoyées dans le cache réparti est réduit à zéro, PUMA ne génère donc pas de surcout.

Lorsque le jeu de données tient dans le cache, la stratégie de cache exclusive dégrade toujours les performances et la moitié de la bande passante réseau disponible est utilisée pour *envoyer* les pages dans le cache de PUMA. Avec la stratégie non-inclusive, lorsque le jeu de données tient dans le cache, les pages lues restent dans le cache, la totalité de la bande passante réseau peut donc être utilisée pour récupérer des pages du cache et améliorer les performances. Mais dans ce cas, le gain est très réduit, de +20% contre ×35 avec des lectures aléatoires, car les accès séquentiels au disque sont déjà très optimisés.

Lorsque l'on active l'option *filtre* de PUMA, les pages lues séquentiellement depuis le disque ne sont pas envoyées dans PUMA. Ces accès ne sont donc pas optimisés, mais nous évitons de dégrader les performances.

5.4 Performances des benchmarks applicatifs

Cette section présente une analyse des performances obtenues avec des macro-benchmarks et de vraies applications. La section 5.4.1 présente les résultats obtenus avec BLAST, dont la particularité est d'avoir une charge principalement séquentielle. La section 5.4.2 étudie le comportement de PUMA en présence d'importantes écritures à l'aide de Postmark. Enfin, la section 5.4.3 présente les performances obtenues avec PUMA avec une charge générée par un serveur de bases de données.

5.4.1 BLAST : une charge partiellement séquentielle

Nous avons vu qu'il est difficile d'améliorer les performances d'une charge intensive en E/S séquentielles, notamment à cause de la latence du disque qui s'en retrouve masquée par le mécanisme de préchargement. La figure 5.4 montre l'accélération obtenue avec BLAST en utilisant PUMA. Comme nous nous y attendions, PUMA dégrade les performances lorsque l'option *filtre* n'est pas activée et que la base de données de BLAST ne tient pas en mémoire. Lorsque la base de données tient dans le cache, PUMA est capable d'améliorer les performances de BLAST jusqu'à +30% avec une stratégie de cache non-inclusive.

Étonnamment, lorsque que nous activons l'option *filtre* de PUMA, les performances sont améliorées de +45% avec une stratégie de cache non-inclusive, et de +35% avec une stratégie exclusive. En effet, comme expliqué dans la section 4.2.4, le bénéfice de l'option *filtre* est double : (*i*) l'application n'est pas ralentie par la bande passante réseau, plus lente que la bande passante du disque dur, et (*ii*) le cache réparti est consacré aux

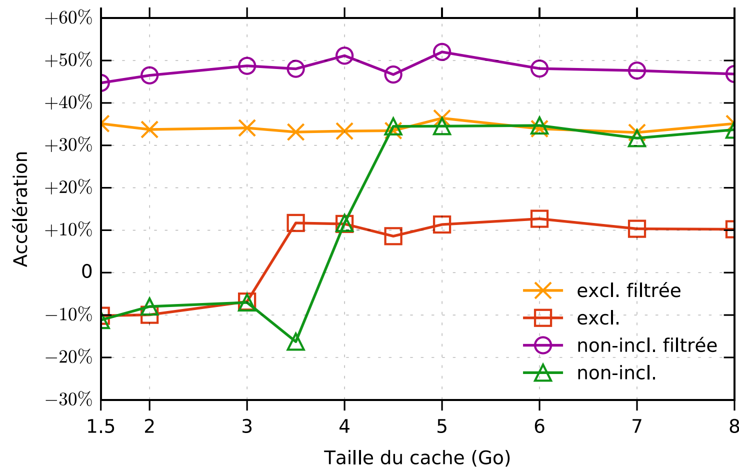


Figure 5.4 – Accélération obtenue avec BLAST

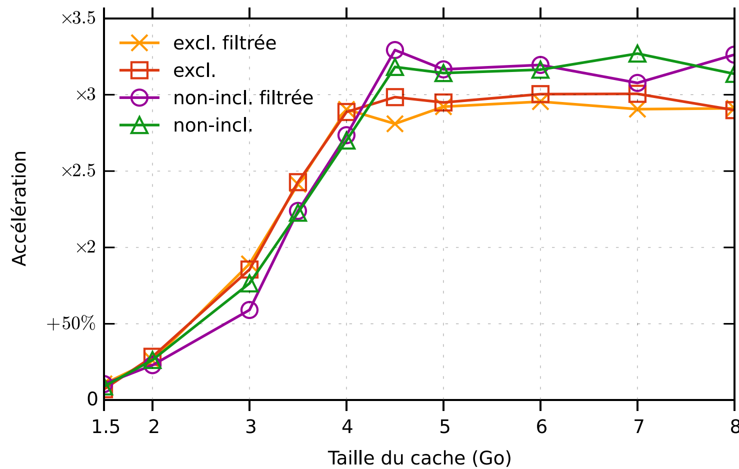
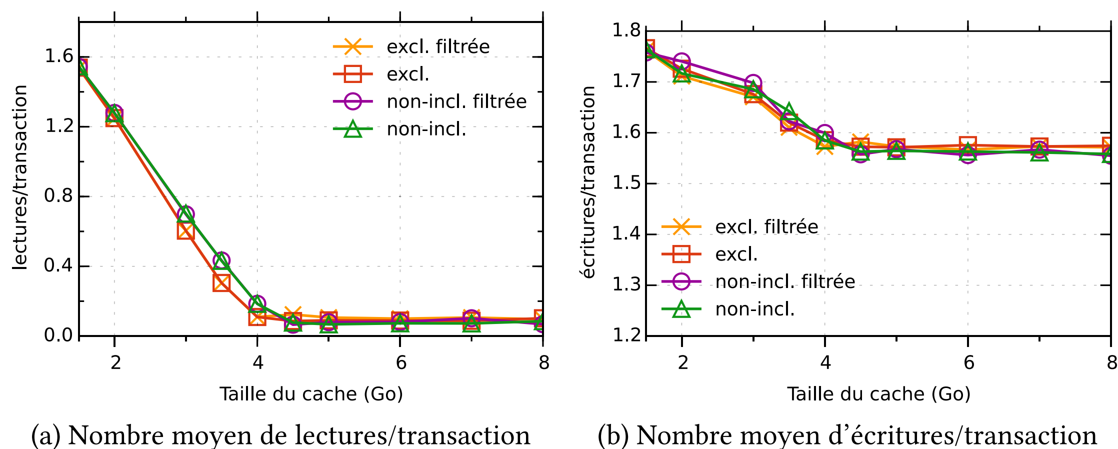


Figure 5.5 – Accélération obtenue avec Postmark

accès aléatoires, plus lents, ce qui permet d'envoyer plus rapidement des messages `get` puisque les files d'attente de Puma ne sont pas surchargées.

5.4.2 Postmark : expérimentations en présence d'écritures

Postmark est conçu pour simuler la charge générée par des applications comme des serveurs d'e-mails qui effectuent beaucoup d'écritures, ce qui pourrait être un des pires scénarios pour Puma puisqu'il ne gère pas les écritures. Cependant, comme le montre la figure 5.5, Puma est capable d'améliorer les performances de Postmark même s'il n'est pas conçu pour : une petite quantité de cache est suffisante pour améliorer les performances de 10%, et il peut multiplier les performances par 3 lorsque l'ensemble des données tient dans le cache.



(a) Nombre moyen de lectures/transaction (b) Nombre moyen d'écritures/transaction

FIGURE 5.6 – Accès au périphérique de stockage avec Postmark

Pour comprendre pourquoi PUMA est capable d'améliorer les performances d'une charge intensive en écritures telle que générée par Postmark, nous avons mesuré le nombre d'E/S envoyées au périphérique bloc pour chaque transaction exécutée. Rappelons que, comme nous l'avons expliqué dans la section 5.1.3.3, le nombre de lectures correspond au nombre de requêtes de lecture soumises au périphérique bloc. De même, le nombre d'écritures correspond au nombre de requêtes d'écriture soumises au périphérique bloc. Les résultats de cette expérience sont présentés dans la figure 5.6. Comme nous nous y attendions, PUMA réduit le nombre de lectures effectuées par transaction (figure 5.6a) pour atteindre une valeur proche de zéro. Il est important de noter qu'il n'est pas possible d'atteindre zéro lecture avec ce benchmark, parce que celui-ci génère de nouvelles données, et en modifie d'autres, ce qui implique parfois des données invalidées dans le cache, qui doivent donc être relues depuis le disque.

Lorsque l'on regarde la figure 5.6b, on remarque que PUMA est également capable de réduire le nombre d'écritures, ce qui est contradictoire avec la conception même de PUMA. Ceci est lié au fait que dans ce benchmark les écritures sont *bufferisées* par le page cache, elles sont donc écrites sur le périphérique bloc lorsque le PFRA choisi de les évincer pour récupérer de la mémoire. Cependant, pour évincer une page sale de la mémoire, le PFRA doit d'abord attendre qu'elle soit écrite sur le disque, il a donc tendance à réclamer en priorité les pages propres. Ainsi, en augmentant la taille du cache à l'aide de PUMA, il y a plus de place dans le cache *local*, ce qui permet de différer plus d'écritures puisque les pages propres peuvent rapidement être évincées et placées sur un autre nœud PUMA.

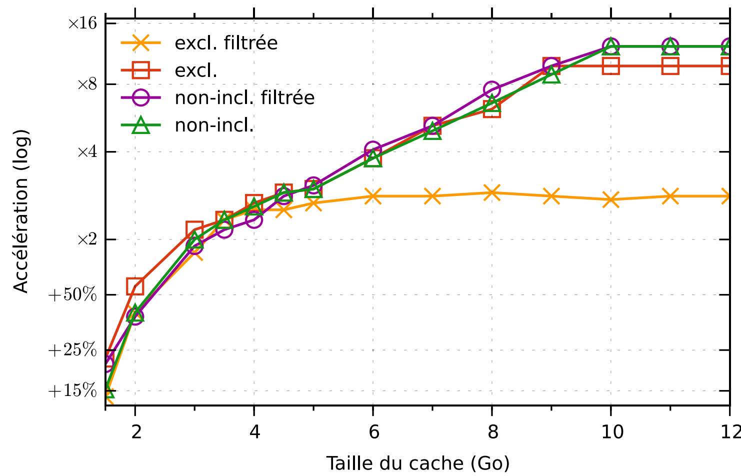


FIGURE 5.7 – Accélération obtenue avec TPC-C

5.4.3 Bases de données

Les benchmarks TPC-C et TPC-H génèrent une forte pression sur le périphérique de stockage avec beaucoup d'E/S concurrentes, et sont très dépendants de la taille du cache [42]. Comme le montrent les figures 5.7 et 5.8, PUMA est capable d'améliorer les performances de ces deux benchmarks même avec une faible quantité de cache, avec une amélioration d'au moins +12% avec TPC-C et +5% avec TPC-H. La concurrence des E/S produites par ces benchmarks peut être observée à l'aide de la stratégie de cache exclusive lorsque l'option *filtre* est activée. Elle est moins efficace que la version non filtrée et atteint très rapidement ses limites : alors que les autres stratégies continuent de bénéficier des augmentations de la taille du cache jusqu'à multiplier les performances par 12 avec TPC-C et par 4 avec TPC-H, la stratégie exclusive avec l'option *filtre* ne dépasse pas un facteur 2,5 avec TPC-C et un facteur 1,5 avec TPC-H.

Ceci est principalement causé par deux facteurs :

- La plupart des E/S sont un mélange d'accès aléatoires et de séquences de taille moyenne. Avec une stratégie de cache non-inclusive (et l'option *filtre* activée), lorsqu'une page est accédée de façon aléatoire, elle reste dans le cache même lorsqu'elle est accédée une seconde fois de façon séquentielle. Dans le cas de la stratégie de cache exclusive avec filtre, les pages récupérées depuis le cache sont retirées de celui-ci lors de leur accès.
- Les accès séquentiels ne sont pas aussi rapides qu'avec le benchmark de lectures séquentielles (section 5.3) ou BLAST (section 5.4.1) parce que plusieurs flux concurrents génèrent des E/S séquentielles. Ces différents flux s'entrelacent, ce qui a tendance à augmenter le temps d'accès au disque (*seek time*). Ainsi, les performances de ces E/S peuvent être plus facilement améliorées en utilisant PUMA.

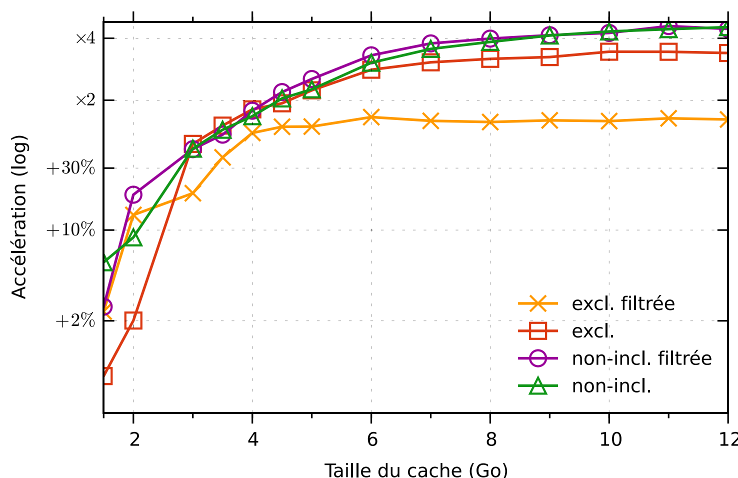


FIGURE 5.8 – Accélération obtenue avec TPC-H

5.5 Analyse de la sensibilité au temps de réponse

La gestion du temps de réponse est un problème important pour PUMA, c'est pourquoi nous avons ajouté un mécanisme de monitoring du temps de réponse entre les nœuds pour éviter de dégrader les performances. Ce mécanisme a été décrit dans la section 4.2.3. Dans cette section, nous montrons que grâce à ce mécanisme, PUMA est capable de limiter son activité pour éviter d'avoir un impact négatif sur les performances des applications

Le problème du temps de réponse peut avoir essentiellement deux causes distinctes :

— une augmentation du temps de traitement par les nœuds, par exemple en cas d'activités concurrentes ;
— une variation de la latence réseau entre les nœuds.

Pour montrer l'impact d'un fort temps de réponse, nous avons utilisé Netem [41] pour injecter de la latence réseau entre les nœuds PUMA, et nous avons mesuré l'accélération obtenue avec nos différentes applications en faisant varier la latence injectée. La figure 5.9 présente les résultats de ces expériences.

Comme nous pouvions nous y attendre, l'accélération des applications obtenue avec PUMA décroît à mesure que la latence réseau augmente. Pire encore, lorsque la latence réseau devient trop élevée PUMA sans son mécanisme de gestion du temps de réponse dégrade de façon significative les performances. Les applications qui sont les plus intensives en E/S sont rapidement impactées par la latence réseau : les performances de Postmark se retrouvent dégradées dès 500μs de latence injectée, et 1.5ms de latence est suffisant pour augmenter considérablement le temps de réponse des transactions de TPC-C.

La figure 5.10 montre les résultats de ces expériences en activant le mécanisme

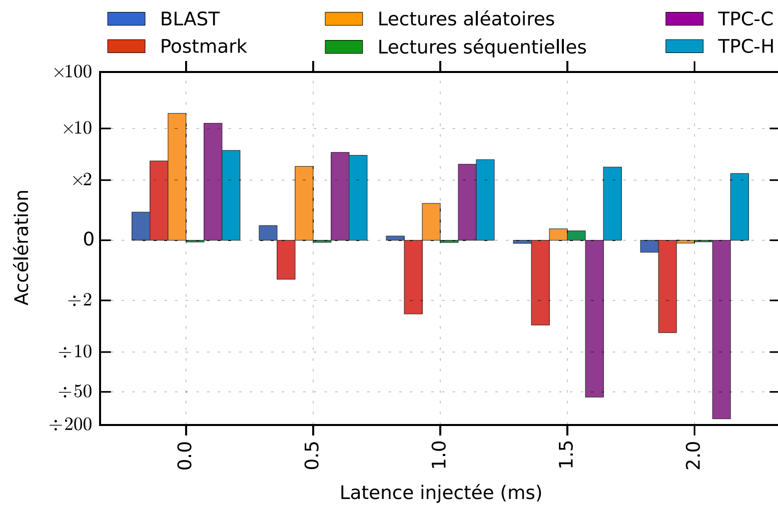


FIGURE 5.9 – PUMA sans contrôle du temps de réponse

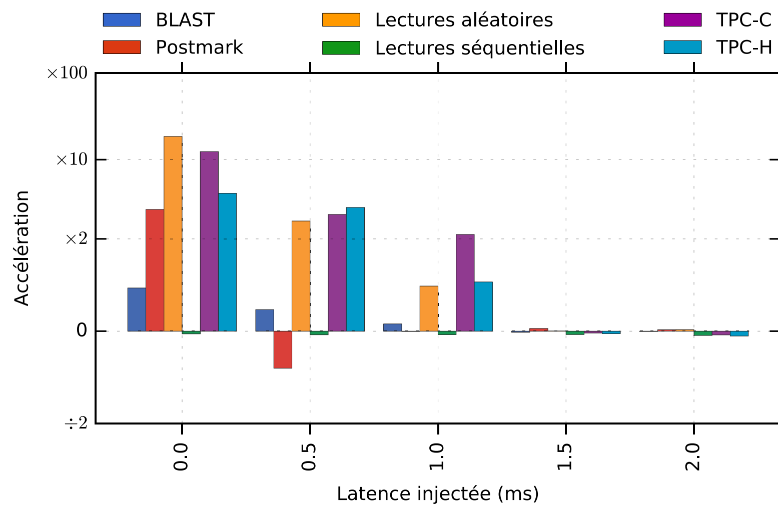


FIGURE 5.10 – PUMA avec contrôle du temps de réponse

de contrôle du temps de réponse de Puma présenté dans la section 4.2.3. Nous l'avons configuré de façon empirique en choisissant $S_{short} = [1.5ms, 40ms]$ et $S_{long} = [1ms, 1.5ms]$. Globalement, on peut observer que ce mécanisme permet à Puma de ne pas ralentir les applications en cas d'augmentation du temps de réponse. La seule exception notable concerne Postmark lorsque nous avons injecté $500\mu s$ de latence réseau, où dans ce cas S_{short} et S_{long} doivent être choisis pour être plus agressifs.

5.6 Comparaison avec un cache sur un SSD

Les disques SSD (*Solid State Drive*) sont des supports de stockage qui utilisent de la mémoire flash, contrairement aux disques dur classiques (HDD) qui utilisent des supports magnétiques. Ils permettent un plus faible temps d'accès et une meilleure bande passante. Cependant, le cout des SSD reste élevé, ce qui ne permet pas de les utiliser pour remplacer les HDD [77], ils sont donc souvent utilisés comme périphériques de cache [8, 54, 79]. Leur fragilité rend cependant cette approche couteuse comparée à Puma, qui consiste à réutiliser des ressources existantes. Cependant, il nous semble intéressant de comparer Puma à de tels caches, c'est pourquoi cette section compare les performances de Puma avec celles d'un cache SSD.

Parmi les solutions de cache sur disque existantes, nous avons choisi d'utiliser *dm-cache* [8], d'une part parce qu'il était déjà intégré au noyau Linux au moment où nous avons effectué cette expérimentation, et d'autre part parce qu'il a été montré qu'il était plus performant que les autres approches [92]. Le module *dm-cache* du noyau Linux permet de créer un périphérique bloc *virtuel* par-dessus un périphérique bloc *réel* (HDD) et un périphérique de cache (SSD). Lorsqu'une opération est soumise au périphérique bloc, dm-cache tente d'abord de l'exécuter sur le périphérique de cache.

Les nœuds de la plateforme Grid'5000 utilisés lors des expériences précédentes ne disposaient malheureusement pas de SSD, nous avons donc utilisé une machine de notre laboratoire. Cette machine est équipée d'un processeur Intel Core i7-2600 (3.4 GHz), de 8 Go de RAM, d'un disque dur classique de 1 To avec une vitesse de rotation de 7200 rpm et d'un SSD Samsung 840 Pro 128GB qui est utilisé exclusivement pour dm-cache. Les résultats présentés dans cette section peuvent donc être légèrement différents de ceux présentés dans les sections précédentes.

Nous avons configuré dm-cache au sein d'une VM possédant 1 Go de mémoire pour qu'il utilise un périphérique de cache de 5 Go, c'est-à-dire que l'ensemble des données de nos benchmarks tiennent dans le cache. Nous avons également lancé ces benchmarks avec Puma et 2 VMs, la première possédant 1 Go de mémoire et la seconde capable d'offrir 5 Go de cache distant.

Nous avons résumé les résultats de ces expériences dans le tableau 5.2. Ces résultats montrent qu'en présence d'accès aléatoires Puma est bien plus efficace qu'un cache

	Lectures aléatoires	Postmark	BLAST	TPC-H
PUMA	**×38**	+55%	**+267%**	**+218%**
dm-cache	×19	**×10**	+98%	+128%

TABLE 5.2 – Comparaison des performances entre un cache SSD (*dm-cache*) et PUMA

SSD, alors que nous nous attendions à obtenir des performances similaires. La principale raison vient du surcout de la virtualisation lié aux changements de contexte entre les VMs et l'hyperviseur [36, 40, 86]. Ce surcout impacte particulièrement les applications qui effectuent beaucoup d'E/S à cause des interruptions `VM Exit` déclenchées lorsqu'une E/S se termine. Ainsi, alors que nous avons mesuré 150μs de latence réseau entre les VMs de PUMA, ce qui est relativement élevé (les VMs étant colocalisées), nous avons mesuré un temps d'accès au SSD depuis la VM de plus de 200μs.

Les résultats des différents benchmarks permettent de montrer les avantages et les inconvénients de deux approches. Avec BLAST, le SSD n'est que très peu impacté par le temps d'accès puisque ceux-ci sont principalement séquentiels. Ainsi, le cache SSD fournit des performances du même ordre de grandeur que celles fournies par PUMA. Cependant, avec cette machine la bande passante réseau disponible pour PUMA est plus élevée que celle disponible avec notre SSD, ce qui permet à PUMA d'obtenir de meilleures performances que dm-cache.

Avec TPC-H, PUMA reste meilleur qu'un cache SSD, mais les écritures générées par ce benchmark (environ 10%) finissent par devoir être écrites sur le disque. Ainsi, puisque nous ne gérons pas les écritures et que les écritures sur un SSD sont bien plus rapides que sur un disque dur classique, la différence entre le cache SSD et PUMA est réduite.

Ce phénomène est amplifié avec Postmark, puisque la majorité de la charge générée par ce benchmark contient des écritures. Ici, le SSD est bien plus rapide que PUMA puisque nous ne gérons pas les écritures, qui finissent par être écrites sur le disque dur classique. Le cache SSD est 10 fois plus rapide que la configuration de référence, ce qui nous montre que le coût des écritures sur un disque magnétique est important : même si la configuration de référence a plus de mémoire pour différer les écritures, elles finissent par devoir être écrites sur le disque magnétique, alors qu'avec le cache SSD elles peuvent être écrites directement sur le SSD.

5.7 Étude de cas : le cloud privé

Pour illustrer l'intérêt de PUMA, nous considérons un scénario dans lequel une entreprise possède ou loue son propre cloud privé, composé de plusieurs nœuds interconnectés par un réseau performant. Une partie de l'activité de cette entreprise repose sur

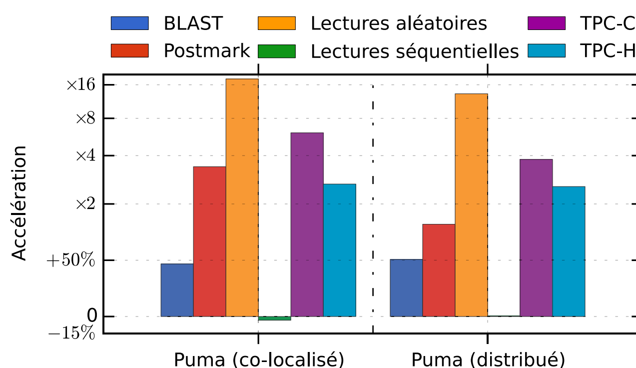


FIGURE 5.11 – Accélération obtenue avec les différents benchmarks et des VMs hébergées sur des nœuds différents

une base de données, elle déploie donc dans son cloud une VM avec 10 Go de mémoire pour y placer sa base de données. Elle possède également une autre VM de 10 Go sur un second nœud qui peut être utilisée en secours en cas de panne du premier nœud. Dans cette situation, la première VM pourrait utiliser la mémoire inutilisée de la seconde VM, pour étendre ses capacités de cache et donc ses performances.

Pour simuler cette étude de cas, nous avons utilisé 2 nœuds du cluster *Paranoia* de la plateforme Grid'5000, dont les caractéristiques sont présentées dans la section 5.1. Ces nœuds sont interconnectés par un réseau Ethernet 10 Gbit/s. Sur le premier nœud, nous avons déployé une VM qui possède 10 Go de mémoire, et nous y exécutons TPC-C pour représenter l'activité de l'entreprise. Nous avons configuré TPC-C pour qu'il utilise 150 entrepôts, ce qui génère environ 15 Go de données. Les 10 Go de mémoire de la VM ne sont donc pas suffisants pour que les données tiennent dans le cache, mais assez pour que celle-ci fournisse des performances acceptables (le temps de réponse des transactions est inférieur à 2 secondes).

Sur le second nœud, nous déployons une VM de secours qui possède 10 Go de mémoire, et peut être utilisée en cas de panne de la première. Ainsi, tant qu'il n'y a pas de panne, la VM de secours peut offrir sa mémoire inutilisée à la VM principale en utilisant PUMA et le réseau Ethernet à 10 Gbit/s.

Les résultats de cette expérience sont présentés dans la figure 5.11. Nous présentons également les résultats de cette expérience avec TPC-H en utilisant 2 VMs contenant 10 Go de mémoire, et nous avons reproduit les résultats des autres benchmarks présentés dans la section 5.1 dans leur configuration avec 8 Go de mémoire (1 + 7). Étonnamment, les performances de PUMA dans cette configuration sont proches des performances avec des VMs colocalisées : en effet, comme nous l'avons déjà expliqué, la virtualisation des E/S est un réel problème, et l'utilisation d'un réseau performant ne fait qu'ajouter quelques dizaines de microsecondes de latence. Malgré tout, PUMA reste capable d'améliorer les performances des applications de façon importante, et la

latence entre deux VMs situées sur des machines physiques différentes n'est supérieure que de quelques dizaines de microsecondes si l'on utilise un réseau performant.

5.8 Conclusion

Ce chapitre a présenté une évaluation poussée de PUMA à l'aide de divers micro-benchmarks et applications. Nous avons montré dans la section 5.2 qu'une stratégie non-inclusive a l'avantage d'être moins coûteuse en échanges de messages, bien que moins efficace qu'une stratégie exclusive en termes de taux de *hit*. Lors d'accès séquentiels, nous avons montré dans la section 5.3 que l'activation du filtre présenté dans la section 4.2.4 permet d'éviter de dégrader les performances des applications. De plus, l'analyse des performances de PUMA de la section 5.4 montre qu'en présence d'accès mixtes (aléatoires/séquentiels), l'activation d'option *filtre* permet d'éviter le *thrashing* sur le cache réparti. Enfin, le temps de réponse entre les nœuds PUMA est critique puisque PUMA repose sur une faible latence pour améliorer les performances. Notre évaluation du mécanisme de contrôle du temps de réponse de PUMA (section 5.5) a montré que l'on pouvait désactiver PUMA lorsque le temps de réponse devient trop élevé pour éviter de dégrader les performances.

Chapitre 6

Gestion dynamique du cache entre machines virtuelles

Sommaire

6.1 Méthodologie . . . 78

6.2 Récupération efficace du cache pour de la mémoire anonyme . . . 79

6.3 Récupération efficace de la mémoire pour du cache . . . 79

6.3.1 Accélérer la récupération : limitation de Puma à la liste inactive . . . 79

6.3.2 Prêter toute sa mémoire : rééquilibrage des listes LRUs . . . 81

6.4 Gestion de la concurrence des accès au cache . . . 82

6.4.1 Détection d'une activité via le *shadow page cache* . . . 84

6.4.2 Détection d'une activité via l'augmentation de la pression mémoire . 85

6.4.3 Combinaison des deux approches . . . 86

6.5 Discussion . . . 86

Nous avons vu dans le chapitre 4 que Puma est directement intégré au page cache du noyau Linux et manipule uniquement des pages « propres », ce qui lui permet d'être efficace et transparent vis-à-vis des applications. Cependant, un des aspects que nous n'avons pas encore abordé dans ce manuscrit concerne la configuration de Puma. Dans la solution présentée précédemment, celle-ci est essentiellement *statique* ; la quantité de mémoire prêtée par un nœud est fixe et prédéfinie. Cette hypothèse est peu réaliste en pratique, en particulier si le nœud est amené à exécuter lui aussi des applications intensives en E/S.

Dans ce chapitre, nous étudions différents mécanismes permettant à Puma d'ajuster automatiquement la quantité de mémoire offerte au cache réparti, sans pénaliser l'activité *locale*. Nous nous intéressons dans un premier temps à la récupération de la

mémoire prêtée par un nœud Puma, puis nous étudions les possibilités de détection de variations d'activité pour mettre d'activer ou de désactiver Puma automatiquement.

Si de nombreuses approches ont été faites dans ce sens [43, 65, 116], elles sont ici difficilement adaptables. En effet, Puma est interfacé avec des mécanismes bas niveau du noyau Linux, il se doit donc de limiter tant son empreinte mémoire et que son cout CPU. L'originalité des mécanismes proposés dans ce chapitre réside dans leur utilisation des fonctionnalités existantes du noyau Linux pour ajuster automatiquement la quantité de mémoire prêtée. On limite ainsi leur impact tant sur les performances de Puma que sur celles du noyau.

6.1 Méthodologie

L'objectif étant à la fois d'automatiser le partage et de minimiser l'impact sur le noyau, la solution que nous proposons doit à la fois être efficace et peu intrusive. Le noyau d'un système d'exploitation étant par définition un logiciel complexe, il est très difficile d'anticiper les effets de bord des mécanismes proposés. Pour bien comprendre la solution que nous apportons il est nécessaire d'associer à sa description un ensemble de micro-expérimentations. Ce chapitre présentera donc non-seulement les mécanismes mais aussi les résultats des micro-expériences. Le chapitre 7 présentera lui une étude exhaustive de la solution retenue.

Pour évaluer Puma en présence d'activité sur plusieurs nœuds, nous avons utilisé 2 machines virtuelles (VMs), la première (VM_1) possède 512 Mo de mémoire, et la seconde (VM_2) possède 700 Mo de mémoire. Nous injectons une charge à l'aide du benchmark de lectures aléatoires de Filebench [70] présenté dans la section 5.1 avec des blocs de 64 ko, et nous monitorons le nombre d'entrées/sorties (E/S) complétées par seconde. Nous mesurons également les variations de la consommation mémoire (page cache, listes LRU, pages hébergées, etc.). Afin de représenter un scénario dans lequel les VMs se « gênent » mutuellement nous avons utilisé un fichier de 500 Mo, ce qui contraint VM_1 à utiliser la mémoire inutilisée de VM_2 pour obtenir de meilleures performances, alors que VM_2 a besoin de la totalité de sa mémoire de disponible. La plateforme d'expérimentations utilisée est présentée en détails dans le chapitre 7.

Dans le reste de ce chapitre, nous représentons dans les résultats de nos expériences la période d'activité de l'activité de VM_1 par ⟷, et la période de l'activité de VM_2 par ⟷.

6.2 Récupération efficace du cache pour de la mémoire anonyme

Lorsqu'un nœud PUMA prête de la mémoire, il doit être capable de la récupérer rapidement, en particulier lorsqu'un processus tente d'allouer de la mémoire. Dans le cas contraire, le processus serait ralenti et pourrait crasher (*OOM-kill*) à l'image de ce que l'on peut observer avec le ballooning [46, 85, 104].

Pour rendre ce mécanisme à la fois automatique et efficace, les pages *hébergées* par PUMA sont directement intégrées dans les listes *Least Recently Used*s (LRUs) du noyau Linux (section 4.3.2). Ainsi, la récupération de la mémoire occupée par ces pages s'effectue de la même manière que les pages locales, via le *Page Frame Reclaiming Algorithm* (PFRA). Dans le cas où le PFRA choisit d'évincer l'une de ces pages, PUMA l'évince immédiatement et notifie de manière asynchrone le nœud d'origine pour qu'il maintienne ses métadonnées. Notons que, comme nous le décrivions dans la section 4.3.3, PUMA ne manipule que des pages propres, il n'est donc pas nécessaire de les renvoyer au nœud client pour qu'il les écrive sur son disque.

6.3 Récupération efficace de la mémoire pour du cache

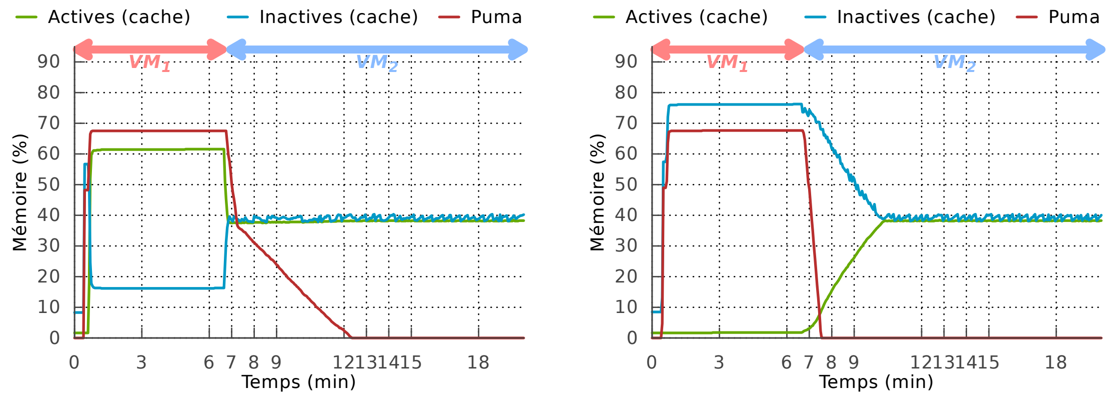
Si, comme nous venons de le voir, le problème de la récupération de la mémoire prêtée dans le cas de la mémoire anonyme (`malloc`) était déjà pris en compte intrinsèquement par le design de PUMA (intégration dans le page cache), il n'en va pas de même lorsqu'une VM doit récupérer les pages pour son propre cache.

Pour commencer nous nous intéressons, dans cette section, à des scénarios *exclusifs*, où les nœuds PUMA n'ont pas d'activité mémoire simultanée. La section suivante (6.4) sera, elle, consacrée au cas plus complexe d'une double activité : les nœuds utilisant leurs caches au même moment.

Nous analysons dans un premier temps le comportement de PUMA lorsqu'un nœud qui *offre* sa mémoire commence une activité intensive en E/S alors que les nœuds à qui il prêtait sa mémoire ne sont plus actifs. Nous étudions ensuite le scénario inverse : un nœud PUMA devient inactif après une activité intensive en E/S et prête sa mémoire à un autre nœud actif.

6.3.1 Accélérer la récupération : limitation de PUMA à la liste inactive

Par définition un cache utilise la mémoire *libre* pour accélérer les futurs accès aux données. PUMA continue donc de stocker les pages des nœuds lorsque ces derniers



(a) Mémoire de la VM_2 en utilisant la totalité du page cache

(b) Mémoire de la VM_2 en utilisant seulement les pages inactives

FIGURE 6.1 – Comparaison du comportement d'un nœud PUMA lorsque les pages distantes sont stockées dans la totalité du page cache ou uniquement dans les pages inactives (VM_1 puis VM_2)

arrêtent de faire des E/S. Cependant, PUMA a aussi pour objectif d'être *transparent*, *i.e.*, de ne pas ralentir les nœuds qui offrent leur mémoire. Ainsi, un nœud PUMA doit être capable de récupérer efficacement sa propre mémoire pour le bénéficie de son cache *local*, géré par le système, sans être pénalisé par les pages d'autres nœuds qu'il héberge. Une première étape vers cet objectif est de pouvoir récupérer cette mémoire lorsqu'elle devient inutile pour les autres nœuds.

Pour évaluer le comportement de PUMA dans cette situation, nous lançons le benchmark de lectures aléatoires sur la VM_1 pendant 400 secondes, puis nous lançons ce même benchmark sur la VM_2 pendant 800 secondes. Pendant toute la durée de l'expérience, nous mesurons les variations de l'utilisation de la mémoire du la VM_2. Les résultats de cette expérience sont présentés dans la figure 6.1a. Initialement, on remarque que la VM_1 rempli la mémoire de la VM_2 avec ses pages, représentées par la courbe *Puma* dans les figures. Dans la suite de ce chapitre, les pages de la VM_1 stockées par la VM_2 seront appelées *pages distantes*. Notons que, comme nous l'avons décrit dans la section 3.3.1, la liste *active* est ici plus grande que la liste *inactive* puisque la pression mémoire sur la VM_2 est nulle pendant les 400 premières secondes.

Lorsque l'activité de la VM_1 s'arrête que celle de la VM_2 démarre, on remarque que la VM_2 récupère lentement les pages de la VM_1 qu'elle héberge : lorsque le benchmark démarre sur la VM_2, les pages actives hébergées sont rapidement récupérées, mais seulement à hauteur de 50%, le reste étant récupéré à une vitesse beaucoup plus faible. En effet, si les premiers 50% sont réclamés en seulement 30 secondes, ce qui correspond à la vitesse des lectures aléatoires sur le disque, le reste est réclamé en 5min. Le problème ici est que la désactivation d'une page active distante la place en tête de la liste inactive, et donc avec une priorité plus élevée qu'une page locale déjà présente

dans cette liste.

Afin de donner une priorité plus élevée aux pages locales, nous avons choisi de ne stocker les pages distantes de PUMA que dans la liste inactive, ce qui par construction leur donne une plus faible priorité que les pages locales puisque celles-ci peuvent être activées. La figure 6.1b montre l'évolution de la mémoire de la VM_2 après cette modification. Comme prévu, les pages distantes sont récupérées beaucoup plus rapidement puisque celles-ci ne sont stockées que dans la liste inactive, ce qui permet au benchmark de la VM_2 d'atteindre sa performance maximale plus rapidement.

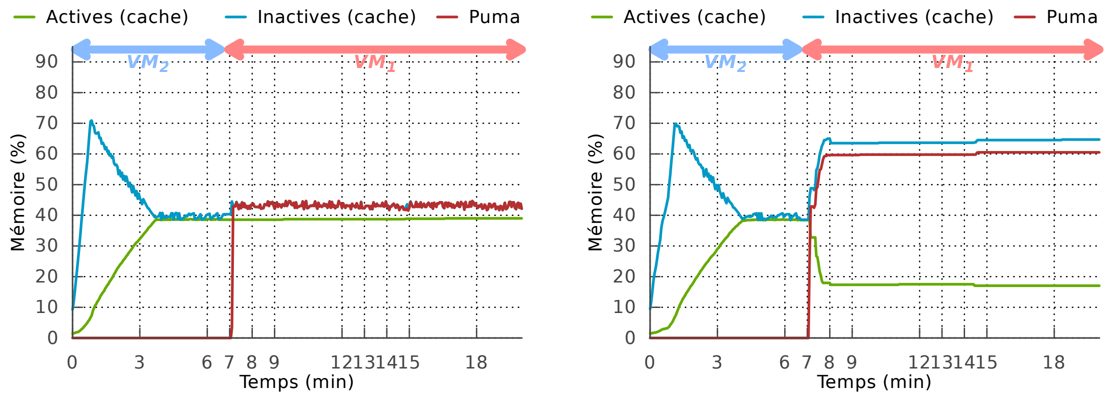
6.3.2 Prêter toute sa mémoire : rééquilibrage des listes LRUs

Comme nous allons le voir dans cette section, la solution présentée précédemment, maintenir les pages de PUMA dans la liste inactive, induit un biais sur le taux de partage maximum.

En effet, lorsqu'un nœud PUMA termine son activité intensive en E/S, il doit être capable de fournir de nouveau sa mémoire inutilisée à d'autres nœuds PUMA. Le problème est ici l'inverse de celui étudié dans la section précédente : les pages locales anciennement actives occupent de la mémoire qu'il doit être possible de prêter à d'autres nœuds. Cependant, nous venons de limiter le placement des pages distantes à la liste inactive, il devient donc impossible de prêter plus de 50% du total du cache d'un nœud PUMA. Pour montrer ce problème, nous avons inversé l'ordre d'exécution des benchmarks de la section précédente. Ainsi, nous avons lancé le benchmark de lectures aléatoires sur la VM_2 pendant 400 secondes, puis pendant 800 secondes sur la VM_1. La figure 6.2a montre l'évolution de la mémoire de la VM_2. On remarque que la VM_1 n'est pas capable d'emprunter plus de 50% du cache disponible de la VM_2.

Pour permettre à une VM d'emprunter plus de mémoire à une autre VM sans la pénaliser, nous avons modifié le noyau Linux pour que des pages actives puissent être désactivées plus fréquemment. En effet, comme expliqué dans la section 3.3.1 le PFRA équilibre la taille des listes actives et inactives lorsqu'il récupère de la mémoire uniquement lorsque $taille(LRU_{active}) > taille(LRU_{inactive})$, c'est pourquoi le page cache tend à contenir environ 50% de pages de chaque type.

Ainsi, pour que PUMA puisse utiliser plus de 50% de la mémoire nous désactivons des pages actives lorsque plus de 90% des pages de la liste inactive contient des pages distantes. Pour éviter que des pages chaudes ne soient activées/désactivées trop souvent, nous avons limité ce mécanisme pour qu'il reste un minimum de pages actives. Nous avons expérimentalement fixé cette limite à 10% de pages actives. La figure 6.2b montre l'évolution de la mémoire de la VM_2 une fois ce mécanisme activé. On remarque que PUMA est maintenant capable d'utiliser tout le page cache, dans les limites que nous avons fixées.



(a) Mémoire de la VM_2 en utilisant seulement les pages inactives

(b) Mémoire de la VM_2 en désactivant les pages actives

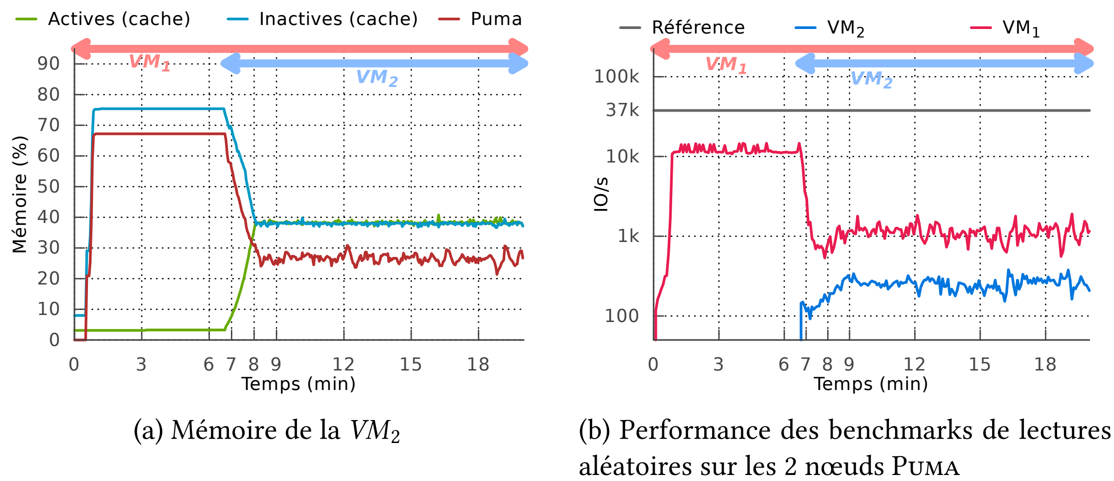
FIGURE 6.2 – Comparaison du comportement d'un nœud PUMA lorsque les pages distantes sont stockées uniquement dans les pages inactives et lorsque PUMA désactive des pages actives lorsque plus de 90% des pages inactives sont des pages distantes (VM_2 puis VM_1)

6.4 Gestion de la concurrence des accès au cache

Nous avons montré dans la section précédente que PUMA est capable de détecter et de récupérer de la mémoire inutilisée au profit d'un autre nœud ou pour lui-même. Cependant, il est possible qu'une activité s'exerce sur un nœud qui prête de la mémoire, dans ce cas il est nécessaire de ne pas dégrader les performances des applications qui s'y exécutent. Ainsi, on parlera d'activité concurrente lorsqu'une VM *active* (VM_1) se sert de la mémoire inutilisée d'une autre VM *inactive* (VM_2) qui finit par redevenir active. La difficulté est ici de déterminer à quel moment l'activité commence et s'arrête, de façon à être capable de prêter de la mémoire à d'autres nœuds en dehors de ces périodes d'activité.

Afin d'étudier le comportement de PUMA dans cette situation, nous reprenons le benchmark Filebench en continu sur la VM_1, puis nous exécutons en parallèle le même benchmark sur la VM_2. Les résultats de cette expérience sont présentés dans la figure 6.3. La figure 6.3a montre l'évolution de la mémoire de la VM_2. On remarque que, grâce aux mécanismes introduits dans la section 6.3, PUMA récupère rapidement une partie des pages qu'il héberge pour son propre usage. Cependant, l'activité concurrente diminue la quantité de mémoire utilisable par la VM_2 puisque VM_1 continue de lui envoyer ses pages évincées.

Les conséquences sur les performances sont énormes, comme le montre la figure 6.3b. Cette figure présente l'évolution au cours de l'expérience des ES/s pour les deux VMs. Elle montre une baisse des performances de la VM_2 s'expliquant simplement par la di-



(a) Mémoire de la VM_2

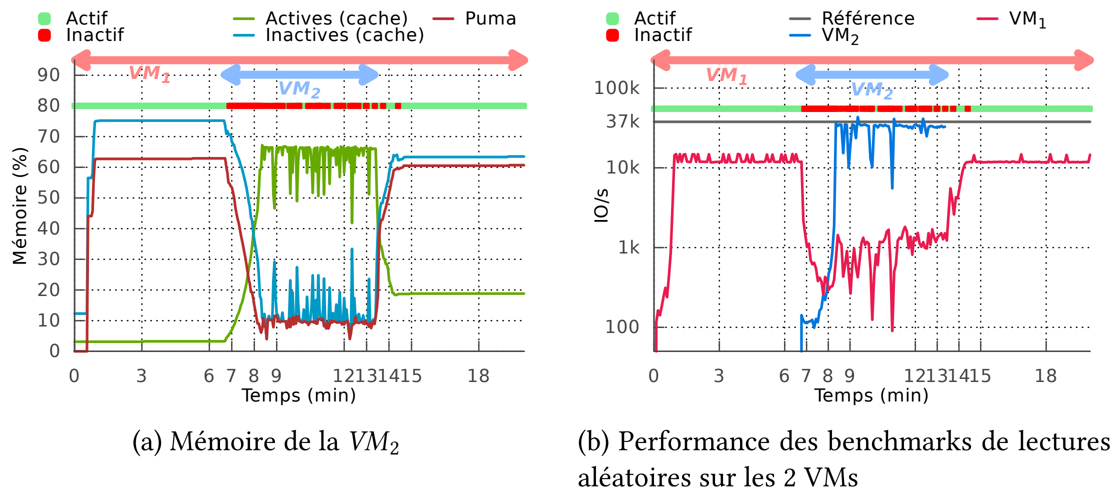
(b) Performance des benchmarks de lectures aléatoires sur les 2 nœuds PUMA

FIGURE 6.3 – Comportement de PUMA en présence de workloads concurrents plusieurs nœuds

minution du nombre de pages hébergées par PUMA. Mais cette figure montre surtout que les performances de la VM_2 restent très faibles (800 ES/s) en comparaison des performances qu'elle aurait obtenu sans l'activation de PUMA (37 000ES/s, noté *référence* sur la figure). PUMA avait pourtant rendu près de 2/3 de la mémoire (figure 6.3a), mais cela reste insuffisant par rapport aux besoins de la VM_1. L'effet de PUMA est donc ici désastreux puisqu'aucune des deux VMs n'atteint des performances acceptables. En conclusion, PUMA ne doit pas rendre une partie de sa mémoire, mais toute la mémoire nécessaire au bon fonctionnement de la VM_2.

Nous proposons donc d'ajouter un mécanisme permettant de détecter l'activité sur la VM_2 pour désactiver temporairement PUMA. Pour ce faire, se reposer simplement sur l'activité CPU n'est pas une solution viable car PUMA serait amené à se désactiver même si de la mémoire reste disponible.

PUMA n'empiétant que sur la mémoire du cache, nous proposons de ne le désactiver qu'une fois une activité intensive en E/S est détectée sur le nœud, ce qui lui permettrait de récupérer sa mémoire sans être dérangé par une autre VM qui continue de lui envoyer des pages. Cette section présente deux heuristiques permettant de détecter une activité « cache » dans le but de désactiver PUMA. Ces heuristiques reposent sur des mécanismes existants du noyau Linux : le *shadow page cache*, dont le fonctionnement a été présenté dans la section 3.3.2, et l'algorithme de récupération de la mémoire (PFRA), qui a été présenté dans la section 3.3.1.



(a) Mémoire de la VM_2

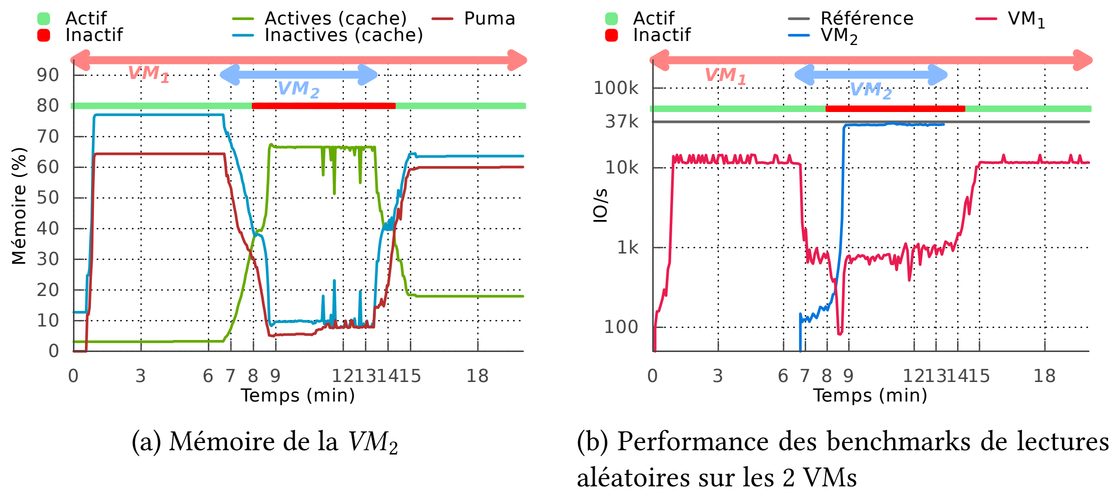
(b) Performance des benchmarks de lectures aléatoires sur les 2 VMs

FIGURE 6.4 – Comportement de PUMA en présence de workloads concurrents sur plusieurs nœuds et en désactivant PUMA lors d'un hit dans les *shadow pages* de la VM_2

6.4.1 Détection d'une activité via le *shadow page cache*

Une première approche pour détecter une activité intensive en E/S est d'utiliser l'algorithme d'estimation du *working set* présenté dans la section 3.3.2, qui repose sur un *shadow page cache*. En effet, lorsque cet algorithme détecte un *shadow hit*, cela veut dire qu'une page *chaude* a été évincée du page cache parce que la liste inactive est trop petite. Ceci peut être causé par PUMA, puisque des pages distantes peuvent avoir « poussé » cette page chaude dehors. Dans le cas contraire, cela veut dire que la VM subit une forte charge en termes d'E/S et nécessite plus de cache. Ainsi, nous décidons de désactiver PUMA à chaque fois que cet algorithme détecte un *shadow hit* pour faire en sorte qu'un nœud n'accepte plus qu'on lui envoie des pages. Le service ne se réactivera qu'en l'absence de *shadow hit* pendant un délais, que nous avons fixé empiriquement à 3 secondes lors de nos expériences.

Nous avons reproduit l'expérience précédente en désactivant PUMA lors d'un *shadow hit* dans le page cache de la VM_2. Ces résultats sont présentés dans la figure 6.4. Dans ces résultats, nous avons indiqué les périodes pendant lesquelles PUMA est actif sur la VM_2 (vert) et inactif (rouge). Sur la figure 6.4a, on remarque que PUMA arrive à détecter l'activité et à se désactiver, ce qui lui permet de récupérer la mémoire dédiée aux pages distantes. Enfin, sur la figure 6.4b on observe que les performances du benchmark sur la VM_2 sont proches de notre référence. Cependant, on remarque sur ces deux figures que PUMA se réactive par courtes périodes pendant la durée du benchmark, ce qui provoque de larges variations dans les performances mesurées. En effet, le mécanisme de réveil montre ses limites : la VM_2 utilise la totalité de son cache local sans pour autant générer de *shadow hit*. C'est le cas lorsque les données à forte localité tiennent entièrement dans la mémoire. Pour limiter ces variations une seconde



(a) Mémoire de la VM_2

(b) Performance des benchmarks de lectures aléatoires sur les 2 VMs

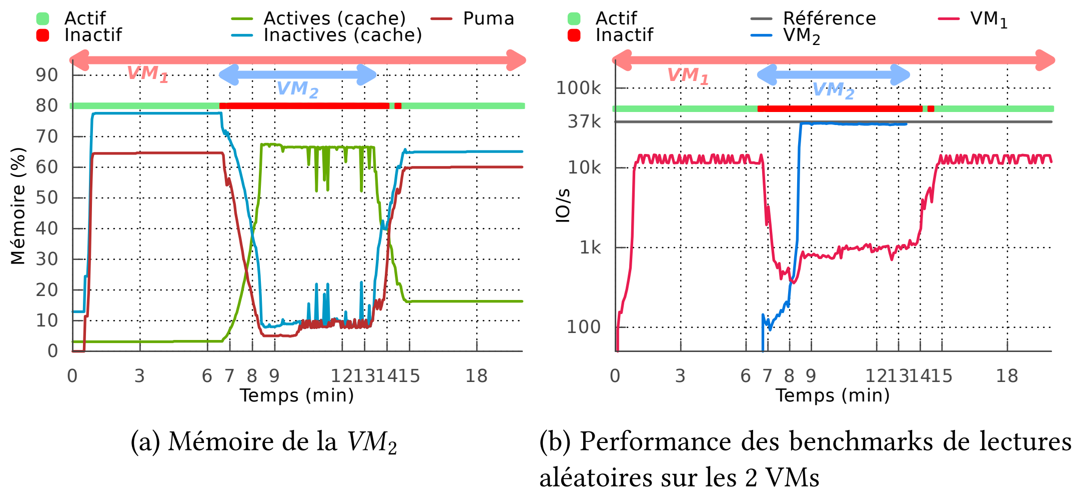
FIGURE 6.5 – Comportement de PUMA en présence de workloads concurrents sur plusieurs nœuds et en désactivant PUMA lorsqu'une page active de la VM_2 est désactivée

métrique pour détecter l'activité est donc nécessaire.

6.4.2 Détection d'une activité via l'augmentation de la pression mémoire

Une seconde méthode pour détecter une activité cache repose sur le rééquilibrage des listes actives/inactives tel que décrit dans la section 3.3.1. En effet, lorsqu'il y a une activité intensive en E/S le PFRA décide de désactiver une page active pour rééquilibrer les listes. Cette situation ne peut se produire qu'en cas de pression mémoire et lorsque la taille de la liste active est supérieure à la taille de la liste inactive. Or, si la liste active est plus grande que la liste inactive, cela veut dire que des pages inactives ont pu être activées et donc que la VM_2 possède une activité suffisamment importante en E/S au point d'activer des pages *et* d'en réclamer.

La figure 6.5 montre les résultats de l'expérience précédente en désactivant PUMA pendant 3 secondes lorsque que les listes LRU sont rééquilibrées par le PFRA. Comparée à l'approche précédente, on remarque que ce détecteur est beaucoup plus précis : PUMA ne se réactive pas avant que le benchmark ne se termine. Cependant, cette approche nécessite un délai important avant d'être efficace, puisqu'il faut attendre que 50% du page cache soit occupé par des pages actives avant que le PFRA ne commence à rééquilibrer les listes. En effet, on peut remarquer qu'il s'écoule environ 1 minute entre le début du benchmark de la VM_2 et l'arrêt de PUMA, alors que l'approche précédente désactivait PUMA immédiatement.



(a) Mémoire de la VM_2

(b) Performance des benchmarks de lectures aléatoires sur les 2 VMs

FIGURE 6.6 – Comportement de PUMA en présence de workloads concurrents sur plusieurs nœuds et en combinant les 2 approches

6.4.3 Combinaison des deux approches

Afin de bénéficier des avantages des deux approches, nous proposons de les combiner pour détecter l'activité d'une VM à la fois rapidement (1re approche, *shadow page cache*) et de façon stable (2e approche, pression mémoire). Nous avons reproduit l'expérience précédente et nous présentons les résultats dans la figure 6.6. Comme nous pouvions nous y attendre, la combinaison des deux approches permet de détecter l'activité de la VM_2 rapidement, ce qui permet de désactiver PUMA pour que celle-ci puisse utiliser son cache sans être pénalisé par la VM_1.

6.5 Discussion

Ce chapitre a présenté plusieurs mécanismes permettant à PUMA d'ajuster dynamiquement la quantité de mémoire qu'un nœud accepte de prêter à d'autres nœuds. Ces mécanismes reposent notamment sur l'intégration des pages distantes au sein du page cache du noyau Linux ce qui permet de les récupérer automatiquement en reposant sur l'algorithme de récupération de la mémoire existant. Nous nous reposons également sur cet algorithme pour détecter une reprise d'activité du nœud, ce qui lui permet de suspendre temporairement le service de cache qu'il offre à d'autres nœuds afin de ne pas dégrader ses propres performances. Couplé à une autre approche basée sur l'activité du *shadow page cache* du noyau Linux, nous arrivons à détecter rapidement une reprise d'activité cache et à suspendre PUMA de façon continue jusqu'à ce que l'activité ne s'arrête.

Il reste cependant difficile de détecter le moment exact où l'activité cache d'une

VM reprend. En effet, une approche « naïve » et précise consisterait à désactiver PUMA lors d'un accès disque ou d'une activation d'une page. Dans ce cas, PUMA risquerai d'être désactivé en permanence, ne serait-ce que parce que certaines tâches du système telles que les journaux font régulièrement de courtes E/S, qui désactiveraient le service alors qu'elles ne nécessitent pas particulièrement de cache. Ainsi, si nos heuristiques permettent de détecter une reprise d'activité, elles nécessitent un certain temps avant que l'activité ne soit réellement détectée :

- pour le *shadow page cache*, un *hit* dans celui-ci ne peut se produire que lorsqu'une page, potentiellement chaude, a déjà été évincée ;
- pour le rééquilibrage des listes LRU, il faut attendre que :
  1. le page cache soit entièrement rempli, sinon le PFRA ne s'active pas ;
  2. la taille de la liste active atteigne 50% de la totalité du page cache.

Chapitre 7

Évaluation de l'automatisation de PUMA

Sommaire

7.1 Plateforme expérimentale **89**
7.1.1 Benchmarks 90
7.2 Récupération automatique du cache pour de la mémoire anonyme . . **91**
7.2.1 Analyse de la récupération de la mémoire 91
7.2.2 Efficacité de la récupération : vitesse des allocations 93
7.2.3 Étude de cas : la consolidation sur un serveur unique 94
7.3 Surcout du mécanisme de partage automatique **97**
7.4 Analyse du seuil de désactivation des pages **100**
7.5 Conclusion **103**

Ce chapitre présente une évaluation globale des mécanismes de gestion dynamique de la mémoire de PUMA présentés dans le chapitre 6. Nous présentons dans un premier temps la plateforme et les benchmarks que nous avons utilisé (section 7.1). La section 7.2 démontre l'efficacité de PUMA à récupérer la mémoire prêtée par un nœud pour allouer de la mémoire anonyme. Nous présentons dans la section 7.3 le surcout des mécanismes de partage dynamique du cache de PUMA. La section 7.4 présente quant à elle une analyse de sensibilité au seuil de désactivation des pages actives, puis la section 7.5 conclut ce chapitre.

7.1 Plateforme expérimentale

Les expérimentations présentées dans ce chapitre ont été effectuées sur une station de travail équipée d'un processeur Intel Core i7-2600 (3.4 GHz), de 8 Go de RAM et d'un

disque dur classique de 1 To avec une vitesse de rotation de 7200 rpm. L'environnement utilisé est similaire à celui présenté dans la section 5.1, que nous rappelons brièvement ici.

Les benchmarks sont déployés dans des machines virtuelles (VMs) en utilisant l'hyperviseur KVM [56] avec la version 1.7.50 de QEMU. Les disques virtuels sont paramétrés pour contourner le page cache du système hôte et nous utilisons l'ordonnanceur d'entrées/sorties (E/S) *deadline* et le système de fichiers *ext4*. Nous utilisons également le framework de paravirtualization *VirtIO* [84] pour accélérer les E/S. Les VMs utilisées sont identiques à celles utilisées dans les chapitres 5 et 6 et sont déployées à l'aide de notre plateforme d'expérimentations dérivée de MOSBENCH [19].

7.1.1 Benchmarks

Les évaluations de ce chapitre reposent essentiellement sur deux microbenchmarks, le premier a pour objectif de générer des E/S, le second permet de générer des allocations de mémoire pour simuler une application gourmande en mémoire.

Lectures aléatoires. Nous utilisons *Filebench* [70], présenté dans la section 5.1.1, pour générer des lectures aléatoires simulant une activité intensive en E/S. Nous avons modifié le benchmark d'origine de façon à reporter en temps réel le nombre d'E/S par seconde, avec une granularité de 5 secondes. En effet, le résultat du benchmark original est une métrique finale, or nous voulons ici étudier l'évolution des performances au cours du temps.

Nous utilisons dans ce chapitre des blocs de 64 ko, la taille du fichier utilisé est variable, elle est précisée lors de la description de chacune des expériences.

Allocations de mémoire. Afin de générer des allocations de mémoire, nous avons développé un microbenchmark qui alloue (`malloc`) progressivement toute la mémoire de la VM sur laquelle il s'exécute. Ces allocations sont effectuées par petits fragments, de taille fixe, que nous validons en écrivant des données dans chacun d'eux pour s'assurer que les pages ont réellement été allouées à la VM par le système d'exploitation. Nous mesurons le temps nécessaire pour créer chaque fragment ($allocation + écriture$), ce qui permet de mesurer le ralentissement dû à PUMA subit par la VM. Une fois la mémoire allouée, le benchmark temporise puis libère progressivement la mémoire.

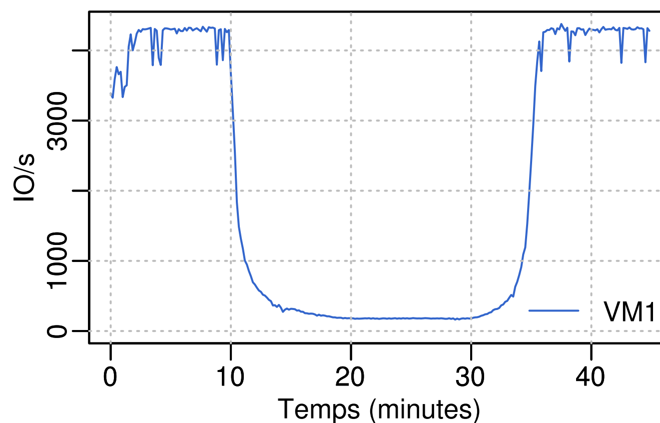


FIGURE 7.1 – Performances de Filebench en nombre d'opérations par seconde dans la VM_1

7.2 Évaluation de la récupération automatique du cache pour de la mémoire anonyme

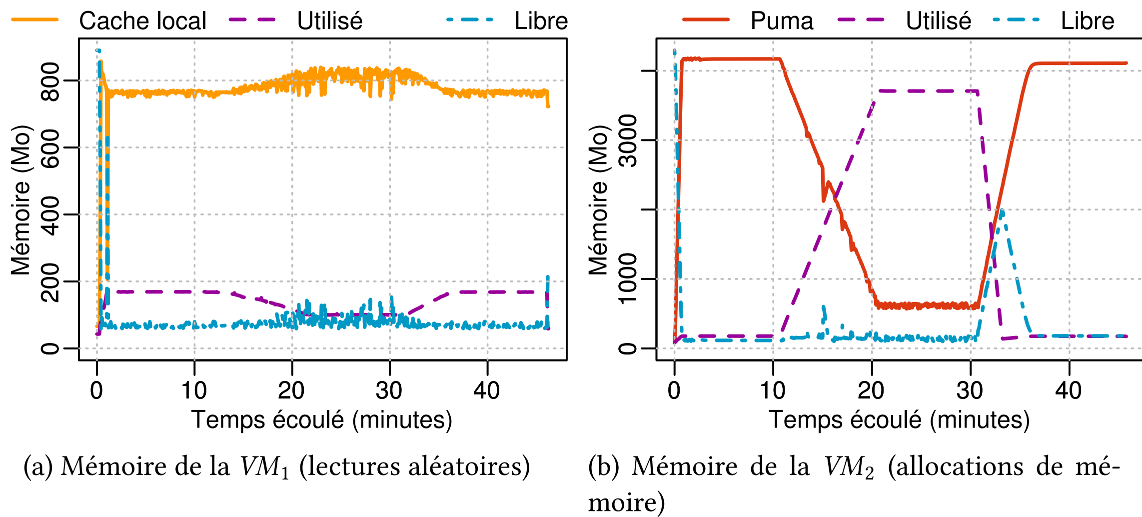
Cette section a pour objectif d'évaluer les performances de la récupération de la mémoire pour allouer des pages anonymes, tel que décrit dans la section 6.2. Nous commençons par analyser le comportement de la récupération de la mémoire (section 7.2.1), puis nous étudions son efficacité dans la section 7.2.2. Enfin, nous démontrons l'intérêt de PUMA comparé à une approche à base de ballooning automatique au travers d'une étude de cas dans la section 7.2.3.

7.2.1 Analyse de la récupération de la mémoire

Cette section a pour objectif d'analyser le comportement de PUMA lorsqu'il doit récupérer sa mémoire pour des pages anonymes. C'est le cas par exemple lorsqu'un processus alloue de la mémoire (`malloc`). Pour cela, nous avons utilisé deux VMs, VM_1 et VM_2. La première (VM_1) possède 1 Go de mémoire et exécute le benchmark de lectures aléatoires présenté dans la section 7.1.1, en utilisant un fichier de 4 Go. Ainsi, celui-ci ne tient pas dans le cache, elle utilisera donc PUMA pour bénéficier de la mémoire inutilisée de VM_2.

La seconde VM (VM_2) possède 4,5 Go de mémoire, qui peuvent être prêtés par PUMA en utilisant la stratégie de cache non-inclusive. Sur cette VM, nous exécutons le benchmark d'allocations de mémoire après 10min. Celui-ci est configuré pour allouer 90% de la mémoire de la VM_2 en 10min, puis d'attendre 10min avant de libérer progressivement la mémoire allouée.

La figure 7.1 montre les performances de Filebench en nombre d'opérations par



(a) Mémoire de la VM_1 (lectures aléatoires)

(b) Mémoire de la VM_2 (allocations de mémoire)

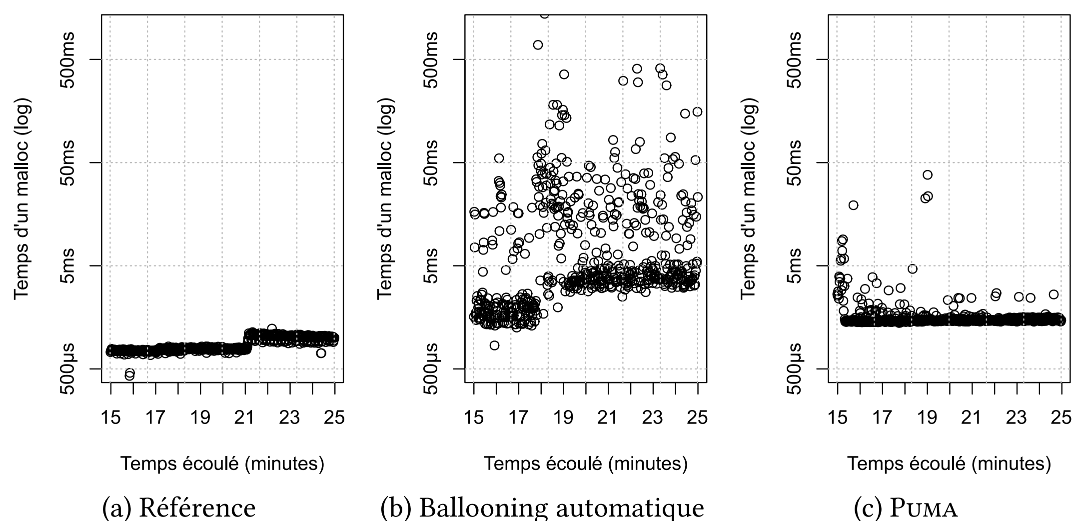
FIGURE 7.2 – Utilisation de la mémoire avec une gestion dynamique PUMA et des besoins variables

seconde dans la VM_1. On y distingue clairement trois phases.

1. Avant que la VM_2 ne commence son activité mémoire, l'ensemble des données tiennent dans le cache (local+distant) et les performances sont au plus haut.
2. Lorsque que l'activité mémoire commence, les performances s'effondrent rapidement. En effet comme nous l'avons vu dans la section 5.2, quelques accès disque sont suffisants pour ralentir considérablement ce type d'application.
3. Enfin, la VM_2 ralentissant son activité mémoire, le cache distant peut se remplir progressivement, ce qui permet d'augmenter le taux de hit de la VM_1 et donc ses performances.

Pour analyser le comportement de PUMA lors des transitions de phases, et donc celui du mécanisme de récupération présenté dans la section 6.2, nous avons mesuré l'évolution de l'utilisation de la mémoire dans la VM_2. Ainsi, la figure 7.2b présente la quantité de mémoire inutilisée (libre) de la VM_2, ainsi que celle occupée par PUMA et par les pages anonymes.

Sur cette figure, on peut voir que le démarrage du benchmark d'allocations de mémoire (à 10min) induit dynamiquement une réduction de la taille de PUMA. Les pages ainsi récupérées sont alors allouées au benchmark. Notons que la baisse de performances observée dans la figure 7.1 ne suit pas la même pente : elle est beaucoup plus rapide. Ceci peut être surprenant sachant que nous n'avons pas choisi une charge séquentielle, particulièrement sensible aux évictions, mais une charge aléatoire. Cependant, les temps d'accès au disque sont extrêmement longs, ce qui dégrade rapidement les performances. Ainsi, les performances des lectures aléatoires (figure 7.1) s'effondrent en quelques dizaines de secondes, alors que les allocations de mémoire du



(a) Référence (b) Ballooning automatique (c) PUMA

FIGURE 7.3 – Latence des allocations de mémoire

benchmark de la VM_2 sont configurées pour s'étaler sur 10 minutes. Pour les mêmes raisons, la hausse des performances n'est obtenue qu'une fois l'essentiel des données mises en cache dans PUMA.

Pour finir, notons que la libération de la mémoire est plus rapide que le rythme des évictions vers PUMA, ce qui explique qu'une partie des pages restent libres pendant plusieurs minutes. Ceci est lié à la vitesse des lectures aléatoires sur le disque par la VM_1, plus lentes que la libération de la mémoire par la VM_2.

L'analyse de la figure 7.2a, pendant de la figure 7.2b sur la VM_1, est moins intuitive. En effet, lorsque la VM_2 réduit dynamiquement la taille du cache (entre 10min et 20min), l'utilisation de la mémoire locale de la VM_1 pour du cache augmente. Ceci peut sembler paradoxal, puisqu'elle aurait dû pouvoir utiliser cette mémoire locale avant que PUMA ne s'active. Ce phénomène s'explique par la réduction du nombre des métadonnées utilisées pour localiser les pages envoyées à la VM_2 qui libère de la mémoire, permettant ainsi d'étendre le cache local.

7.2.2 Efficacité de la récupération : vitesse des allocations

Pour vérifier l'efficacité du mécanisme, nous avons mesuré le temps de chaque allocation de mémoire. Les résultats sont présentés dans la figure 7.3 sous forme de graphiques temporels, où chaque point indique le temps d'une allocation (échelle logarithmique). On y compare 3 configurations :

— ***Référence*** (figure 7.3a) : exécution du benchmark d'allocations de mémoire sur une VM sans aucun mécanisme de partage de la mémoire.
— ***Ballooning automatique*** (figure 7.3b) : exécution de Filebench et du bench-

mark d'allocations de mémoire dans deux VMs, de la même façon que décrite précédemment (voir section 7.2.1), mais en utilisant un ballooning automatique pour KVM [23] (voir section 2.5.3). Dans cette configuration, nous avons fixé la quantité de mémoire de la VM_1 (qui exécute Filebench) à 1 Go, et celle de la VM_2 (qui effectue les allocations de mémoire) à 4,5 Go. Pour reproduire un environnement dans lequel toute la mémoire est utilisée par les VMs, nous avons limité la mémoire de l'hôte à 6 Go. L'hôte dispose également d'un espace de *swap*, qu'il peut utiliser si nécessaire.

— ***PUMA*** : même configuration que précédemment (section 7.2.1).

Avec la configuration de référence (figure 7.3a), les allocations de mémoire prennent moins de 1 ms avec un écart type de 0,1.

Lorsque l'on utilise le ballooning automatique (figure 7.3b), les allocations de mémoire prennent 20 ms en moyenne, avec un important écart type (76), certaines valeurs que nous n'avons pas représentées sur cette figure dépassent 1 s. Cette latence est causée par la VM_1 qui exécute le benchmark de lectures aléatoires : toute la mémoire est utilisée pour son propre cache, et il n'est pas possible de dégonfler le ballon de la VM_2 pour son activité mémoire sans swapper. Ceci est causé par le *gap sémantique* entre l'hyperviseur et les VMs : l'hôte ne sait pas que la mémoire est utilisée pour du cache et peut donc être récupérée. Il est alors impossible de récupérer la mémoire sans ajouter de couteux mécanismes de synchronisation (entre VMs et hyperviseur).

Avec PUMA (figure 7.3c), comme nous embarquons la logique dans les VMs, nous pouvons récupérer la mémoire utilisée pour du cache sans l'aide de l'hyperviseur. En effet, la mémoire prêtée via PUMA ne peut, par choix de conception, servir qu'à stocker des pages de cache propres. Récupérer la mémoire peut donc se faire sans aucune synchronisation, ce qui rend notre approche particulièrement efficace. Dans ce contexte, nous avons mesuré en moyenne 1,8ms de latence pour les allocations de mémoire, avec un écart type de 2,2.

7.2.3 Étude de cas : la consolidation sur un serveur unique

Pour illustrer l'intérêt de PUMA comparé à une approche reposant sur le ballooning, cette section propose d'étudier le scénario d'une entreprise consolidant ses activités au sein d'un unique serveur. Ainsi, nous prenons l'exemple d'une entreprise dans le domaine des nouvelles technologies qui déploie une base de données dans une VM, et un système de gestion de versions ou un serveur d'intégration continu (activité de développement) dans une autre VM.

Pour consolider son activité, cette entreprise achète une machine puissante capable d'accueillir ses deux VMs en même temps. Cependant, compte du tenu du prix d'une telle machine, elle souhaite exploiter au maximum ses capacités. Cela passe par exemple par l'utilisation d'une approche à base de ballooning automatique. En effet, si

	TPC-C (temps de réponse)	1 git clone	3 git clone
Référence	3,354s	215s	204s
Auto-ballooning	**crash**	339s	447s
PUMA	1,186s	184s	251s

TABLE 7.1 – Temps de réponse du ballooning automatique.

l'activité de la base de données peut être continue, l'activité « développement » ne l'est pas : elle est composée de longues périodes d'inactivité et de pics d'activité. C'est le cas par exemple si les développeurs de l'entreprise pensent à transférer leurs travaux en cours lors d'une pause, à l'heure du gouter. Cette opération, ponctuelle, peut nécessiter une large quantité de mémoire (recompilation, tests unitaires, etc.).

Pour représenter ce scénario, nous utilisons un nœud du cluster *Paranoia* de la plateforme Grid'5000 [17] (site de Rennes), dont les caractéristiques sont décrites dans la section 5.1. Nous avons limité la quantité de mémoire de ce nœud à 16 Go pour simplifier notre expérience : utiliser toute la mémoire disponible (128 Go) la rendrait excessivement longue. Nous réservons 2 Go de ce nœud au système d'exploitation hôte, ce qui laisse environ 14 Go de mémoire qui peuvent être répartis entre les VMs.

Dans ces expériences, nous utilisons une première VM dans laquelle nous déployons TPC-C, décrit dans la section 5.1.1, pour simuler l'activité de cette entreprise. Nous le configurons pour qu'il utilise 150 entrepôts, ce qui donne un jeu de données d'environ 15 Go. Cette VM dispose de 10 Go de mémoire, ce qui est suffisant pour offrir des performances acceptables (de l'ordre de la seconde).

Pour simuler l'activité de développement de l'entreprise, nous avons déployé un serveur Git sur une seconde VM qui contient un miroir du dépôt Git du noyau Linux. Avec ce serveur, chaque `git clone` génère un pic d'utilisation de la mémoire. En effet, dans sa configuration de base, lors d'un `git clone` un serveur Git doit allouer de la mémoire : il génère une archive du dépôt, en mémoire, et la compresse pour l'envoyer au client. Dans le cas d'un dépôt important, tel que celui du noyau Linux, cette action peut consommer une quantité de mémoire importante. Nous avons configuré cette VM pour qu'elle dispose de 4 Go de mémoire.

Notre simulation de ce scénario repose sur l'exécution de TPC-C en continu sur la première VM. Pendant son exécution nous effectuons un ou trois `git clone` concurrents du dépôt de la seconde VM depuis une autre machine, cliente. Nous mesurons le temps d'exécution du `git clone` : plus cette opération dure longtemps, plus le temps du gouter du développeur sera court. Nous reportons pour TPC-C le temps de réponse des transactions (9e décile).

Le tableau 7.1 présente les résultats de cette expérience sans mécanisme de partage de la mémoire (**Référence**), avec du ballooning automatique (**Auto-ballooning**) et

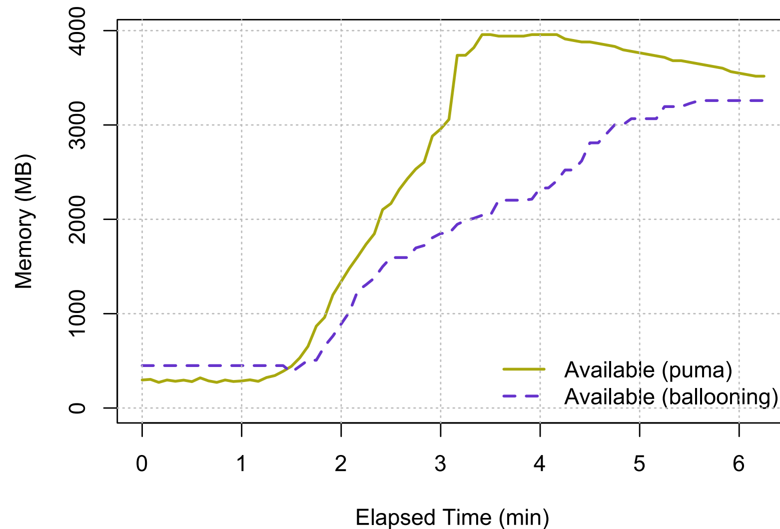


FIGURE 7.4 – Mémoire disponible sur la VM Git avec PUMA et du ballooning automatique

avec PUMA. Sans aucun mécanisme de partage, les performances de TPC-C sont acceptables et les `git clone` s'exécutent en isolation de la base de données.

Lorsque l'on active le ballooning automatique, la VM exécutant TPC-C a initialement récupéré la mémoire *libre* de la VM Git, ce qui a amélioré ses performances. Cependant, à cause du *gap sémantique* entre les VMs et l'hyperviseur, celui-ci n'a pas pu récupérer la mémoire pour la rendre à la VM Git, ce qui l'a forcé à swapper des pages des VM, et donc réduit considérablement les performances des `git clone` (+219%). De plus, si les performances de TPC-C étaient excellentes (0,123s de temps de réponse), son système d'exploitation invité a fini par tuer (*OOM-kill*) le système de gestion de base de données (ici PostgreSQL), par manque de mémoire, lorsque les `git clone` se sont exécutés sur l'autre VM.

À l'inverse, si nous utilisons PUMA les `git clone` restent ralentis (+23%), mais les performances de TPC-C peuvent être améliorées (×3) lorsque l'activité de la VM de développement est faible, sans faire crasher PostgreSQL.

Pour mieux comprendre pourquoi le ballooning automatique ne fonctionne pas, et pour montrer pourquoi l'approche choisie par PUMA est efficace, nous avons mesuré la quantité de mémoire utilisable par la VM Git dans les deux cas (figure 7.4). Pour PUMA, nous représentons la quantité de mémoire libre disponible dans la VM moins la quantité de pages distantes hébergées. Dans le cas du ballooning automatique nous mesurons la quantité de mémoire disponible pour la VM (mémoire de la VM moins la taille du ballon).

Sur ces courbes, on distingue clairement trois phases. Initialement, il n'y a presque pas de mémoire : celle-ci est soit prêtée au cache réparti (via PUMA), soit donné à l'autre

VM (via le ballooning automatique). La seconde phase de ces courbes (à 1'30") correspond au moment où nous effectuons les `git clone`, ce qui nécessite de la mémoire. On remarque clairement que Puma est capable de récupérer sa mémoire (prêtée) plus rapidement que le ballooning automatique (donnée). En effet, dans le cas du ballooning automatique, l'hyperviseur doit swapper des pages des VMs pour pouvoir augmenter la quantité de mémoire de la VM hébergeant le serveur Git, ce qui est extrêmement lent. Ces résultats sont cohérents avec ce que nous avons observé dans les expériences précédentes (figures 7.2b et 7.3). Lors de la troisième phase (à 3'30" pour Puma), les `git clone` se sont terminés et Puma recommence à prêter sa mémoire la VM TPC-C.

7.3 Surcout du mécanisme de partage automatique

L'introduction de mécanismes de partage automatique de la mémoire a pour objectif d'automatiser le fonctionnement de Puma tout en limitant le surcout du service offert. Nous nous concentrons dans cette section sur l'activité concurrente de deux VMs, où une première VM (VM_1) bénéficie d'une seconde VM (VM_2), inactive. Comme nous l'avons détaillé dans la section 6.4, l'objectif est ici de détecter que VM_2 redevient active pour éviter de dégrader ses performances.

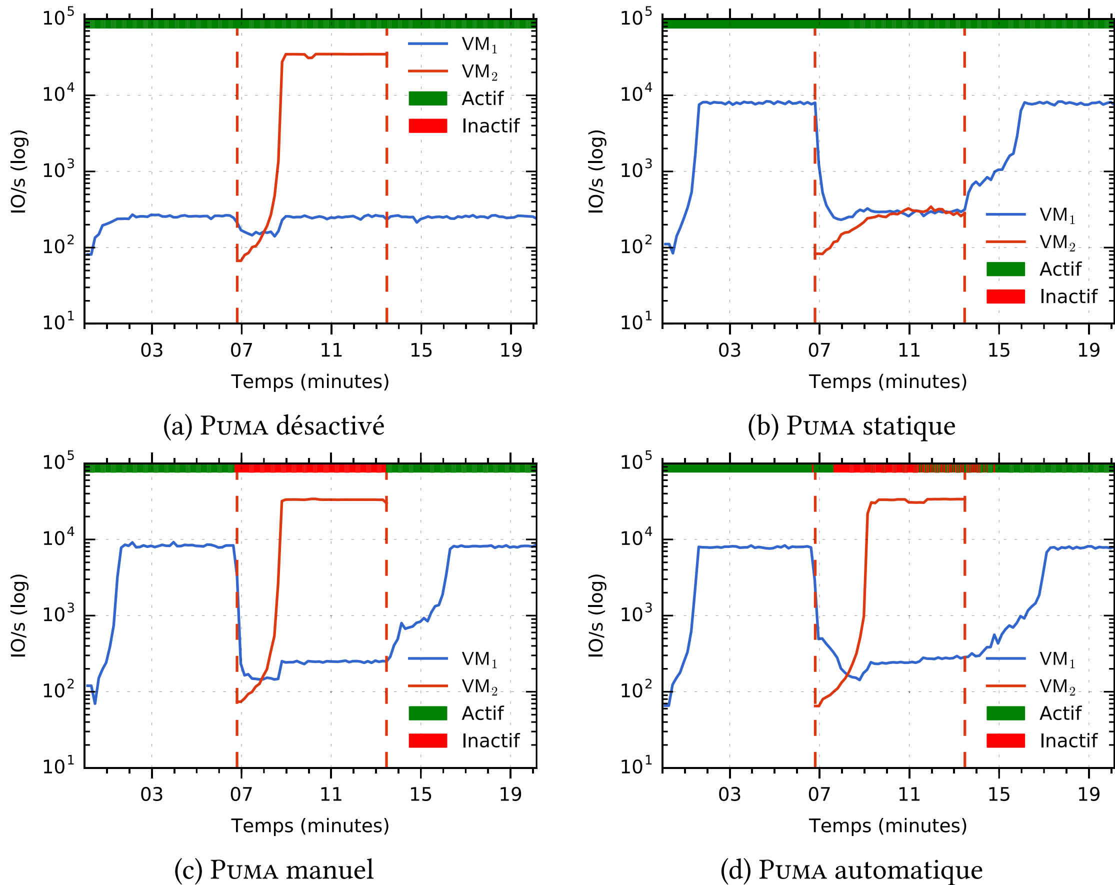
Nous mesurons dans cette section l'efficacité de la solution présentée dans la section 6.4.3. Pour rappel, celle-ci est composée d'un mécanisme de détection d'activité cache reposant sur le *shadow page cache*, et d'un autre mécanisme reposant sur la désactivation des pages actives.

Nous utilisons le benchmark de lectures aléatoires sur les deux VMs pour générer une activité concurrente. Les lectures aléatoires sont effectuées dans un fichier de 512 Mo. La première VM possède 448 Mo de mémoire, le fichier ne tient donc pas dans le cache. La seconde VM possède 768 Mo de mémoire, ce qui permet au fichier d'y tenir pour obtenir des performances maximales. Ainsi, la VM_1 peut utiliser la mémoire de la VM_2 pour obtenir de meilleures performances, mais lorsque la VM_2 commence son activité elle a besoin de récupérer la mémoire qu'elle prête pour maximiser ses performances.

Pour cela, nous lançons ce benchmark sur la VM_1 pendant 20min. En parallèle, nous lançons au bout d'environ 7min (1/3 de 20min) le même benchmark sur VM_2, pendant une durée d'environ 7min. Le but étant de mesurer le surcout lié à Puma. Idéalement les performances du benchmark de la VM_2 devraient être proches des performances obtenues en l'absence de Puma.

Nous considérons 4 configurations différentes qui permettent de situer le surcout de Puma en version automatique.

- ***Puma désactivé*** : dans cette configuration Puma est absent, la VM_2 s'exécute en parfaite isolation de la VM_1 et devrait donc avoir les meilleures performances



(a) PUMA désactivé (b) PUMA statique (c) PUMA manuel (d) PUMA automatique

FIGURE 7.5 – Surcout du mécanisme de partage automatique

possibles. En revanche, les performances de la VM_1 sont limitées puisqu'elle ne bénéficie pas de la mémoire inutilisée de la VM_2.

— ***PUMA « statique »*** : cette configuration reprend PUMA tel que présenté dans le chapitre 4, où sa configuration est entièrement « statique » : la VM_1 bénéficie de la mémoire inutilisée de la VM_2, mais celle-ci n'est pas capable de la récupérer intelligemment.

— ***PUMA manuel*** : nous désactivons PUMA manuellement lorsque le benchmark de la VM_2 démarre, puis nous réactivons PUMA lorsqu'il se termine. Cette configuration permet de simuler une heuristique « parfaite » où PUMA est capable de détecter immédiatement un changement d'activité.

— ***PUMA automatique*** : PUMA détecte automatiquement un changement d'activité en composant les heuristiques présentées dans la section 6.4.3, basées sur le *shadow page cache* et la désactivation des pages actives.

Les résultats de ces expériences sont présentés dans la figure 7.5. Dans ces courbes, le benchmark de la VM_1 s'exécute pendant toute la durée de l'expérience. Le début et la fin du benchmark de la VM_2 est indiqué par des pointillés verticaux. Nous affichons

également sur le haut des figures l'état de Puma (actif/inactif).

Puma désactivé. La figure 7.5a présente les résultats de cette expérience avec la configuration sans Puma. Ici, la VM_2 utilise pleinement toute sa mémoire pour accélérer les E/S du benchmark de lectures aléatoires, jusqu'à atteindre 34 500 ES/s.

Les performances du benchmark de la VM_1 ne dépasseront pas 2 700 ES/s. Notons que, lors du démarrage du benchmark de la VM_2, les performances de la VM_1 chutent parce que le disque dur principal de la machine hôte est partagé entre les deux VMs. Lorsque la totalité des données manipulées par la VM_2 tiennent dans le cache, le disque dur redevient exclusivement utilisé par la VM_1 qui retrouve donc ses performances maximales.

Puma statique. Les performances de Puma en version statique sont présentées dans la figure 7.5b. Dans cette configuration, qui correspond à la version de Puma présentée dans le chapitre 4, la VM_2 n'est pas capable de récupérer efficacement la mémoire qu'elle prête à la VM_1 : ces résultats sont similaires à ceux observés dans la section 6.4 (figure 6.3).

Contrairement à l'expérience sans Puma (figure 7.5a), ici Puma permet d'améliorer les performances de la VM_2 : celle-ci atteint 8 000 ES/s lorsque la VM_2 n'a pas d'activité. En revanche, les performances de la VM_2 sont limitées par l'activité de la VM_1, la vitesse des lectures aléatoires ne dépassent pas 300 ES/s.

Puma manuel. Dans la version manuelle de Puma, nous désactivons Puma avant de démarrer le benchmark de lectures aléatoires sur la VM_2, et nous le réactivons lorsque ce benchmark est terminé. Cette expérience représente les performances qui seraient obtenues sur les deux VMs avec une heuristique parfaite, capable de détecter un changement d'activité immédiatement.

Les résultats pour cette configuration sont présentés dans la figure 7.5c. On remarque que les performances des lectures aléatoires de la VM_2 sont proches d'une configuration sans Puma, où la VM_1 est parfaitement isolée (figure 7.5a). On y observe un léger ralentissement de la VM_2 par rapport à la configuration sans Puma (3,5%). Ce ralentissement est causé par le surcout de Puma lors de l'éviction des pages distantes qu'il héberge : il doit, par exemple, informer la VM_1 de l'éviction des pages pour qu'elle mette à jour ses métadonnées.

Les performances des lectures aléatoires de la VM_1 sont identiques à Puma statique lorsque la VM_2 n'a pas d'activité, et identiques à la configuration sans Puma lorsque la VM_2 a une activité.

Puma automatique. La figure 7.5d présente les résultats de cette expérience avec l'heuristique que nous avons présentée dans la section 6.4.3. L'allure globale des performances des deux VMs est proche de ce qu'on obtient avec la configuration manuelle. On note cependant trois différences : (*i*) les performances maximales de la VM_2 arrivent plus tard (environ 30 secondes) qu'avec la configuration manuelle ; (*ii*) les performances des lectures aléatoires de la VM_1 diminuent en deux temps lorsque la VM_2 démarre son activité ; et (*iii*) le rétablissement des performances de la VM_1 est plus lent.

Ces différences sont liées au fait que notre heuristique ne peut détecter immédiatement qu'une activité intensive en E/S a commencé sur la VM_2. En effet, comme nous l'expliquions dans la section 6.5, il est nécessaire d'attendre qu'une page de la VM_2 récemment évincée soit accédée de nouveau pour que l'heuristique basée sur le *shadow page cache* détecte l'activité. De même, l'heuristique basée sur le rééquilibrage des listes LRU nécessite que la liste active atteigne 50% de la totalité du page cache.

Ainsi, le délai nécessaire pour détecter l'activité augmente le temps nécessaire pour que la VM_2 atteigne ses performances maximales, puisque pendant ce temps le cache est partagé entre les deux VMs.

De même, la première baisse de performance des lectures aléatoires de la VM_1 est liée à l'éviction d'une partie de ses pages distantes par la VM_2, elle doit donc effectuer des E/S sur le disque, plus lent. Ensuite, comme Puma reste activé plus longtemps que dans la configuration manuelle, la VM_1 continue de lui envoyer des pages ce qui lui permet de disposer d'un cache plus grand pendant plus de temps.

Cette étude de performances a montré que les mécanismes d'automatisation de Puma que nous avons introduits dans le chapite 6 permettent d'obtenir des performances proches d'une solution *idéale* qui détecterait un changement d'activité immédiatement.

7.4 Analyse du seuil de désactivation des pages

Nous avons vu dans la section 6.3.2 qu'il est nécessaire de modifier l'équilibre des listes LRU du noyau Linux si l'on veut pouvoir prêter plus de 50% de la mémoire. Pour cela, lorsque le nombre de pages distantes au sein de la LRU inactive dépasse un certain seuil nous déclenchons le mécanisme du *Page Frame Reclaiming Algorithm* (PFRA) chargé de désactiver un groupe de pages actives (généralement 32).

Pour évaluer l'impact du choix du seuil de désactivation des pages, nous avons reproduit l'expérience décrite dans la section 6.3.2 en faisant varier ce seuil. Pour rappel, cette expérience consiste à exécuter le benchmark de lectures aléatoires sur deux VMs (VM_1 et VM_2). Les lectures aléatoires sont effectuées dans un fichier de 512 Mo, il tient dans la mémoire de la VM_2, qui possède 768 Mo de mémoire, mais pas dans celle de la

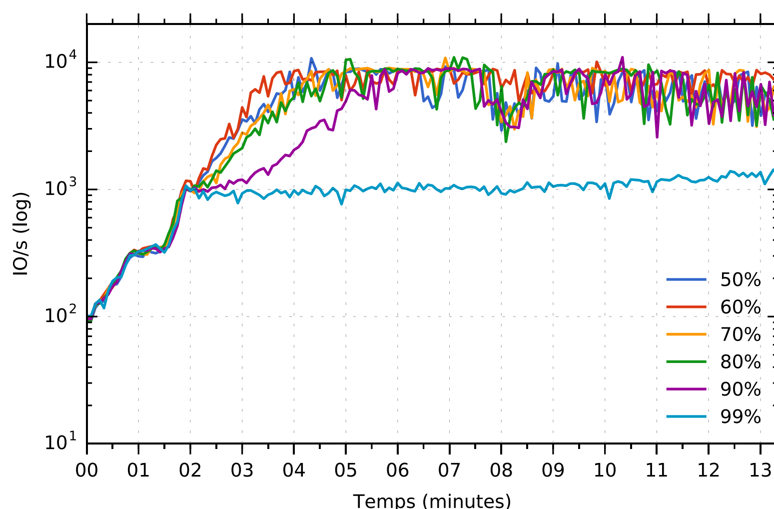


Figure 7.6 – Performance du benchmark de lectures aléatoires en faisant varier le seuil de désactivation des pages actives – VM_2 puis VM_1

VM_1, qui ne possède que 448 Mo de mémoire.

Dans cette expérience, nous exécutons d'abord le benchmark sur la VM_2 pendant environ 7min, puis sur la VM_1 pendant environ 14min. Ainsi, la VM_1 devra utiliser la mémoire inutilisée de la VM_2 afin d'obtenir des performances maximales. Nous faisons varier le seuil de désactivation des pages actives, de 50% à 99%. Nous présentons les performances des lectures aléatoires de la VM_1 dans la figure 7.6.

Sur ces résultats, on remarque d'abord une première phase pendant laquelle ce seuil n'a aucun impact. Il s'agit du temps nécessaire pour que la VM_1 ait remplit son propre cache local et commence à envoyer des pages à la VM_2. Ceci s'observe très bien lorsque l'on regarde le nombre de pages actives et de pages hébergées (distantes) de la VM_2 présenté dans les figures 7.7 et 7.8. Sur ces courbes, on remarque que des pages sont envoyées sur la VM_2 au bout de 1'30".

Une seconde phase peut être observée sur ces courbes, entre 1'30" et 2'. En effet, sur la figure 7.6 on remarque une augmentation brutale des performances, peu importe le seuil de désactivation des pages choisi. Cette période correspond au moment où la VM_1 bénéficie du cache de la VM_2, jusqu'à atteindre le seuil de désactivation des pages que nous avons configuré.

Enfin, lorsque ce seuil est dépassé des pages actives peuvent être désactivées pour fournir plus de mémoire à VM_1 : c'est ce qu'on observe sur toutes les figures à partir de 2'. Le choix de ce seuil est déterminant pour les performances de la VM_1. En effet, un seuil trop élevé (99%) ne permet pas de désactiver des pages actives suffisamment souvent, la taille du cache offerte augmente donc lentement. À l'inverse, un seuil plus bas permet de désactiver des pages actives plus souvent, pour le bénéfice de la VM_1.

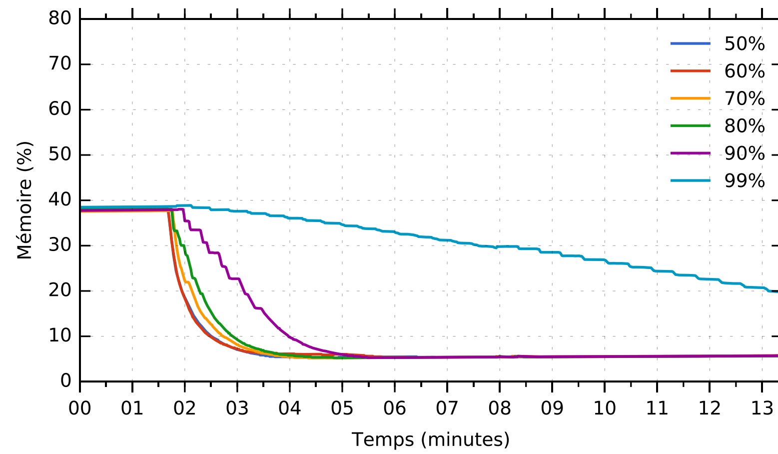


FIGURE 7.7 – Variation du nombre de pages actives sur la VM_2

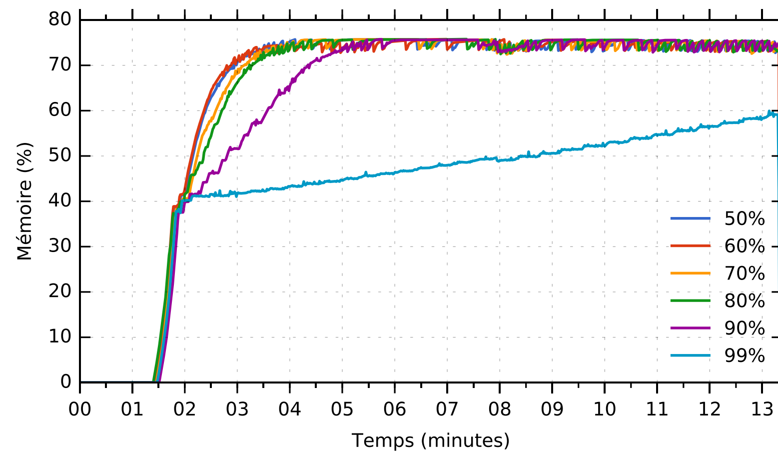


FIGURE 7.8 – Variation du nombre de pages distantes hébergées par la VM_2

On remarque cependant qu'en dessous de 80% le gain observé est négligeable.

7.5 Conclusion

Ce chapitre a présenté une évaluation des mécanismes permettant d'automatiser PUMA que nous avons présenté dans le chapitre 6. Nous avons montré que l'intégration de PUMA au sein des listes LRU du noyau Linux permettait de récupérer efficacement la mémoire pour des pages anonymes. En particulier, PUMA est capable de récupérer la mémoire 10 fois plus rapidement que des approches agnostiques au page cache telles que le ballooning.

Lorsque PUMA doit récupérer la mémoire prêtée pour son propre cache, nous avons montré que nos heuristiques (combinées), en permettant de détecter une reprise d'activité, permettaient d'atteindre des performances proches d'une heuristique idéale, où la reprise d'activité serait détectée immédiatement. Cependant, il reste difficile de détecter à l'avance une reprise d'activité, c'est pourquoi il subsiste un délai de plusieurs dizaines de secondes entre la reprise et sa détection.

Chapitre 8

Conclusions et perspectives

Sommaire

8.1 **Synthèse** 105
8.2 **Perspectives** 107

8.1 Synthèse

Les travaux présentés dans cette thèse s'inscrivent dans le contexte du *cloud computing* et des environnements virtualisés. Les architectures de type cloud reposent en grande partie sur l'utilisation de la virtualisation. C'est en effet un élément clé, qui permet au cloud d'offrir flexibilité, portabilité et isolation. Un des usages de la virtualisation consiste à remplacer une multitude de petits serveurs physiques par un seul, puissant, sur lequel sera déployé autant de machines virtuelles (VMs) que d'applications à isoler. L'avantage d'une telle approche est indéniable : réduction des couts et de la consommation énergétique (moins de serveurs), simplification de la maintenance (migrations), élasticité (ajout ou suppression de VMs), etc.

L'isolation entre les applications apportée par la virtualisation a cependant un prix : la fragmentation de la mémoire. En effet, la quantité de mémoire allouée à une VM par l'administrateur dépend souvent du pire cas estimé, par exemple lors des pics de charge. En dehors de ces périodes, la mémoire allouée aux VMs est sous-utilisée. Si les systèmes d'exploitation exploitent cette mémoire inutilisée, par exemple pour accélérer les entrées/sorties (E/S) en faisant du cache, cela ne bénéficie qu'aux applications dont les performances sont étroitement liées à l'efficacité des E/S. Ainsi, les VMs dont les performances ne reposent pas sur des E/S efficaces « gaspillent » cette mémoire, qui

pourrait être utilisée par les autres VMs.

Les travaux présentés dans cette thèse ont porté sur des mécanismes permettant de réutiliser intelligemment la mémoire *inutilisée* des VMs pour le bénéficie des VMs dont les performances des applications sont directement liées à l'efficacité des E/S. Pour cela, la solution que nous proposons intégrera les propriétés suivantes :

— la diversité des environnements et des applications déployés dans les VMs nécessite un mécanisme *générique*, capable de fonctionner efficacement sans nécessiter d'adapter les applications existantes ou l'environnement dont elles dépendent (*i.e.*, système de fichiers) ;
— la variabilité de la charge des VMs, en particulier dans les clouds, nécessite un mécanisme capable de s'adapter *automatiquement* pour prêter ou récupérer sa mémoire en fonction des besoins des VMs.

Pour répondre à ces problématiques, nous avons proposé Puma [1] [TSI-2015, SYSTOR'15, ComPAS'2014], un système de cache réparti capable de mutualiser la mémoire inutilisée des VMs à l'échelle d'un centre de données. Notre approche au niveau du noyau des VMs est indépendante des périphériques blocs, des systèmes de fichiers et de l'hyperviseur. Puma repose sur la présence d'un réseau performant entre les VMs, que l'on retrouve naturellement entre des VMs colocalisées et facilement entre les serveurs des centres de données d'aujourd'hui. Cette approche « réseau » permet à Puma de fonctionner à la fois localement, entre des VMs hébergées sur le même hôte, et en réseau, entre des VMs hébergées sur plusieurs hôtes.

Une des caractéristiques clé de Puma est que la mémoire n'est pas « donnée » d'une VM à une autre, mais « prêtée » : les VMs s'échangent des *copies* des pages, mais restent maîtres de leur propre mémoire. Contrairement au *memory ballooning*, nous avons choisi de ne traiter que les pages *propres*, ce qui permet de simplifier la gestion de la cohérence et de récupérer facilement la mémoire prêtée à d'autres VMs. De plus, Puma est directement intégré au sein du *page cache* du noyau Linux, ce qui permet (*i*) de récupérer rapidement la mémoire prêtée si un processus local en a besoin ; et (*ii*) si une VM qui prête sa mémoire a besoin de récupérer son cache, nous donnons une priorité plus faible aux pages *hébergées* pour qu'elles soient évincées plus rapidement.

Une des difficultés majeures dans la réalisation d'un tel cache est de ne pas dégrader les performances. Nous avons identifié plusieurs scénarios pouvant dégrader les performances, notamment lors d'une panne ou d'un ralentissement d'un nœud Puma, lors d'accès séquentiels et lors d'une variation d'activité sur un nœud, par exemple lors d'un pic d'activités. Ce dernier scénario a fait l'objet d'une contribution dédiée [Compas'2015] (chapitres 6 et 7).

Pour éviter les dégradations de performances liées à des variations d'activité, nous avons défini deux métriques nous permettant de détecter une reprise d'activité d'une

1. Pour *Pooling Unused memory in virtual MAchines*

VM qui offre sa mémoire au cache réparti. Ces métriques reposent sur deux mécanismes existants de la gestion de la mémoire du noyau Linux : le rééquilibrage des listes actives/inactives et le *shadow page cache*. Lorsqu'une reprise d'activité est détectée, PUMA est désactivé temporairement ce qui lui permet de récupérer sa mémoire plus rapidement et de ne pas être dérangé par d'autres VMs qui voudraient lui en emprunter.

Nous avons montré grâce à une évaluation approfondie que notre approche permet aux applications qui effectuent beaucoup d'E/S d'utiliser dynamiquement la mémoire inutilisée d'autres VMs pour améliorer leurs performances. Nos évaluations ont reposé sur les benchmarks TPC-C, TPC-H, Postmark et BLAST en utilisant des VMs colocalisées ou hébergées sur des hôtes différents. Ces évaluation démontrent l'efficacité qu'a PUMA à accélérer de façon importante les performances d'applications qui effectuent beaucoup d'E/S, de +12% en récupérant 500 Mo de mémoire inutilisée, et jusqu'à 4 fois plus en réutilisant 6 Go de mémoire dans le cas de TPC-C. Nous avons également montré qu'une VM était capable de récupérer rapidement la mémoire qu'elle « prête » à une autre, jusqu'à 10 fois plus rapidement qu'une approche reposant sur du *ballooning* automatique. Lors de nos évaluations, PUMA était capable de détecter une variation d'activité, notamment en termes d'E/S, ce qui permet de désactiver le service de cache réparti pour éviter de dégrader les performances.

8.2 Perspectives

Ces travaux ouvrent plusieurs perspectives, à plus ou moins court terme, que nous détaillons dans cette section.

Amélioration de la détection de l'activité

Les perspectives à court terme concernent l'amélioration de la détection de l'activité d'une VM, que nous avons présenté dans le chapitre 6 et évalué dans le chapitre 7. Il est en effet difficile de détecter à quel moment le *page cache* redevient important pour les performances des applications. Nos évaluations ont montré que les métriques choisies impliquent nécessairement un délai avant la détection, qui peut être non négligeable si la VM possède une grande quantité de mémoire. Il sera nécessaire d'approfondir ces évaluations, notamment à l'aide de traces et d'applications réelles, pour déterminer dans quelle mesure ce délai de détection impacte les performances des applications qui effectuent beaucoup d'E/S.

Gestion des écritures synchrones

Dans cette thèse, nous avons choisi de ne pas traiter les écritures dans le cache réparti, d'une part afin de simplifier le problème, et d'autre part parce que la majorité des accès en écritures sont différés dans le cache local (voir chapitre 4). Cependant, les écritures synchrones sont essentielles pour les applications qui ont de fortes contraintes de cohérence et de durabilité, telles que les bases de données ou les systèmes de stockage de type clé/valeur.

Une extension de Puma aux écritures est possible, mais il est alors nécessaire de mettre en place des mécanismes plus complexes de tolérance aux fautes et de cohérence de données. En effet, si un nœud effectue une écriture *synchrone* dans le cache réparti, cette écriture doit *à terme* être propagée sur le périphérique de stockage concerné. Ceci implique soit que le cache réparti (*i.e.*, les nœuds qui le composent) ait un accès aux périphériques de stockage des autres nœuds, soit qu'on limite la possibilité d'écrire dans le cache aux environnements reposants sur un périphérique de stockage partageable entre les nœuds (*i.e.*, iSCSI, système de fichiers distribué, etc.). De plus, des mécanismes de redondance des données du cache réparti doivent être mis en place pour tolérer les fautes des nœuds qui hébergent ces données. Ces différents systèmes auront nécessairement un impact sur les performances du cache réparti.

Paravirtualisation de Puma

Un des défauts majeurs de Puma est que la communication entre les VMs se fait *via* un réseau TCP/IP. L'inconvénient est ici la latence supplémentaire, même lorsque les VMs sont colocalisées. Comme nous l'avons vu dans le chapitre 5, le cout de la latence réseau n'est pas négligeable, et est amplifié par le surcout de la virtualisation [36, 40, 86]. Une approche plus efficace consisterait à communiquer directement avec les autres VMs en utilisant un moyen de communication *paravirtualisé*, reposant par exemple sur l'API *VirtIO* [84]. Il serait alors possible d'échanger directement deux pages mémoire des VMs, plutôt que d'envoyer des copies des pages en réseau. L'hyperviseur jouerait alors le rôle de coordinateur pour garantir l'échange des pages entre les VMs.

Ce type d'approche pourrait facilement être intégré dans Puma. En effet, la couche de communication réseau est indépendante des différents mécanismes que nous avons décrits dans le chapitre 4, il serait donc possible de la remplacer par une couche de communication *paravirtualisée*. Cependant, cette opération aurait un cout non négligeable :

— il serait nécessaire de modifier l'hyperviseur pour coordonner les échanges, Puma deviendrait donc limité à un hyperviseur spécifique ;
— il ne serait plus possible de mutualiser la mémoire de VMs qui sont hébergées sur des hôtes différents, à moins de conserver les deux couches de communication

et de détecter si deux VMs sont colocalisées.

Vers un ballooning adapté au cache

Une approche, concurrente à celle que nous proposons avec PUMA, consisterait à réaliser un ballooning spécialisé pour du cache. En effet, le problème majeur du ballooning classique est qu'il ne fait pas la différence entre les pages du page cache et les pages anonymes (*gap sémantique*). C'est pour cette raison qu'il lui est difficile de *récupérer* de la mémoire après l'avoir attribuée à une VM : si celle-ci s'en sert pour des pages anonymes, elle refusera de la rendre.

Pour réaliser un tel ballooning, nous pourrions contraindre le *driver* du ballon, qui se situe dans la VM, à ne confier les pages qu'il reçoit lorsqu'il se dégonfle au reste du système que si elles servent au page cache. Ainsi, il y aurait toujours un minimum de pages dans le page cache qui seraient réclamables : l'hyperviseur sait alors qu'il peut demander à la VM de gonfler son ballon d'autant qu'il lui a permis de se dégonfler précédemment.

De même que pour la paravirtualisation de PUMA, le défaut de cette solution est de se limiter aux VMs hébergées par le même hôte. Une approche *hybride* serait de combiner PUMA (actuel) pour mutualiser la mémoire inutilisée entre des VMs hébergées sur des hôtes différents avec cette approche lorsque les VMs en jeu sont colocalisées.

Bibliographie

[1] Keith ADAMS et Ole AGESEN. « A Comparison of Software and Hardware Techniques for x86 Virtualization ». In : *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS XII. San Jose, California, USA : ACM, 2006, p. 2–13.

[2] Jeongseob AHN, Changdae KIM, Jaeung HAN, Young-Ri CHOI et Jaehyuk HUH. « Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources ». In : *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing.* HotCloud'12. Boston, MA : USENIX Association, 2012, p. 19–19.

[3] Stephen F. ALTSCHUL, Warren GISH, Webb MILLER, Eugene W. MYERS et David J. LIPMAN. « Basic local alignment search tool ». In : *Journal of molecular biology* 215.3 (1990), p. 403–410.

[4] AMAZON. *Aws reference architecture.* `http://aws.amazon.com/architecture/`. 2015.

[5] Nadav AMIT, Dan TSAFRIR et Assaf SCHUSTER. « VSwapper : A Memory Swapper for Virtualized Environments ». In : *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '14. Salt Lake City, Utah, USA : ACM, 2014, p. 349–366.

[6] T. E. ANDERSON, M. D. DAHLIN, J. M. NEEFE, D. A. PATTERSON, D. S. ROSELLI et R. Y. WANG. « Serverless Network File Systems ». In : *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles.* SOSP '95. Copper Mountain, Colorado, USA : ACM, 1995, p. 109–126.

[7] Siddhartha ANNAPUREDDY, Michael J. FREEDMAN et David MAZIÈRES. « Shark : scaling file servers via cooperative caching ». In : *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation.* T. 2. NSDI'05. Berkeley, CA, USA, 2005, p. 129–142.

[8] ARTEAGA, DULCARDO, OTSTOTT, DOUGLAS et ZHAO, MING. « Dynamic Block-level Cache Management for Cloud Computing Systems ». In : *Proceedings of the 10th USENIX Conference on File and Storage Technologies.* FAST'12. San Jose, CA : USENIX Association, 2012.

[9] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff et John K. Ousterhout. « Measurements of a Distributed File System ». In : *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles.* SOSP '91. Pacific Grove, California, USA : ACM, 1991, p. 198–212.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt et Andrew Warfield. « Xen and the Art of Virtualization ». In : *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles.* SOSP '03. Bolton Landing, NY, USA : ACM, 2003, p. 164–177.

[11] Alexandros Batsakis et Randal Burns. « Cluster Delegation : High-performance, Fault-tolerant Data Sharing in NFS ». In : *Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium.* HPDC '05. Washington, DC, USA : IEEE Computer Society, 2005, p. 100–109.

[12] Alexandros Batsakis et Randal Burns. « NFS-CD : Write-Enabled Cooperative Caching in NFS ». In : *IEEE Transactions on Parallel and Distributed Systems* 19.3 (mar. 2008), p. 323–333.

[13] László A. Bélády. « A Study of Replacement Algorithms for a Virtual-storage Computer ». In : *IBM Systems Journal* 5.2 (juin 1966), p. 78–101.

[14] Mihir Bellare et Daniele Micciancio. « A New Paradigm for Collision-free Hashing : Incrementality at Reduced Cost ». In : *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques.* EUROCRYPT'97. Konstanz, Germany : Springer-Verlag, 1997, p. 163–192.

[15] Robert Birke, Mathias Björkqvist, Lydia Y. Chen, Evgenia Smirni et Ton Engbersen. « (Big)Data in a Virtualized World : Volume, Velocity, and Variety in Cloud Datacenters ». In : *Proceedings of the 12th USENIX Conference on File and Storage Technologies.* FAST'14. Santa Clara, CA : USENIX Association, 2014, p. 177–189.

[16] Robert Birke, Andrej Podzimek, Lydia Y. Chen et Evgenia Smirni. « State-of-the-practice in Data Center Virtualization : Toward a Better Understanding of VM Usage ». In : *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* DSN '13. Washington, DC, USA : IEEE Computer Society, 2013, p. 1–12.

[17] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi et Iréa Touche. « Grid'5000 : A Large Scale And Highly Reconfigurable Experimental Grid Testbed ». In : *International Journal of High Performance Computing Applications* 20.4 (nov. 2006), p. 481–494.

[18] Daniel P. Bovet et Marco Cesati. *Understanding the Linux kernel.* O'Reilly Media, 2005.

[19] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, Frans Kaashoek, Robert Morris et Nickolai Zeldovich. « An Analysis of Linux Scalability to Many Cores ». In : *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.* OSDI'10. Vancouver, BC, Canada : USENIX Association, 2010, p. 1–8.

[20] Edouard Bugnion, Scott Devine et Mendel Rosenblum. « Disco : Running Commodity Operating Systems on Scalable Multiprocessors ». In : *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles.* SOSP '97. Saint Malo, France : ACM, 1997, p. 143–156.

[21] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman et Edward Y. Wang. « Bringing Virtualization to the x86 Architecture with the Original VMware Workstation ». In : *ACM Transactions on Computer Systems* 30.4 (nov. 2012), 12 :1–12 :51.

[22] Arthur W. Burks, Herman H. Goldstine et John von Neumann. « Preliminary Discussion of the Logical Design of an Electronic Computing Instrument ». In : *The Institute for Advanced Study* (1946).

[23] Luiz Capitulino. *Automatic Memory Ballooning.* KVM Forum. 2013.

[24] Peter M. Chen et Brian D. Noble. « When Virtual Is Better Than Real ». In : *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems.* HOTOS '01. Washington, DC, USA : IEEE Computer Society, 2001, p. 133–.

[25] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott et Berni Schiefer. « Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems ». In : *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.* SIGMETRICS '05. Banff, Alberta, Canada : ACM, 2005, p. 145–156.

[26] Douglas Comer et Jim Griffioen. « A New Design for Distributed Systems : The Remote Memory Model ». In : *Proceedings of the Usenix Summer 1990 Technical Conference, Anaheim, California, USA, June 1990.* 1990, p. 127–136.

[27] Jonathan Corbet. *Better active/inactive list balancing.* `http://lwn.net/Articles/495543`. 2012.

[28] Toni Cortes, Sergi Girona et Jesús Labarta. « Avoiding the Cache-Coherence Problem in a Parallel/Distributed File System ». In : *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking.* HPCN Europe '97. London, UK, UK : Springer-Verlag, 1997, p. 860–869.

[29] Toni Cortes, Sergi Girona et Jesús Labarta. « Design Issues of a Cooperative Cache with No Coherence Problems ». In : *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems.* IOPADS '97. San Jose, California, USA : ACM, 1997, p. 37–46.

[30] Francisco Matias Cuenca-Acuna et Thu D. Nguyen. « Cooperative Caching Middleware for Cluster-Based Servers ». In : *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*. HPDC '01. Washington, DC, USA : IEEE Computer Society, 2001, p. 303–314.

[31] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson et David A. Patterson. « Cooperative caching : using remote client memory to improve file system performance ». In : *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. OSDI '94. Berkeley, CA, USA, 1994.

[32] Peter J. Denning. « The Locality Principle ». In : *Communications of the ACM* 48.7 (juil. 2005), p. 19–24.

[33] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy et Chandramohan A. Thekkath. « Implementing Global Memory Management in a Workstation Cluster ». In : *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA : ACM, 1995, p. 201–212.

[34] Brad Fitzpatrick. « Distributed Caching with Memcached ». In : *Linux Journal* 2004.124 (août 2004).

[35] Gartner. *Gartner Says Worldwide Public Cloud Services Market to Total $131 Billion*. http://www.gartner.com/newsroom/id/2352816. Février 2013.

[36] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster et Dan Tsafrir. « ELI : Bare-metal Performance for I/O Virtualization ». In : *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK, 2012, p. 411–422.

[37] Cary G. Gray et David R. Cheriton. « Leases : An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency ». In : *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*. SOSP '89. New York, NY, USA : ACM, 1989, p. 202–210.

[38] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker et Amin Vahdat. « Difference Engine : Harnessing Memory Redundancy in Virtual Machines ». In : *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California : USENIX Association, 2008, p. 309–322.

[39] Hyuck Han, Young Choon Lee, Woong Shin, Hyungsoo Jung, Heon Y. Yeom et Albert Y. Zomaya. « Cashing in on the Cache in the Cloud ». In : *IEEE Transactions on Parallel and Distributed Systems* 23.8 (août 2012), p. 1387–1399.

[40] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger et Razya Ladelsky. « Efficient and Scalable Paravirtual I/O System ». In : *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA : USENIX Association, 2013, p. 231–242.

[41] Stephen HEMMINGER. « Netem-emulating real networks in the lab ». In : *Proceedings of the 2005 Linux Conference Australia, Canberra, Australia*. 2005.

[42] John L. HENNESSY et David A. PATTERSON. *Computer Architecture, Fifth Edition : A Quantitative Approach*. 5th. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011.

[43] Jin HEO, Xiaoyun ZHU, Pradeep PADALA et Zhikui WANG. « Memory Overbooking and Dynamic Control of Xen Virtual Machines in Consolidated Environments ». In : *Proceedings of the 11th IFIP/IEEE International Conference on Symposium on Integrated Network Management*. IM'09. New York, NY, USA : IEEE Press, 2009, p. 630–637.

[44] Maurice HERLIHY et Nir SHAVIT. *The Art of Multiprocessor Programming*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008.

[45] Fabien HERMENIER, Julia LAWALL et Gilles MULLER. « BtrPlace : A Flexible Consolidation Manager for Highly Available Applications ». In : *IEEE Trans. Dependable Secur. Comput.* 10.5 (sept. 2013), p. 273–286.

[46] Jinho HWANG, Ahsen UPPAL, Timothy WOOD et Howie HUANG. « Mortar : Filling the Gaps in Data Center Memory ». In : *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '14. Salt Lake City, Utah, USA : ACM, 2014, p. 53–64.

[47] IBM. *Best practices for KVM*. White Paper. Nov. 2010.

[48] « IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7 ». In : *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (déc. 2008), p. c1–3826.

[49] Florin ISAILA, Guido MALPOHL, Vlad OLARU, Gabor SZEDER et Walter TICHY. « Integrating Collective I/O and Cooperative Caching into the "Clusterfile" Parallel File System ». In : *Proceedings of the 18th Annual International Conference on Supercomputing*. ICS '04. Saint Malo, France : ACM, 2004, p. 58–67.

[50] Aamer JALEEL, Eric BORCH, Malini BHANDARU, Simon C. STEELY JR. et Joel EMER. « Achieving Non-Inclusive Cache Performance with Inclusive Caches : Temporal Locality Aware (TLA) Cache Management Policies ». In : *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. Washington, DC, USA : IEEE Computer Society, 2010, p. 151–162.

[51] Theodore JOHNSON et Dennis SHASHA. « 2Q : A Low Overhead High Performance Buffer Management Replacement Algorithm ». In : *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1994, p. 439–450.

[52] Stephen T. JONES, Andrea C. ARPACI-DUSSEAU et Remzi H. ARPACI-DUSSEAU. « Geiger : Monitoring the Buffer Cache in a Virtual Machine Environment ». In : *Proceedings of the 12th International Conference on Architectural Support for*

Programming Languages and Operating Systems. ASPLOS XII. San Jose, California, USA : ACM, 2006, p. 14–24.

[53] Jeffrey Katcher. *Postmark : A new file system benchmark.* Rapp. tech. TR3022. Network Appliance, 1997.

[54] Taeho Kgil et Trevor Mudge. « FlashCache : a NAND flash memory file cache for low power web servers ». In : *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems.* ACM, 2006, p. 103–112.

[55] Hwanju Kim, Heeseung Jo et Joonwon Lee. « XHive : Efficient Cooperative Caching for Virtual Machines ». In : *IEEE Transactions on Computers* 60.1 (jan. 2011), p. 106–119.

[56] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin et Anthony Liguori. « kvm : the Linux virtual machine monitor ». In : *Proceedings of the Linux Symposium.* T. 1. 2007, p. 225–230.

[57] Jacob Faber Kloster, Jesper Kristensen et Arne Mejlholm. *Determining the use of interdomain shareable pages using kernel introspection.* Rapp. tech. Technical report, Aalborg University, 2007.

[58] Damien Le Moal, Zvonimir Bandic et Cyril Guyot. « Shingled file system host-side management of Shingled Magnetic Recording disks ». In : *IEEE International Conference on Consumer Electronics (ICCE).* Jan. 2012, p. 425–426.

[59] Changman Lee, Dongho Sim, Joo-Young Hwang et Sangyeun Cho. « F2FS : A New File System for Flash Storage ». In : *Proceedings of the 13th USENIX Conference on File and Storage Technologies.* FAST'15. Santa Clara, CA : USENIX Association, 2015, p. 273–286.

[60] Avraham Leff, Joel L. Wolf et Philip S. Yu. « Replication Algorithms in a Remote Caching Architecture ». In : *IEEE Transactions on Parallel and Distributed Systems* 4.11 (nov. 1993), p. 1185–1204.

[61] Avraham Leff, Philip S. Yu et Joel L. Wolf. « Policies for Efficient Memory Utilization in a Remote Caching Architecture ». In : *Proceedings of the First International Conference on Parallel and Distributed Information Systems.* PDIS '91. Miami, Florida, USA : IEEE Computer Society Press, 1991, p. 198–209.

[62] *Les rectifications de l'orthographe.* Journal officiel de la république française. Académie française. Déc. 1990.

[63] Andrew W. Leung, Shankar Pasupathy, Garth Goodson et Ethan L. Miller. « Measurement and Analysis of Large-scale Network File System Workloads ». In : *USENIX 2008 Annual Technical Conference.* ATC'08. Boston, Massachusetts : USENIX Association, 2008, p. 213–226.

[64] John S. Liptay. « Structural Aspects of the System/360 Model 85 : II the Cache ». In : *IBM Systems Journal* 7.1 (mar. 1968), p. 15–21.

[65] Adam LITKE. *Manage resources on overcommitted KVM hosts.* http://www.ibm.com/developerworks/library/l-overcommit-kvm-resources/. IBM, 2011.

[66] Dan MAGENHEIMER. « Transcendent memory in a nutshell ». In : *LWN.net* (août 2011).

[67] Dan MAGENHEIMER. « Zcache and RAMster ». In : *Linux Storage, Filesystem and Memory Management Summit.* 2012.

[68] Dan MAGENHEIMER, Chris MASON, Dave MCCRACKEN et Kurt HACKEL. « Transcendent memory and linux ». In : *Proceedings of the 11th Linux Symposium.* Montréal, Québec, Canada, 2009, p. 191–200.

[69] Avantika MATHUR, Mingming CAO, Suparna BHATTACHARYA, Andreas DILGER, Alex TOMAS et Laurent VIVIER. « The new ext4 filesystem : current status and future plans ». In : *Proceedings of the Linux Symposium.* T. 2. 2007, p. 21–33.

[70] Jim MAURO, Spencer SHEPLER et Vasily TARASOV. *Filebench.* http://sourceforge.net/projects/filebench/.

[71] Dirk MERKEL. « Docker : Lightweight Linux Containers for Consistent Development and Deployment ». In : *Linux Journal* 2014.239 (mar. 2014).

[72] Konrad MILLER, Fabian FRANZ, Thorsten GROENINGER, Marc RITTINGHAUS, Marius HILLENBRAND et Frank BELLOSA. « KSM++ : Using I/O-based hints to make memory-deduplication scanners more efficient ». In : *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12).* 2012.

[73] Grzegorz MIŁÓS, Derek G. MURRAY, Steven HAND et Michael A. FETTERMAN. « Satori : Enlightened Page Sharing ». In : *Proceedings of the 2009 Conference on USENIX Annual Technical Conference.* USENIX'09. San Diego, California : USENIX Association, 2009.

[74] David C. van MOOLENBROEK, Raja APPUSWAMY et Andrew S. TANENBAUM. « Towards a Flexible, Lightweight Virtualization Alternative ». In : *Proceedings of International Conference on Systems and Storage.* SYSTOR 2014. Haifa, Israel : ACM, 2014, 8 :1–8 :7.

[75] James H. MORRIS, Mahadev SATYANARAYANAN, Michael H. CONNER, John H. HOWARD, David S. ROSENTHAL et F. Donelson SMITH. « Andrew : A Distributed Personal Computing Environment ». In : *Communications of the ACM* 29.3 (mar. 1986), p. 184–201.

[76] Jun NAKAJIMA et Asit K MALLICK. « Hybrid-virtualization : Enhanced virtualization for linux ». In : *Proceedings of the Linux Symposium.* T. 2. 2007, p. 87–96.

[77] Dushyanth NARAYANAN, Eno THERESKA, Austin DONNELLY, Sameh ELNIKETY et Antony ROWSTRON. « Migrating server storage to SSDs : analysis of tradeoffs ». In : *Proceedings of the 4th ACM European conference on Computer systems.* ACM, 2009, p. 145–158.

[78] Michael N. NELSON, Brent B. WELCH et John K. OUSTERHOUT. « Caching in the Sprite Network File System ». In : *ACM Transactions on Computer Systems* 6.1 (fév. 1988), p. 134–154.

[79] Kent OVERSTREET. *bcache : Linux kernel block layer cache.* `http://bcache.evilpiepirate.org/`. 2013.

[80] David A. PATTERSON, Garth GIBSON et Randy H. KATZ. « A Case for Redundant Arrays of Inexpensive Disks (RAID) ». In : *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. SIGMOD '88. Chicago, Illinois, USA : ACM, 1988, p. 109–116.

[81] Brian PAWLOWSKI, Spencer SHEPLER, Carl BEAME, Brent CALLAGHAN, Michael EISLER, David NOVECK, David ROBINSON et Robert THURLOW. « The NFS version 4 protocol ». In : *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*. 2000.

[82] K. T. PEDRETTI, T. L. CASAVANT, R. C. BRAUN, T. E. SCHEETZ, C. L. BIRKETT et C. A. ROBERTS. « Three Complementary Approaches to Parallelization of Local BLAST Service on Workstation Clusters ». en. In : *Parallel Computing Technologies*. Sous la dir. de Victor MALYSHKIN. Lecture Notes in Computer Science 1662. Springer Berlin Heidelberg, 1999, p. 271–282.

[83] Ohad RODEH, Josef BACIK et Chris MASON. « BTRFS : The Linux B-Tree Filesystem ». In : *ACM Transactions on Storage* 9.3 (août 2013), 9 :1–9 :32.

[84] Rusty RUSSELL. « Virtio : Towards a De-facto Standard for Virtual I/O Devices ». In : *SIGOPS Operating Systems Review* 42.5 (juil. 2008), p. 95–103.

[85] Tudor-Ioan SALOMIE, Gustavo ALONSO, Timothy ROSCOE et Kevin ELPHINSTONE. « Application Level Ballooning for Efficient Server Consolidation ». In : *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic : ACM, 2013, p. 337–350.

[86] Jose Renato SANTOS, Yoshio TURNER, G. JANAKIRAMAN et Ian PRATT. « Bridging the Gap Between Software and Hardware Techniques for I/O Virtualization ». In : *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. ATC'08. Boston, Massachusetts, 2008, p. 29–42.

[87] Prasenjit SARKAR et John HARTMAN. « Efficient Cooperative Caching Using Hints ». In : *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*. OSDI '96. Seattle, Washington, USA, 1996, p. 35–46.

[88] Mahadev SATYANARAYANAN, John H. HOWARD, David A. NICHOLS, Robert N. SIDEBOTHAM, Alfred Z. SPECTOR et Michael J. WEST. « The ITC Distributed File System : Principles and Design ». In : *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*. SOSP '85. Orcas Island, Washington, USA : ACM, 1985, p. 35–50.

[89] Joel H. SCHOPP, Keir FRASER et Martine J. SILBERMANN. « Resizing memory with balloons and hotplug ». In : *Proceedings of the Linux Symposium*. T. 2. 2006, p. 313–319.

[90] S. SHEPLER, B. CALLAGHAN, D. ROBINSON, R. THURLOW, C. BEAME, M. EISLER et D. NOVECK. *Network File System (NFS) version 4 Protocol.* RFC 3530 (Proposed Standard). Obsoleted by RFC 7530. Internet Engineering Task Force, avr. 2003.

[91] Liuba SHRIRA et Ben YODER. « Trust but Check : Mutable Objects in Untrusted Cooperative Caches ». In : *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3) : Advances in Persistent Object Systems.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999, p. 29–36.

[92] Mike SNITZER, Kent OVERSTREET, Alasdair KERGON et Darrick WONG. « dm-cache and bcache : the future of two storage-caching technologies for Linux ». In : *Linux Storage, Filesystem, and Memory Management Summit.* LSFMM. 2013.

[93] Stephen SOLTESZ, Herbert PÖTZL, Marc E. FIUCZYNSKI, Andy BAVIER et Larry PETERSON. « Container-based Operating System Virtualization : A Scalable, High-performance Alternative to Hypervisors ». In : *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007.* EuroSys '07. Lisbon, Portugal : ACM, 2007, p. 275–287.

[94] Michael STONEBRAKER et Lawrence A. ROWE. « The Design of POSTGRES ». In : *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data.* SIGMOD '86. Washington, D.C., USA : ACM, 1986, p. 340–355.

[95] Anand SURESH, Garth GIBSON et Greg GANGER. *Shingled Magnetic Recording for Big Data Applications.* Rapp. tech. CMU-PDL-12-105. Carnegie Mellon University, mai 2012.

[96] Vasily TARASOV, Saumitra BHANAGE, Erez ZADOK et Margo SELTZER. « Benchmarking File System Benchmarking : It *IS* Rocket Science ». In : *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems.* HotOS'13. Napa, California : USENIX Association, 2011, p. 9–9.

[97] *TPC-C.* Transaction Processing Performance Council. 2010.

[98] *TPC-H.* Transaction Processing Performance Council. 2014.

[99] Avishay TRAEGER, Erez ZADOK, Nikolai JOUKOV et Charles P. WRIGHT. « A Nine Year Study of File System and Storage Benchmarking ». In : *ACM Transactions on Storage* 4.2 (mai 2008), 5 :1–5 :56.

[100] Irina Chihaia TUDUCE et Thomas GROSS. « Adaptive Main Memory Compression ». In : *Proceedings of the 2005 USENIX Annual Technical Conference.* ATC '05. Anaheim, CA : USENIX Association, 2005, p. 237–250.

[101] Eric VAN HENSBERGEN et Ming ZHAO. *Dynamic policy disk caching for storage networking.* Rapp. tech. RC24123. IBM, 2006.

[102] Anthony VELTE et Toby VELTE. *Microsoft Virtualization with Hyper-V.* 1re éd. New York, NY, USA : McGraw-Hill, Inc., 2010.

[103] Akshat VERMA, Puneet AHUJA et Anindya NEOGI. « pMapper : Power and Migration Cost Aware Application Placement in Virtualized Systems ». In : *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware.*

Middleware '08. Leuven, Belgium : Springer-Verlag New York, Inc., 2008, p. 243–264.

[104] Carl A. Waldspurger. « Memory Resource Management in VMware ESX Server ». In : *Proceedings of the 5th Symposium on Operating Systems Design and implementation*. OSDI '02. Boston, Massachusetts : ACM, 2002, p. 181–194.

[105] Jon Watson. « VirtualBox : Bits and Bytes Masquerading As Machines ». In : *Linux J.* 2008.166 (fév. 2008).

[106] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long et Carlos Maltzahn. « Ceph : A Scalable, High-performance Distributed File System ». In : *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington : USENIX Association, 2006, p. 307–320.

[107] Johannes Weiner. *Thrash detection-based file cache sizing*. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a528910e12ec7ee203095eb1711468a66b9b60b0`. Avr. 2014.

[108] Maurice V. Wilkes. « Slave memories and dynamic storage allocation ». In : *IEEE Transactions on Electronic Computers* 2.EC-14 (1965), p. 270–271.

[109] Mark Wong. *OSDL DBT-2*. `http://sourceforge.net/projects/osdldbt/files/dbt2/0.40/`. Open Source Development Labs, 2007.

[110] Theodore M. Wong et John Wilkes. « My Cache or Yours ? Making Storage More Exclusive ». In : *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*. ATEC '02. Berkeley, CA, USA : USENIX Association, 2002, p. 161–175.

[111] David Woodhouse. « JFFS : The journalling flash file system ». In : *Ottawa Linux Symposium*. T. 2001. 2001.

[112] Ying Xu et Brett D. Fleisch. « NFS-Cc : Tuning NFS for Concurrent Read Sharing ». In : *International Journal of High Performance Computing and Networking* 1.4 (déc. 2004), p. 203–213.

[113] Gala Yadgar, Michael Factor, Kai Li et Assaf Schuster. « MC2 : Multiple Clients on a Multilevel Cache ». In : *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*. ICDCS '08. Washington, DC, USA : IEEE Computer Society, 2008, p. 722–730.

[114] Mohamed Zahran, Kursad Albayraktaroglu et Manoj Franklin. « Non-inclusion property in multi-level caches revisited ». In : *International Journal of Computers and Their Applications* 14.2 (2007), p. 99.

[115] Jenny Zhang, Mark Wong et Monty Taylor. *OSDL DBT-3*. `http://sourceforge.net/projects/osdldbt/files/dbt3/1.10/`. Open Source Development Labs, 2007.

[116] Weiming Zhao et Zhenlin Wang. « Dynamic Memory Balancing for Virtual Machines ». In : *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International*

Conference on Virtual Execution Environments. VEE '09. Washington, DC, USA : ACM, 2009, p. 21–30.

Maxime Lorrillere

Caches collaboratifs noyau adaptés aux environnements virtualisés

Résumé

Avec l'avènement du *cloud computing*, la virtualisation est devenue aujourd'hui incontournable. Elle offre isolation et flexibilité, mais implique une fragmentation des ressources, notamment de la mémoire. Les performances des applications qui effectuent beaucoup d'entrées/sorties (E/S) en sont particulièrement impactées. En effet, celles-ci reposent en grande partie sur la présence de mémoire libre, utilisée par le système pour faire du cache et ainsi accélérer les E/S. Ajuster dynamiquement les ressources d'une machine virtuelle devient donc un enjeu majeur.

Dans cette thèse nous nous intéressons à ce problème, et nous proposons Puma, un cache réparti permettant de mutualiser la mémoire inutilisée des machines virtuelles pour améliorer les performances des applications qui effectuent beaucoup d'E/S. Contrairement aux solutions existantes, notre approche noyau permet à Puma de fonctionner avec les applications sans adaptation ni système de fichiers spécifique. Nous proposons plusieurs métriques, reposant sur des mécanismes existants du noyau Linux, qui permettent de définir le niveau d'activité « cache » du système. Ces métriques sont utilisées par Puma pour automatiser le niveau de contribution d'un nœud au cache réparti. Nos évaluations de Puma montrent qu'il est capable d'améliorer significativement les performances d'applications qui effectuent beaucoup d'E/S et de s'adapter dynamiquement afin de ne pas dégrader leurs performances.

Abstract

With the advent of cloud architectures, virtualization has become a key mechanism for ensuring isolation and flexibility. However, a drawback of using virtual machines (VMs) is the fragmentation of physical resources. As operating systems leverage free memory for I/O caching, memory fragmentation is particularly problematic for I/O-intensive applications, which suffer a significant performance drop. In this context, providing the ability to dynamically adjust the resources allocated among the VMs is a primary concern.

To address this issue, this thesis proposes a distributed cache mechanism called Puma. Puma pools together the free memory left unused by VMs: it enables a VM to entrust clean page-cache pages to other VMs. Puma extends the Linux kernel page cache, and thus remains transparent, to both applications and the rest of the operating system. Puma adjusts itself dynamically to the caching activity of a VM, which Puma evaluates by means of metrics derived from existing Linux kernel memory management mechanisms. Our experiments show that Puma significantly improves the performance of I/O-intensive applications and that it adapts well to dynamically changing conditions.