ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

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

Maxime Lorrillere, Julien Sopena, Sébastien Monnet et Pierre Sens

Sorbonne Universités, UPMC Univ Paris 06, F-75005, Paris, France
CNRS, UMR_7606, LIP6, F-75005, Paris, France
Inria, Équipe Regal, F-75005, Paris, France
Email: prénom.nom@lip6.fr

Résumé
Certaines applications, comme les serveurs d'e-commerce, effectuent beaucoup d'opérations d'entrée/ sortie (E/S) sur disque. Leurs performances sont étroitement liées à l'efficacité de ces opérations. Pour les améliorer, les systèmes d'exploitation tels que Linux utilisent la mémoire libre pour fournir des mécanismes de cache. Cependant, les ressources d'aujourd'hui sont virtualisées : par exemple, dans les *clouds*, les machines virtuelles (MVs) peuvent être déployées et déplacées entre les hôtes physiques pour offrir à la fois isolation et flexibilité. Ceci implique une fragmentation des ressources physiques, dont la mémoire. Cette fragmentation limite la quantité de mémoire disponible d'une MV pour y mettre ses données en cache. Dans cet article, nous proposons PUMA, un mécanisme de cache distant qui permet à une MV d'étendre son cache en utilisant la mémoire d'autres MVs situées sur un hôte local ou distant. Nos évaluations montrent que PUMA permet de multiplier les performances des applications par 9.

Mots-clés : cache réparti, système d'exploitation, virtualisation, mémoire

1. Introduction

Les applications qui travaillent avec une grande quantité de données, comme les serveurs d'e-commerce ou les applications scientifiques telles que BLAST [3] ont leur performances directement liées aux performances des entrées/sorties (E/S). Pour améliorer l'efficacité des E/S, les systèmes d'exploitation (OS) tels que Linux se servent de la mémoire inutilisée par les processus pour offrir des mécanismes de cache. En conservant en mémoire les données déjà accédées, le système peut accélérer les accès ultérieurs à celles-ci en exploitant la localité des applications. Les *clouds* utilisent intensivement la virtualisation, qui sert de base pour offrir flexibilité, portabilité et isolation. En effet, la virtualisation permet de partager les ressources physiques, comme les serveurs, entre plusieurs machines virtuelles (MVs).
Cependant, la *fragmentation* des ressources est le prix à payer pour la virtualisation. La mémoire des serveurs physiques est partitionnée entre les MVs. Comme il est difficile de prévoir la quantité de mémoire nécessaire à une application, la quantité de mémoire allouée aux MVs est généralement surdimensionnée. Le problème vient du fait que la mémoire des MVs est allouée *statiquement* ; la quantité allouée est choisie parmi plusieurs configurations offertes par le fournisseur de cloud. Sur un hôte, la mémoire disponible pour le cache est ainsi partitionnée entre les MVs qu'il héberge. Dans ce contexte, mutualiser la mémoire inutilisée permettrait d'améliorer les performances générales des mécanismes de cache.
Plusieurs solutions permettent d'offrir plus de flexibilité dans la gestion de la mémoire, comme la déduplication [22, 5, 26], le *memory ballooning* [24, 26] ou les caches coopératifs [12]. Cependant, ces solutions sont fortement limitées. Le *memory ballooning* permet uniquement de redimensionner la mémoire des MVs sur un seul hôte, alors que la déduplication vise à optimiser la mémoire utilisée par les processus. Finalement, les caches coopératifs sont soit partie intégrante d'un système de fichiers réparti [4, 7, 11, 17], ce qui empêche les utilisateurs d'utiliser le système de fichiers qui correspond à leurs besoins ; soit associés à un périphérique bloc virtuel [25, 13], ce qui, à l'inverse, ne permet pas l'utilisation d'un système de

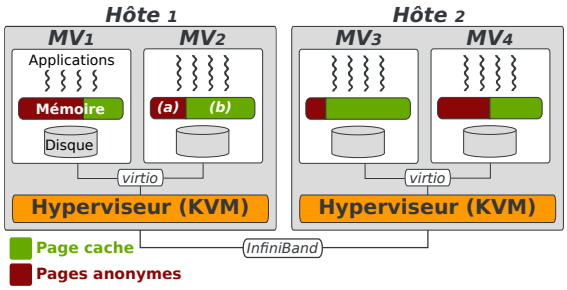


FIGURE 1 – Virtualisation et *page cache* du système d'exploitation

fichiers réparti. RAMster [21], qui a été intégré au noyau Linux, permet à une MV d'utiliser la mémoire d'autres MVs à la fois pour la mémoire des processus et pour le cache. Cependant, comme RAMster est invoqué en cas de forte contention mémoire, notre évaluation montre que RAMster est inefficace dans le cas général. Notre solution se concentre sur la mémoire utilisée pour mettre en cache les données des E/S, car nous supposons que la mémoire allouée aux MVs est suffisante pour contenir les processus applicatifs.

Dans cet article, nous proposons PUMA: un mécanisme de cache distant *générique* et *efficace* qui permet de mutualiser la mémoire inutilisée des MVs, utilisable avec des périphériques bloc mais aussi par des systèmes de fichiers répartis. Nous montrons à travers une évaluation approfondie que notre mécanisme permet aux applications d'utiliser la mémoire d'autres MVs pour améliorer leurs performances.

Nos contributions sont les suivantes :

— un mécanisme de cache distant implémenté au niveau noyau qui est : (*i*) indépendant du périphérique bloc, du système de fichiers et de l'hyperviseur ; et (*ii*) efficace à la fois localement et à distance ;

— une évaluation approfondie en utilisant à la fois des micro-benchmarks et des applications réelles, en tenant compte des caractéristiques des architectures utilisées pour le cloud.

Cet article est organisé comme suit. La section 2 présente les travaux connexes et montre leurs différences avec notre approche, puis la section 3 détaille l'architecture et l'implémentation de PUMA. La section 4 présente les résultats de nos expérimentations puis la section 5 conclut cet article.

2. État de l'art

Cette section présente un aperçu du contexte de ce travail, puis décrit les travaux connexes à PUMA.

2.1. Contexte

La virtualisation. La virtualisation permet de mutualiser les ressources d'un serveur physique. Un *hyperviseur* [26, 6] s'exécute sur un *hôte* physique et est responsable de la gestion des OS *invités* (MVs). Les OS invités peuvent s'exécuter sans modification (c.-à-d. virtualisation complète) au prix d'une dégradation des performances. La *paravirtualisation* est une technique de virtualisation qui présente des interfaces spécifiques aux MVs pour réduire le surcoût de la virtualisation. La paravirtualisation est particulièrement efficace pour améliorer les performances des E/S.

La figure 1 montre une configuration virtualisée classique avec 2 hôtes, chacun hébergeant 2 OS invités sur un hyperviseur (p. ex. *KVM*), avec un réseau paravirtualisé (p. ex. *virtio*). Utiliser un réseau paravirtualisé améliore les performances globales, mais permet aussi aux OS invités hébergés sur un même hôte de communiquer via un réseau haute performance.

Le page cache. Pour améliorer les performances des accès aux fichiers, les systèmes d'exploitation conservent les données lues depuis le disque dans un cache appelé le *page cache*. L'essentiel de la mémoire est remplie par des pages du page cache quand le système est inactif. Lorsque la charge augmente, la mémoire se remplit des pages des processus actifs, donc la taille du page cache se réduit. Les pages des processus sont appelées pages *anonymes*, à l'inverse des pages du page cache qui ont une représentation sur le disque. Dans la figure 1, nous avons représenté les pages anonymes (a) et les pages du page cache (b) pour chaque VM.

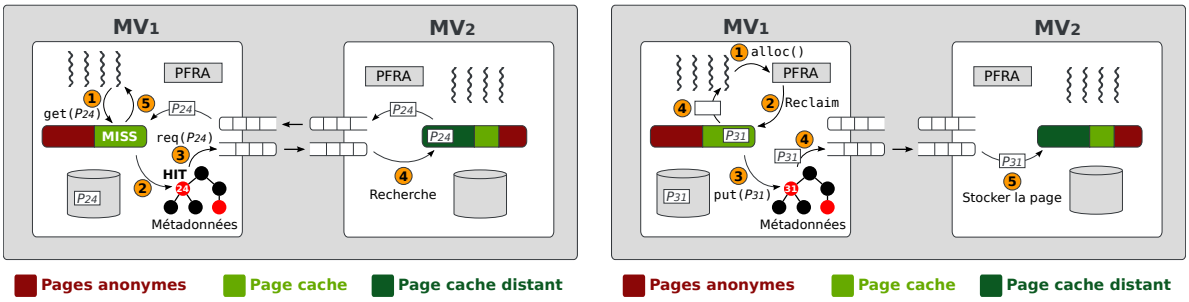
Le récupération de la mémoire. Lorsqu'il y a un besoin de mémoire, par exemple lorsqu'un processus

tente d'allouer de la mémoire ou lorsque le processus noyau *kswapd* se réveille, l'algorithme de récupération de mémoire (PFRA, pour *Page Frame Reclaiming Algorithm*) du noyau Linux libère des pages *victimes* de la mémoire. Pour cela, les pages anonymes et les pages du page cache sont stockées dans deux listes *LRU*. 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.-à-d. modifiées), elles doivent d'abord être écrites sur le disque. Lorsque des pages anonymes doivent être libérées, elles doivent être écrites sur le disque dans l'espace de *swap*.
Pour modifier la stratégie de cache, il est nécessaire de capturer les défauts et les évictions du page cache. Cependant, c'est une tâche difficile d'un point de vue technique car il y a des dizaines de chemins d'exécution qui accèdent au page cache. Heureusement, des travaux récents ont proposé l'API *cleancache* [21, 20]. Cette API fourni des *hooks* pour capturer à la fois les évictions (opération *put*) et les défauts (opération *get*) du page cache. Plusieurs implémentations de cette API existent, comme *zcache*, un page cache compressé, ou RAMster [20].

2.2. Positionnement

Beaucoup de travaux ont abordé le problème de l'amélioration de la gestion de la mémoire des MVs dans le cloud. La plupart de ces travaux permettent de partager la mémoire entre plusieurs machines physiques, et seuls quelques uns se sont concentrés sur l'optimisation de l'utilisation de la mémoire « inutilisée » (c.-à-d. page cache). Dans cette section, nous présentons rapidement ces différentes approches et nous expliquons en quoi notre solution est différente.
Partage de pages (déduplication). Le partage de pages transparent [26] est une technique largement utilisée pour réduire l'empreinte mémoire des MVs. Pour cela, l'hyperviseur scanne périodiquement la mémoire de chaque invité, et lorsque des pages identiques sont trouvées, elles sont partagées de sorte que les MVs accèdent aux mêmes pages physiques. L'inconvénient majeur est que scanner la mémoire utilise du temps processeur et de la mémoire. Si plusieurs solutions de partage de pages existent [5, 22], elles peuvent *compresser* la mémoire allouée à une MV, mais que faire si la MV a besoin de plus de mémoire ? Ces approches sont complémentaires à notre mécanisme.
Équilibrage de la mémoire (*Memory balancing*). Le partage de pages est souvent utilisé avec des techniques d'équilibrage de la mémoire entre les MVs. Le *memory ballooning* [26, 6, 24] est une solution d'équilibrage de la mémoire qui permet à un hyperviseur de demander à un invité de « gonfler » un ballon, ce qui immobilise des pages mémoire dans le système invité. Pour augmenter la mémoire d'une autre MV, l'hyperviseur demande à ce que le ballon de la MV se « dégonfle ». Cette solution a plusieurs inconvénients : (*i*) gonfler/dégonfler les ballons peut entraîner des temps de latence, (*ii*) cette solution est spécifique à un hyperviseur donné, et (*iii*) elle ne fonctionne que pour des MVs qui sont hébergées par le même hôte.
Caches coopératifs. Les caches coopératifs [12] utilisent la mémoire des nœuds participants pour étendre le cache *local* d'autres nœuds. De nombreux travaux ont utilisé des caches coopératifs pour améliorer les performances de systèmes de fichiers répartis [17, 12, 4, 7]. Cependant, ces solutions forcent l'utilisation soit d'un système de fichiers spécifique, soit d'un périphérique bloc virtuel [25, 13].
D'autres travaux ont proposé des caches coopératifs sans forcer l'utilisation d'un système de fichiers spécifique. XHive [18] est d'une certaine manière un cache coopératif paravirtualisé. Cependant, si leur approche fournit de bonnes performances, elle nécessite des modifications complexes à la fois dans l'hyperviseur et dans l'OS invité. De plus, comme le *memory ballooning*, XHive ne peut pas être utilisé pour mutualiser la mémoire de MVs hébergées sur des hôtes différents.
Magenheimer *et al.* [21, 20] ont proposé RAMster, un mécanisme de cache distant utilisé pour envoyer les pages évincées du cache à une machine distante, et pour les récupérer en cas de défaut dans le cache local. Ils ont proposé une API étroitement intégrée au page cache et à la couche virtuelle des systèmes de fichiers (VFS) du noyau Linux pour capturer les défauts et les évictions du cache. Notre approche, similaire à la leur, est différente en deux points :

- Alors que RAMster tient compte à la fois des pages anonymes et des pages du page cache, nous nous concentrons uniquement sur les pages du page cache car nous pensons que la mémoire allouée à une MV doit être correctement dimensionnée.
- RAMster fonctionne avec une stratégie « au mieux », dans le sens où chaque page évincée peut



(a) **get()** récupère une page du cache distant.

(b) **put()** envoie une page dans le cache distant.

FIGURE 2 – Architecture de PUMA

ne pas être envoyée dans le cache distant. Notre évaluation, section 4, montre que c'est le cas la plupart du temps, les pages évincées ne sont pas envoyées dans le cache distant, conduisant à une faible amélioration des performances. Avec PUMA, nous nous efforçons d'envoyer systématiquement les pages au cache distant pour augmenter le taux de *succès* dans le cache distant.

RAMster étant la solution la plus proche de PUMA, nous comparons les deux approches dans notre évaluation.

3. Architecture de PUMA

Le principe de PUMA est de permettre à tout nœud d'un cluster d'utiliser la mémoire d'un autre nœud pour étendre son page cache. PUMA est conçu selon un modèle client/serveur, où un nœud *client* peut utiliser la mémoire libre disponible d'un nœud

serveur. Chaque nœud peut devenir soit un client PUMA soit un serveur PUMA. Ceci peut changer au cours du temps en fonction des besoins en mémoire. Cette section décrit l'architecture de PUMA, les stratégies de cache et donne quelques détails d'implémentation.

3.1. Conception d'un cache coopératif au niveau du page cache

Nous souhaitons que notre mécanisme de cache coopératif soit le plus *générique* possible, utilisable non seulement avec des périphériques bloc, mais aussi par des systèmes de fichiers distribués. Une première approche consiste à proposer un périphérique bloc *virtuel* au-dessus d'un périphérique bloc *cible* [25]. Cependant, bien que cette solution semble simple et élégante, elle limite la portée du cache coopératif aux périphériques bloc uniquement, ce qui empêche les systèmes de fichiers distribués d'en profiter.

Nous préférons adopter une approche plus générale, qui capture directement les accès au page cache en utilisant les opérations de l'API *cleancache*. Les figures 2a et 2b représente l'architecture générale où VM_1 et VM_2 agissent respectivement comme client et serveur.

Opération *get()*. Lors d'un défaut sur une page du page cache local (étape 1 de la figure 2a), PUMA vérifie dans ses métadonnées si cette page a déjà été envoyée au serveur (2). Ces métadonnées, maintenues par le client, enregistrent les identifiants des pages qui ont été envoyées au serveur lors d'une opération *put* et sont utilisées pour éviter d'envoyer des requêtes *get* inutiles au serveur. Lorsque l'identifiant de la page est présent dans les métadonnées, PUMA envoie une requête au serveur (3), puis le serveur renvoie la page au client (4).

Opération *put()*. Lorsqu'une page victime est choisie par le PFRA pour libérer de la mémoire (étapes 1 et 2 de la figure 2b), l'opération *put* de l'API *cleancache* est appelée (3). PUMA copie ensuite le contenu de la page dans un tampon et la page est libérée (4). Enfin, la page contenue dans le buffer est envoyée au serveur pour qu'il la stocke (5).

Le principal problème avec l'API *cleancache* est que (*i*) elle ne garantit pas qu'une opération *put* réussisse et (*ii*) elle n'est adaptée à l'algorithme de préchargement du noyau. En premier lieu, *cleancache* est défini suivant une politique « au mieux » : il n'y a pas de garantie qu'une opération *put* réussisse. Par exemple, s'il n'y a plus assez de mémoire libre pour allouer les structures de données nécessaires à l'envoi des pages, comme des tampons réseau ou des métadonnées locales, l'opération est avortée. Dans ce cas, la page est libérée et une opération *get* ultérieure devra la relire depuis le disque. Concernant le préchargement, l'opération *get* étant bloquante pour chaque page à lire, le processus appelant est bloqué

pour chaque page présente dans la fenêtre de préchargement ; ceci alors qu'elles pourraient être lues de manière asynchrone. Cela n'est pas un problème pour les implémentations « optimisées » de l'opération *get* comme zcache, qui utilise l'API *cleancache* pour compresser le cache local. Cependant, dans notre cas la latence réseau rend ce comportement inefficace. Pour contourner ce problème et augmenter les performances du cache, PUMA est basé sur les 3 principes suivants.

Pré-allocation des messages. Nous nous efforçons d'envoyer les pages dans le cache distant pour augmenter les chances d'un succès. Ainsi, pour favoriser la réussite des opérations *put*, la mémoire nécessaire pour traiter les requêtes est allouée depuis des *pools* de mémoire pré-alloués. Grâce à cela, nous n'ajoutons pas de pression au PFRA. Nous synchronisons périodiquement le *PFRA* avec PUMA pour limiter le nombre de pages en instance d'envoi au cache distant, ce qui nous assure que ce *pool* n'est jamais vide. Le surcoût de cette synchronisation est détaillé dans la section 4.2.2.

Agrégation des requêtes de la fenêtre de préchargement. Pour éviter de bloquer les processus à chaque fois qu'une page est lue, nous rassemblons toutes les pages de la fenêtre de préchargement au sein d'une seule opération *get* bloquante. Cette approche nous permet de profiter de l'algorithme de préchargement de pages déjà présent au sein du noyau.

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. Envoyer des dizaines de messages n'est pas efficace et utilise de la mémoire alors même que nous souhaitons en libérer. Pour éviter de créer trop de petits messages, nous utilisons un tampon de messages par processus pour regrouper les messages.

3.2. Stratégies de cache

Dans cette section, nous présentons les stratégies de cache que nous avons implémentées pour PUMA.

3.2.1. Exclusive

Avec la stratégie exclusive, lorsque le client demande une page, le serveur la retire de sa mémoire. Cette stratégie ne nécessite pas de maintenir différentes copies de la même page. De plus, puisque que le cache distant est utilisé *en supplément* du page cache local, la capacité totale de cache est la somme des caches client et serveur. Cependant, cette stratégie peut nécessiter que le client envoie à plusieurs reprises une même page au serveur, en particulier lorsqu'il y a beaucoup de lectures.

3.2.2. Inclusive

La stratégie de cache inclusive a pour objectif de réduire le charge du client et l'utilisation du réseau pour des *workloads* intensifs en lectures. Afin de limiter le nombre de fois qu'une page est envoyée au serveur, ce dernier conserve les pages en mémoire même lorsqu'elles sont récupérées par un client. Ainsi, les pages « chaudes » restent dans la mémoire du serveur, ce qui permet d'éviter au client d'avoir à renvoyer cette page au serveur si elle est de nouveau choisie comme victime.

Cependant, comme les clients ne travaillent que sur des pages propres [1], nous n'avons aucun moyen de savoir si une page donnée a été modifiée depuis la dernière fois que PUMA l'a récupérée. Une solution est d'ajouter un drapeau « *dirtied* » qui est activé dès qu'une page est modifiée, puis de vérifier son état dès que la page est évincée du page cache. Avec la stratégie inclusive, une page peut être à la fois dans le cache local et dans le cache distant, ce qui peut conduire à des incohérences comme cela est détaillé dans la section suivante.

3.3. Détails d'implémentation

Nous avons implémenté PUMA dans le noyau Linux 3.7. L'implémentation passe la suite de tests Linux Test Project [2]. L'essentiel de notre implémentation est contenue dans ~7000 lignes de code au sein d'un module noyau. Quelques modifications au cœur du noyau ont été nécessaires : ~100 lignes pour le sous-système de gestion de la mémoire et ~50 lignes pour le VFS.

Gestion des metadonnées. Pour chaque page envoyée au serveur, les clients conservent des métadonnées dans un arbre rouge/noir. Cela permet de gérer les problèmes de cohérence et de détecter localement si une page est présente sur le serveur.

Au total, pour chaque page (4 Ko) stockée dans le cache distant, le client doit conserver seulement 60 octets de métadonnées, ce qui correspond à 16 Mo de mémoire côté client pour gérer 1 Go de cache distant. De plus, le client a toujours la possibilité de réclamer la mémoire utilisée pour les métadonnées,

1. Les pages « sales » sont écrites sur le périphérique de stockage avant que l'opération *put* ne soit appelée.

et d'invalider les pages correspondantes du cache distant.
Implémentation côté serveur. Le serveur offre une quantité de mémoire prédéfinie comme cache distant et traite les requêtes du client. Les pages côté serveur sont stockées en mémoire dans un arbre rouge/ noir et liées entre elles dans une liste *LRU*. Avec la stratégie de cache exclusive, le serveur supprime les pages demandées par le client de sa mémoire, sinon il déplace les pages accédées par les requêtes en tête de la liste LRU. Lorsque le serveur n'a plus de mémoire disponible, il en récupère en supprimant les dernières pages de la liste LRU, puis envoie un message au client afin qu'il mette à jour ses métadonnées.
Cohérence de la stratégie de cache inclusive. À cause de la mise en tampon, une page sale peut être mise dans le tampon de messages alors qu'un processus est en train de la récupérer. Ceci peut conduire à un scénario où la requête pour la page atteint le serveur (qui stocke potentiellement une ancienne version de la page) avant la page elle même. Nous résolvons ce problème de concurrence en ajoutant un bit de synchronisation aux métadonnées du client.

4. Évaluation

Dans cette section, nous présentons notre plateforme expérimentale ; puis nous détaillons et analysons les résultats obtenus avec différents benchmarks et applications dans diverses configurations.

4.1. Protocole expérimental

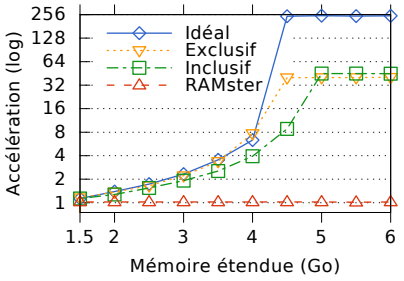
Nos expérimentations permettent de comparer les performances de PUMA à un noyau Linux et à RAMster. Nous les avons réalisées avec des MVs configurées avec 2 vCPU et un disque dur virtuel dédié pour stocker les données manipulées par l'ensemble des benchmarks. Les autres paramètres ont été configurés en suivant les recommandations de [15] : tous les disques virtuels sont paramétrés pour contourner le *page cache* du système hôte, l'ordonnanceur d'E/S *deadline* est utilisé, ainsi que le système de fichiers *ext4*, à la fois dans l'hôte et dans les VMs. Nous avons utilisé l'hyperviseur KVM [19] pour nos expérimentations, il se base sur le composant *virtio* [23] pour paravirtualiser les E/S.
Dans un premier temps, nous exécutons tous les benchmarks sur une seule MV avec un noyau Linux 3.7.0 sans cache additionnel et avec 1 Go de mémoire pour calculer notre point de référence. Ensuite, nous lançons les mêmes benchmarks avec 2 MVs en utilisant PUMA puis RAMster. La première MV a 1 Go de mémoire et contient ~100 Mo pour l'OS et la partie cliente du cache. La seconde MV fournit de 512 Mo à 5 Go de mémoire au cache distant, et agit comme serveur de cache. Nous calculons l'accélération obtenue pour chaque benchmark s'exécutant avec un cache de taille $(client + serveur)$ relativement au point de référence. Nous évaluons également un noyau Linux sans cache distant additionnel avec 1.5 Go à 6 Go de mémoire, que nous appelons *Idéal* par la suite.
Nous avons utilisé la version publique de MOSBENCH [9], une plateforme expérimentale conçue pour mesurer le passage à l'échelle des systèmes d'exploitation. Nous l'avons modifiée pour déployer les benchmarks sur des MVs en faisant varier la quantité de mémoire de celles-ci.
Les expérimentations décrites dans les sections 4.2, 4.3 et 4.5 ont été effectuées sur des MVs hébergées sur une seule machine physique équipée d'un processeur Intel Core i7-2600 (4 cœurs) et de 8 Go de mémoire. Dans ces expérimentations, nous profitons de la faible latence et de la forte bande passante du réseau paravirtualisé. Les disques virtuels sont stockés sur le disque local de la machine, qui est un disque dur de 1 To avec une vitesse de rotation de 7200 rpm.

4.2. Micro-benchmark en lecture seule

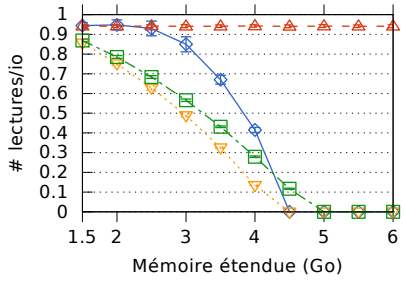
Dans cette section, nous évaluons un benchmark de lectures aléatoires en utilisant la version 1.4.9.1 de Filebench [1] ; puis nous analysons le comportement des stratégies de cache de PUMA.

4.2.1. Micro-benchmark de lectures aléatoires

Ce benchmark effectue des lectures de 4 Ko à des positions aléatoires dans un fichier de 4 Go. Nous mesurons ensuite le nombre de lectures par seconde (IO/s).
La figure 3a montre l'accélération obtenue avec PUMA et RAMster comparée au point de référence. On remarque que la totalité du fichier (4 Go) tient en mémoire à partir de 4,5 Go (à cause de la mémoire utilisée par l'OS). PUMA permet un gain de performances de 15% à 1,5 Go, et nous obtenons une accélération de 40 lorsque l'ensemble des données tiennent dans le cache (c.-à-d., local+distant). On observe que le cache exclusif est plus efficace que le cache inclusif, ce qui est analysé dans la section 4.2.2.

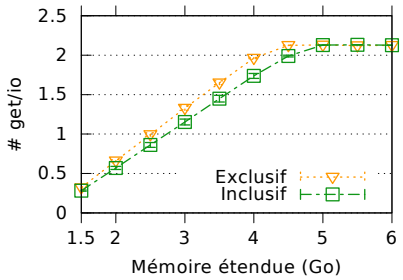


(a) Accélération

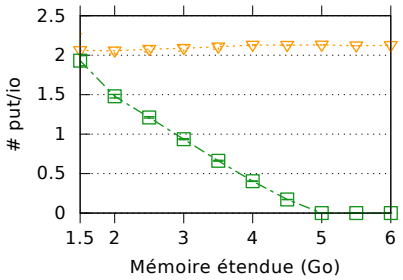


(b) Taux d'accès au disque (# d'accès par requête d'E/S)

FIGURE 3 – Performances obtenues en lectures aléatoires.



(a) Nombre de *get* (lectures aléatoires)



(b) Nombre de *put* (lectures aléatoires)

FIGURE 4 – Nombre de *get*/*put* pour le benchmarks de lectures aléatoires.

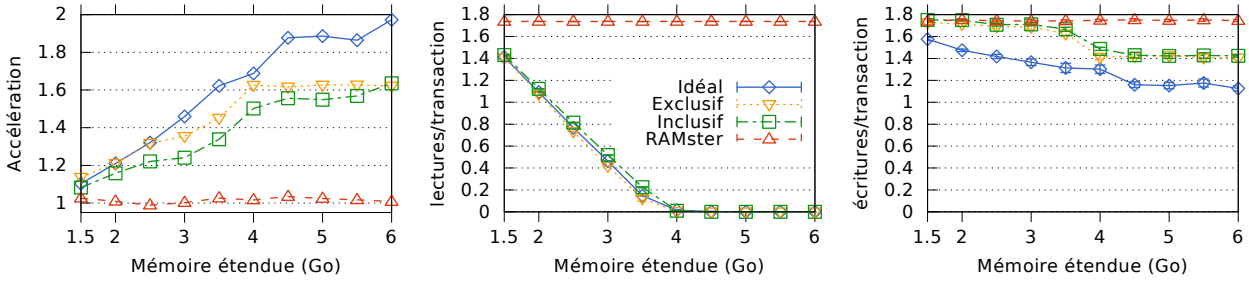
En revanche, RAMster n'améliore pas du tout les performances. Tout au long de nos expérimentations, nous avons observé que la plupart des pages sont évincées avant que RAMster ne les envoie dans le cache distant, à cause du PFRA qui réclame des pages à RAMster ou des échecs d'allocation de mémoire. Ainsi RAMster n'assure pas que les pages soient effectivement stockées dans le cache distant lorsqu'il y a une forte pression mémoire. PUMA repose sur des *pools* de mémoire pré-allouée pour augmenter le taux de réussite des opérations *put*.

Nous nous attendions à ce que la configuration idéale obtienne de meilleures performances que PUMA puisqu'un accès à la mémoire locale est plus rapide qu'un accès à une mémoire distante, ce qui n'est pas le cas lorsque la taille du cache est plus petite que la taille des données manipulées. La figure 3b montre le nombre de lectures envoyées au périphérique bloc (c.-à-d. le taux de *miss*) et permet d'expliquer ce comportement. On remarque que PUMA réduit plus le nombre de lectures que la configuration idéale. Cependant, les performances restent similaires puisqu'un succès dans le cache de la configuration idéale (mémoire locale) est beaucoup plus rapide qu'un succès dans le cache distant.

Ce résultat s'explique par la *double-LRU* du *PFRA* du noyau Linux qui n'est pas adaptée aux accès entièrement aléatoires. Le noyau Linux maintient une liste de pages actives qui ne peuvent pas être évincées et une liste inactive où les pages victimes sont choisies. Avec des lectures aléatoires, des pages peuvent être accédées plusieurs fois et incluses dans la liste active, même si elles ne sont pas utiles dans le futur. Ceci implique une augmentation de la taille de la liste active qui dégrade les performances globales de la stratégie de cache. Ce comportement affecte aussi PUMA, mais dans une moindre mesure : seule la mémoire locale est affectée (c.-à-d. moins de ~900 Mo), alors que le reste du cache est géré par une liste LRU classique côté serveur.

4.2.2. Analyse des stratégies de cache

Pour montrer l'intérêt de chaque stratégie de cache, nous avons mesuré le nombre de moyen de pages envoyées et récupérées depuis notre cache distant. La figure 4 montre le nombre de pages envoyées/ reçues par E/S avec le benchmark de lectures aléatoires. On remarque que le nombre moyen de pages récupérées (figure 4a) est inversement proportionnel au taux d'accès au disque (figure 3b). L'écart entre



(a) Accélération

(b) Nombre moyen de lectures par transaction

(c) Nombre moyen d'écritures par transaction

FIGURE 5 – Performances obtenues avec Postmark.

les deux stratégies s'explique par la taille du cache : le cache exclusif est plus grand (cache local+cache distant) que le cache inclusif, il a donc un taux de *hit* plus élevé.

La figure 4b montre le nombre de pages envoyées au cache distant par E/S avec le benchmark de lectures aléatoires. Avec une stratégie de cache inclusive, on observe que le nombre de pages envoyées au cache distant décroît en même temps que la taille du cache augmente, alors qu'il reste constant pour la stratégie exclusive. Ceci est dû à la nature de ce benchmark, en lecture seule, et qui permet à la stratégie inclusive de ne pas renvoyer des pages au serveur s'il les possède déjà. Cependant, la figure 3a montre que la stratégie exclusive est plus efficace que la stratégie inclusive jusqu'à ce que le l'ensemble des données tiennent dans le cache. Ceci illustre le surcoût de l'opération *put* de PUMA.

4.3. Applications réelles

4.3.1. Postmark: expérimentations en présence d'écritures

Notre approche fonctionne uniquement sur les pages *propres*, les pages *sales* sont donc d'abord écrites sur le disque avant d'être gérées par PUMA. Nous avons choisi d'utiliser Postmark 1.51 [16] pour évaluer les performances de PUMA en présence d'écritures.

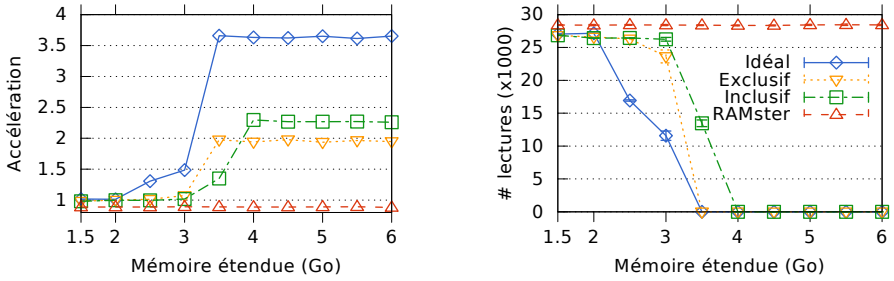
Postmark est un benchmark qui mesure la performance d'un système de fichiers avec une charge composée de beaucoup de petits fichiers, ce qui est typique des applications de type e-mail. Postmark génère des opérations à la fois sur les données et les métadonnées des fichiers, et est composé de multiples petites écritures. Il définit une *transaction* comme étant une lecture ou un ajout à un fichier existant, suivi de la création ou la suppression d'un fichier.

Nous avons utilisé une configuration de Postmark défavorable à PUMA, dans laquelle plus de 80% des E/S sont des écritures. Nous avons configuré Postmark pour générer 20 000 transactions sur 25 000 fichiers. Chaque fichier a une taille comprise entre 512 octets et 64 Ko, ce qui nous donne un total d'environ 3,5 Go de données. Postmark est configuré pour générer le même ratio de lectures/ajouts et le même ratio de créations/suppressions.

La figure 5a montre l'accélération obtenue avec Postmark ; on remarque qu'une petite quantité de cache est suffisante pour obtenir un gain d'environ 10%. Ce gain augmente linéairement jusqu'à ce que l'ensemble des données tiennent dans le cache : nous obtenons plus de 50% d'amélioration de performances avec une stratégie inclusive, et plus de 60% avec une stratégie exclusive. Les résultats de la configuration idéale ne sont pas aussi efficace que ce à quoi nous nous attendions. Ainsi, nous avons mesuré le nombre moyen d'E/S par transaction pour comprendre comment le noyau Linux gère les E/S en présence d'écritures lorsqu'il a une grande quantité de cache local.

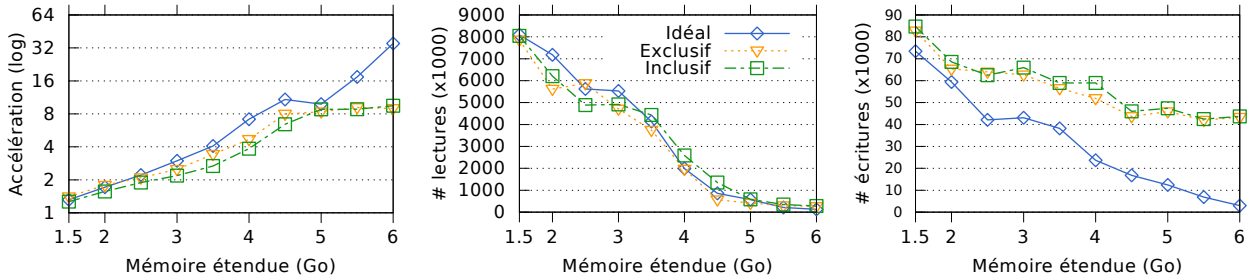
La figure 5b montre le nombre de lectures par transaction, qui décroît linéairement jusqu'à atteindre 0 à la fois pour la configuration idéale et pour PUMA. En observant le nombre d'écritures par transaction (figure 5c), on remarque que la configuration idéale peut différer plus d'écritures puisqu'elle a plus de mémoire locale. Ainsi, les écritures générées par ce workload (environ 80% [2]) permettent à la configuration idéale de réduire la pression sur le périphérique de stockage en différant les écritures. Avec PUMA, nous pouvons également réduire la pression sur le périphérique de stockage parce que le PFRA réclame

2. Les métriques utilisées dans les figures 5b et 5c peuvent faire penser qu'il y a presque autant de lectures que d'écritures, mais si on tient compte du nombre de secteurs, il y a bien 80% d'écritures et 20% de lectures.



(a) Accélération (b) Nombre total de lectures

FIGURE 6 – Performances obtenues avec BLAST.



(a) Accélération (b) Nombre total de lectures (c) Nombre total d'écritures

FIGURE 7 – Performances obtenues avec TPC-H.

en priorité les pages propres, ce qui laisse l'essentiel de la mémoire locale pour différer les écritures.

4.3.2. BLAST

BLAST [3] est utilisé par les chercheurs en bio-informatique pour trouver les regions similaires entre une séquence donnée et une base de données de séquences de nucléotides ou d'acides aminés. La majorité des accès disque de BLAST se font de façon séquentielle dans la base de données, alors qu'un index est manipulé de façon plus aléatoire. Pour nos expérimentations, nous avons utilisé la base de données *patnt* qui a une taille d'environ 3 Go. Les requêtes correspondent à l'extraction de 5 séquences de 600 caractères depuis cette base de données.

La figure 6a montre l'accélération obtenue avec BLAST. Alors que PUMA ralentit légèrement BLAST (2%) lorsque le cache distant ne fourni que 512 Mo de cache, il n'y a aucune accélération ou dégradation des performances jusqu'à ce que l'ensemble de la base de données ne tienne dans le cache. Lorsque le cache est suffisamment grand, PUMA permet d'aller 2,3 fois plus vite avec un cache inclusif. Ces résultats montrent que même en augmentant la mémoire locale disponible, la configuration idéale ne peut augmenter les performances puisque la base de données est accédée séquentiellement avec une faible localité temporelle. La stratégie exclusive n'est pas adaptée à ce type d'usage puisqu'il n'y a pas d'amélioration des performances tant que l'ensemble des données ne tient pas en dans le cache. Dans ce cas, un cache inclusif est plus efficace.

4.3.3. Expérimentations avec une base de données (TPC-H)

Dans cette section, nous étudions les performances de PUMA avec une charge de type de base de données. Nous avons utilisé la version 9.3.1 de PostgreSQL et nous avons utilisé la base de données et les requêtes définies par le benchmark TPC-H [10]. TPC-H définit un ensemble de 22 transactions complexes, dont la majorité sont en lecture seule.

Nous avons configuré TPC-H avec un facteur d'échelle à 3, ce qui génère une base de données d'environ 3 Go, et nous mesurons le temps d'exécution total des 22 requêtes. La figure 7a présente l'accélération obtenue avec TPC-H. On observe d'abord que l'ajout de seulement 512 Mo de cache permet d'améliorer les performances de 25% avec un cache inclusif et de 40% avec un cache exclusif. Ensuite, on remarque

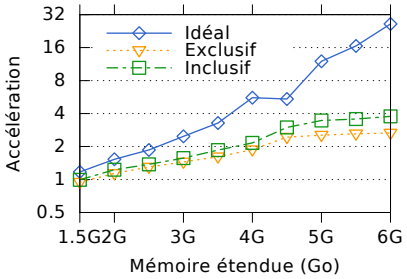


FIGURE 8 – Accélération de TPC-H exécuté avec un cache distant sur InfiniBand.

	RR (IO/s)	Pm (T/s)	BLAST (s)	TPC-H (s)
Référence	166	64.7	42.3	7299.0
Idéal	40400	219.2	11.5	233.5
PUMA (excl.)	5885	104.6	21.2	910.0
PUMA (incl.)	**6310**	100.5	**17**	**861.8**
dm-cache	3250	**695.2**	21.3	1148.1

RR lectures aléatoires(I/O/sec) **Bold** Meilleur
Pm Postmark (transactions/sec)

TABLE 1 – Comparaison des performances entre un cache SSD et PUMA, avec 1 Go de mémoire locale et 5 Go de cache additionnel.

que l'accélération augmente linéairement avec l'augmentation de la quantité de cache, ce qui permet à TPC-H d'aller 2,5 fois plus vite avec 2,5 Go de cache distant. Enfin, les performances augmentent nettement lorsque l'essentiel des données manipulées tiennent entièrement dans le cache : le benchmark est 9 fois plus rapide avec PUMA et 6 Go de cache additionnel.
Avec un cache exclusif, les performances de PUMA sont proches des performances idéales jusqu'à 5 Go de mémoire. La figure 7b montre qu'au delà de 4 Go l'essentiel des données manipulées tiennent dans le cache, ce qui permet une nette amélioration des performances. En revanche, la figure 7c montre que même s'il y a beaucoup moins d'écritures que de lectures, au delà de 5 Go de mémoire, les écritures deviennent le goulet d'étranglement pour PUMA puisqu'il ne dispose que de 1 Go de mémoire locale pour différer l'écriture des données sur le disque.

4.4. Configuration répartie avec *InfiniBand*

Les précédentes évaluations se sont concentrées sur les performances de PUMA sur un seul nœud, et utilisaient le réseau local paravirtualisé comme support de communication haute performance. Cependant, puisque PUMA a l'avantage de pouvoir utiliser un réseau distant pour construire un cache coopératif entre des MVs hébergées sur plusieurs nœuds différents, nous avons évalué les performances de PUMA avec un réseau *InfiniBand*.
Pour ces expériences, nous avons utilisé la grappe de serveurs *graphene* de la plateforme Grid'5000 [8], où chaque nœud est composé d'un processeur Intel Xeon X3440 à 4 cœurs, de 16 Go de mémoire et de cartes InfiniBand 20Gb/s.
Nous avons exécuté le benchmark TPC-H avec la même configuration que celle décrite dans la section 4.3.3. Ensuite, nous avons déployé 2 MVs sur 2 nœuds différents. Les MVs sont interconnectées en utilisant IP sur InfiniBand (*IPoIB*). Nous n'avons pas pu accéder à InfiniBand en utilisant des technologies comme *SR-IOV* ou *PCI-passthrough* parce que ces nœuds n'étaient pas assez récents. Nous avons donc dû utiliser des tables de routage statiques dans les hôtes pour forcer les MVs à dialoguer via IPoIB, ce qui est un désavantage pour PUMA puisque cette configuration est plus lente que si nous avions eu un accès direct à InfiniBand.
La figure 8 montre l'accélération obtenue avec InfiniBand. Avec 512 Mo de cache distant, nous avons observé un ralentissement de l'ordre de 5% avec un cache exclusif et de seulement 1% avec un cache inclusif. Cependant, à partir de 1 Go de cache distant, PUMA peut améliorer les performances, de 20% (avec 1 Go de cache distant) et jusqu'à 2,5 fois plus rapide avec un cache exclusif et 3,7 fois plus rapide avec un cache inclusif.
Avec InfiniBand, la stratégie de cache inclusive qui génére moins de messages est toujours meilleure que la stratégie exclusive, pour deux principales raisons : (*i*) la latence entre les MVs est plus élevée qu'avec un seul hôte ; et (*ii*) nous n'avons pas pu obtenir plus de 10Gb/s de bande passante avec IPoIB.
Sur InfiniBand, PUMA est plus lent que sur l'expérimentation précédente avec des MVs hébergées sur le même hôte. Cependant, nous pensons que nous aurions obtenu de meilleures performances si nous avions pu utiliser les technologies *SR-IOV* ou *PCI-passthrough*.

4.5. Comparaison avec un cache sur un SSD

Les disques SSD 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. Puisque que les SSDs restent plus chers que des HDDs, ils sont souvent utilisés comme support de cache pour les HDDs.

Pour nos expérimentations, nous avons utilisé un SSD Samsung 840 Pro 128GB avec le module *dm-cache* du noyau Linux 3.11.6. dm-cache 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. Nous avons configuré dm-cache au sein d'une MV pour qu'il utilise un périphérique de cache de 5 Go, c.-à-d. que l'ensemble des données de nos benchmarks tiennent dans le cache. Nous avons aussi lancé ces benchmarks avec PUMA et 5 Go de cache distant.

Les résultats de ces expérimentations sont présentés dans la table 1. Ces résultats montrent qu'avec les benchmarks aléatoires, PUMA est plus efficace qu'un cache sur SSD, alors que nous nous attendions à ce que le cache sur SSD soit à peu près aussi efficace. Ceci s'explique par le surcoût lié à la virtualisation des E/S [14] : nous avons mesuré plus de 200µs de temps d'accès au SSD depuis les MVs.

Avec BLAST, le SSD n'est pas impacté par le temps d'accès puisque les accès sont principalement séquentiels. Ainsi, le cache SSD est aussi rapide que PUMA, sauf avec notre stratégie inclusive qui excelle avec les charges séquentielles et en lecture seule.

Avec TPC-H, PUMA reste meilleur qu'un cache SSD, mais 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.

Enfin, avec Postmark le cache SSD est bien plus rapide que PUMA puisque nous ne gérons pas les écritures, qui doivent donc être écrites sur le disque dur magnétique. Le cache SSD est 3 fois plus rapide que la configuration idéale, ce qui nous montre que le coût des écritures sur un disque magnétique est énorme : même si la configuration idéale 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.

Le cache SSD est très performant en présence d'écritures. Nous avons remarqué qu'à cause du surcoût de la virtualisation, les performances du cache SSD au sein d'une MV ne sont pas très élevées. Nous pensons que ce problème s'améliorera dans le futur, les technologies de virtualisation étant de plus en plus efficaces. Les performances de PUMA pourraient aussi bénéficier d'une amélioration des technologies de virtualisation réseau.

5. Conclusions

Beaucoup d'applications ont leurs performances qui sont directement liées à l'efficacité des E/S. Dans les architectures de type cloud, l'utilisation intensive de la virtualisation conduit à une fragmentation de la mémoire, et les systèmes d'exploitation qui s'exécutent dans les MVs ont seulement une petite quantité de mémoire disponible pour les mécanismes de cache.

Pour s'attaquer à ce problème, nous avons conçu PUMA, un mécanisme de cache distant efficace qui permet de mutualiser la mémoire inutilisée des MVs. Notre approche au niveau noyau est indépendante du périphérique bloc, du système de fichiers et de l'hyperviseur. De plus, PUMA peut fonctionner à la fois localement et à distance. Nous avons montré, grâce à une évaluation approfondie, que PUMA permet aux applications d'utiliser de la mémoire libre d'autres MVs pour améliorer leurs performances. Il est jusqu'à 40 fois plus rapide avec un benchmark de lectures aléatoires et 9 fois plus rapide avec un benchmark de base de données.

Remerciements

Certaines expérimentations présentées dans cet article ont été effectuées sur la plateforme Grid'5000, issue des actions de développement de l'initiative ALADDIN pour l'Inria, avec le support du CNRS, de RENATER et de plusieurs universités et d'autres contributeurs (voir `https://www.grid5000.fr`).

Ces travaux ont été financés par l'Agence nationale française pour la recherche (ANR), dans le cadre du projet MyCloud (ANR-10-SEGI-0009, `http://mycloud.inrialpes.fr`) ainsi que par le projet Nu@age (`http://www.nuage-france.fr`).

Bibliographie

1. Filebench. – `http://filebench.sourceforge.net`.
2. Linux test project. – `http://ltp.sourceforge.net`.
3. Altschul (S. F.), Gish (W.), Miller (W.), Myers (E. W.) et Lipman (D. J.). – Basic local alignment search tool. *Journal of molecular biology*, vol. 215, n3, 1990, pp. 403–410.
4. Annapureddy (S.), Freedman (M. J.) et Mazières (D.). – Shark: scaling file servers via cooperative caching. *In : Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*.
5. Arcangeli (A.), Eidus (I.) et Wright (C.). – Increasing memory density by using KSM. *In : Proceedings of the linux symposium*, pp. 19–28.
6. Barham (P.), Dragovic (B.), Fraser (K.), Hand (S.), Harris (T.), Ho (A.), Neugebauer (R.), Pratt (I.) et Warfield (A.). – Xen and the art of virtualization. *In : Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pp. 164–177.
7. Batsakis (A.) et Burns (R.). – NFS-CD: write-enabled cooperative caching in NFS. *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, n3, mars 2008, pp. 323 –333.
8. Bolze (R.), Cappello (F.), Caron (E.), Daydé (M.), Desprez (F.), Jeannot (E.), Jégou (Y.), Lanteri (S.), Leduc (J.) et Melab (N.). – Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, vol. 20, n4, 2006.
9. Boyd-Wickizer (S.), Clements (A. T.), Mao (Y.), Pesterev (A.), Kaashoek (M. F.), Morris (R.) et Zeldovich (N.). – An analysis of linux scalability to many cores. *In : Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pp. 1–16.
10. Council (T. P. P.). – Tpc-h. – `http://www.tpc.org/tpch/spec/tpch2.16.0.pdf`, 2013.
11. Dahlin (M. D.), Wang (R. Y.), Anderson (T. E.), Neefe (J. M.), Patterson (D. A.) et Roselli (D. S.). – Serverless network file systems. *ACM Transactions on Computer Systems*, vol. 14, n1, 1996, pp. 41–79.
12. Dahlin (M. D.), Wang (R. Y.), Anderson (T. E.) et Patterson (D. A.). – Cooperative caching: using remote client memory to improve file system performance. *In : Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*.
13. Han (H.), Lee (Y. C.), Shin (W.), Jung (H.), Yeom (H. Y.) et Zomaya (A. Y.). – Cashing in on the cache in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, n8, 2012, pp. 1387–1399.
14. Har'El (N.), Gordon (A.), Landau (A.), Ben-Yehuda (M.), Traeger (A.) et Ladelsky (R.). – Efficient and scalable paravirtual I/O system. *In : Proceedings of the 2013 USENIX Annual Technical Conference*.
15. IBM. – Best practices for kvm. – White Paper, Nov. 2010.
16. Katcher (J.). – *Postmark: A new file system benchmark*. – Rapport technique, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.
17. Kazar (M. L.). – Synchronization and caching issues in the andrew file system. *In : Proceedings of the USENIX Winter Conference*.
18. Kim (H.), Jo (H.) et Lee (J.). – Xhive: Efficient cooperative caching for virtual machines. *Computers, IEEE Transactions on*, vol. 60, n1, 2011, pp. 106–119.
19. Kivity (A.), Kamay (Y.), Laor (D.), Lublin (U.) et Liguori (A.). – kvm: the linux virtual machine monitor. *In : Proceedings of the Linux Symposium*, pp. 225–230.
20. Magenheimer (D.). – Transcendent memory in a nutshell. *LWN.net*, August 2011.
21. Magenheimer (D.), Mason (C.), McCracken (D.) et Hackel (K.). – Transcendent memory and linux. *In : Proceedings of the 11th Linux Symposium*, pp. 191–200.
22. Miłós (G.), Murray (D. G.), Hand (S.) et Fetterman (M. A.). – Satori: Enlightened page sharing. *In : Proceedings of the 2009 USENIX Annual technical conference*.
23. Russell (R.). – Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, vol. 42, n5, juillet 2008, pp. 95–103.
24. Schopp (J. H.), Fraser (K.) et Silbermann (M. J.). – Resizing memory with balloons and hotplug. *In : Proceedings of the Linux Symposium*, pp. 313–319.
25. Van Hensbergen (E.) et Zhao (M.). – *Dynamic policy disk caching for storage networking*. – Rapport technique nRC24123, IBM, 2006.
26. Waldspurger (C. A.). – Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, vol. 36, nSI, décembre 2002, pp. 181–194.