Compas'2015 : Parallélisme/ Architecture / Système / Répartition
Lille, France, du 30 juin au 3 juillet 2015

Gestion dynamique du cache entre machines virtuelles

Maxime Lorrillere, Joel Poudroux, Julien Sopena et Sébastien Monnet

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

PUMA est un mécanisme de cache réparti intégré au noyau Linux qui permet de mutualiser la mémoire inutilisée de machines virtuelles (MVs) pour améliorer les performances des applications intensives en entrées/sorties (E/S). Dans le design initial que nous avons proposé, la configuration de PUMA était essentiellement *statique*, la quantité de mémoire prêtée par un nœud était difficilement modifiable en cas de variation de charge sur l'un des nœud. Dans cet article, nous proposons plusieurs mécanismes permettant à PUMA d'ajuster dynamiquement la quantité de mémoire prêtée aux MVs en fonction des besoins. Nos évaluations, reposant sur des lectures aléatoires et des allocations de mémoire, montrent que PUMA est capable d'ajuster son activité et la quantité de mémoire prêtée à d'autres nœuds pour éviter de dégrader les performances. Comparé à une approche à de type *ballooning* automatique, PUMA est capable de récupérer 10 fois plus rapidement la mémoire allouée.

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

1 Introduction

Le *cloud computing* utilise largement la virtualisation, qui est la base pour assurer flexibilité, portabilité et isolation. Cependant, la généralisation de la virtualisation tend a augmenter de façon importante la quantité de mémoire inutilisée. En effet, comme il est difficile de prévoir la quantité de mémoire nécessaire à une application, la quantité allouée aux MVs est généralement surdimensionnée. De plus, la quantité de mémoire est souvent choisie parmi des configurations prédéfinies offertes par les fournisseurs de cloud [2]. Ainsi, sur une machine physique la quantité de mémoire disponible est *partitionnée* entre les MVs, conduisant à une sous-utilisation de la mémoire et un surcout important : certaines MVs peuvent manquer de mémoire alors que d'autres pourraient en utiliser moins sans dégrader les performances.
Dans ce contexte, nous avons proposé PUMA [8], un système basé sur un mécanisme de cache distant qui permet de mutualiser la mémoire inutilisée des MVs pour améliorer les performances des applications intensives en E/S. PUMA est directement intégré au page cache du noyau Linux et manipule uniquement les pages « propres », ce qui lui permet d'être efficace et transparent vis-à-vis des applications. Cependant, un défaut majeur de PUMA est que sa configuration est essentiellement *statique* ; la quantité de mémoire prêtée par un nœud PUMA est prédéfinie et difficilement modifiable, en particulier si le nœud exécute lui aussi des applications intensives en E/S. La première difficulté est ici de savoir quand un nœud devient inactif, pour que sa mémoire inutilisée (généralement du *cache local*), puisse être récupérée et prêtée au

cache réparti. De même, lorsqu'un nœud redevient actif, il doit être capable de récupérer efficacement la mémoire qu'il prête pour ne pas être pénalisé. La seconde difficulté est d'être en mesure de gérer des charges concurrentes sur le nœud *client* et sur le nœud *serveur* : comment détecter cette activité concurrente sans dégrader les performances ?
Dans cet article, nous proposons une série de modifications à PUMA permettant d'ajuster dynamiquement la quantité de mémoire offerte au cache, sans pénaliser l'activité *locale*. Ces modifications reposent sur des mécanismes du noyau Linux existants, ce qui permet de limiter l'impact de celles-ci tant à PUMA qu'au noyau Linux. Nous montrons à travers plusieurs expérimentations que nos améliorations permettent à PUMA d'ajuster automatiquement la quantité de mémoire prêtée à d'autres MVs pour améliorer leurs performances. Nos évaluations, qui simulent des applications intensives en E/S et en mémoire, montrent que :

- PUMA est capable de récupérer la mémoire prêtée à d'autres MVs jusqu'à 10 fois plus rapidement qu'une approche à base de *ballooning* automatique ;
- PUMA est capable d'ajuster automatiquement la quantité de mémoire prêtée en fonction des *workloads* qui s'exécutent sur le client et le serveur.

Le reste de cet article est organisé comme suit. La section 2 détaille les prérequis nécessaires à la compréhension de notre contribution, présente les approches existantes de rééquilibrage dynamique de la mémoire et montre en quoi elles ne sont pas adaptées. La section 3 détaille et évalue les modifications que nous avons apportées à PUMA pour gérer automatiquement la mémoire. Enfin, la section 4 conclut cet article.

2 État de l'art et motivations

2.1 Gestion de la mémoire dans le noyau Linux

Le page cache. Afin d'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.
En cas de pression mémoire, par exemple lorsqu'un processus tente d'allouer de la mémoire, l'algorithme de récupération de mémoire (*PFRA*, pour *Page Frame Reclaiming Algorithm*) du noyau Linux libère des pages *victimes* du page cache.

Activation/Désactivation des pages. Les pages du page cache sont stockées dans deux listes *LRU*. Lors d'un premier accès à une page, elle est placée en tête de la liste *inactive*. Lorsqu'une page est accédée une seconde fois, elle est considérée *chaude* et est promue dans la liste *active*. La figure 1 est un exemple du fonctionnement des listes LRU.
Pour rééquilibrer la taille des listes, en cas de pression mémoire si la liste active est plus grande que la liste inactive les pages les plus anciennes 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 pas plus de 50% de la taille du page cache. S'il n'y a pas de pression mémoire, 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.

Estimation de la taille du *working set*. Il se peut que des pages lues depuis le disque soient évincées trop rapidement alors qu'elles auraient pu être activées si la liste inactive était plus grande. Ce cas de figure peut se produire, par exemple, avec un accès séquentiel répétitif dont la longueur est de plus de 50% de la taille du page cache : comme la liste active peut occuper jusqu'à 50% du page cache, la liste inactive n'est pas assez grande pour détecter que toutes les

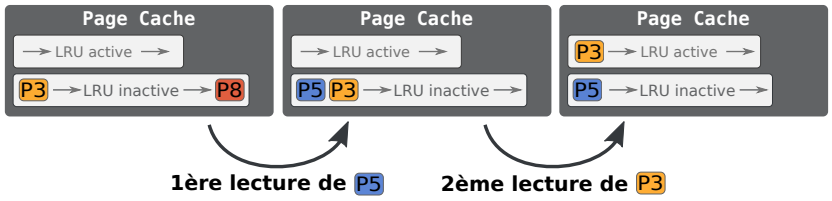


FIGURE 1 – Fonctionnement des listes *LRU* du noyau Linux. Lors de la première lecture de la page P5, la page P8, qui est en queue de liste inactive, est évincée pour faire de la place. Lors d'un accès à la page P3, celle-ci est promue dans la liste active car elle était déjà présente dans la liste inactive.

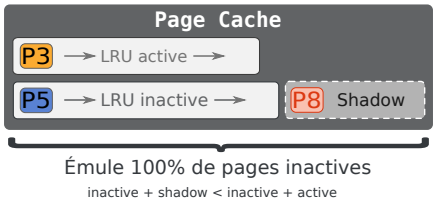


FIGURE 2 – *Shadow page cache*

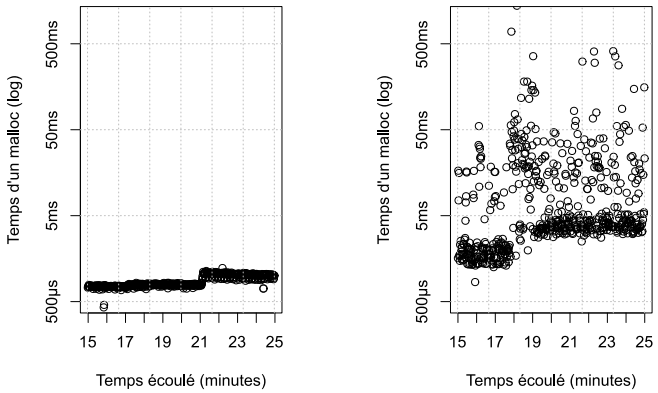
pages pourraient être chaudes.
Pour corriger ce problème, le noyau Linux dispose depuis la version 3.15 d'un mécanisme permettant de conserver l'historique des pages évincées [5, 12]. Ainsi, lorsqu'une page évincée récemment est accédée depuis le disque (*shadow hit*), le système la considère comme *chaude* et l'active immédiatement. La figure 2 illustre ce mécanisme.

2.2 Motivations

Les besoins en mémoire évoluant avec le temps, il est important d'offrir la posibilité de pouvoir modifier dynamiquement la quantité de mémoire utilisable par une MV.
Pour changer dynamiquement la quantité de mémoire allouée à une MV, Waldspruger *et al.* [11] ont proposé un mécanisme appelé *ballooning*. Leur approche consiste en l'ajout d'un pilote dans la MV chargé d'allouer de la mémoire (*gonflage* du ballon), la mémoire est ainsi rendue à l'hyperviseur qui peut la réattribuer à une autre MV, laquelle pourra *dégonfler* son ballon pour augmenter sa quantité de mémoire. Cette solution a été depuis adaptée à d'autres hyperviseurs comme Xen [1] ou KVM [7, 10].
Si le *ballooning* est généralement contrôlé manuellement, il peut être automatisé depuis l'hyperviseur en utilisant des techniques d'échantillonnage des pages utilisées [11], ou à l'aide d'approximations de la LRU avec des histogrammes [13]. Dans KVM, l'automatisation du *ballooning* [3] repose sur des informations sur la pression mémoire en provenance des MVs.
L'approche à base de ballons présente deux défauts majeurs principalement dus au *gap sémantique* [4] entre les MVs et l'hyperviseur :

— Il est difficile de tenir compte des applications intensives en E/S, qui allouent peu de mémoire *anonyme* mais nécessitent beaucoup de cache pour atteindre des performances acceptables. L'approche que nous avons proposée avec PUMA [8] permet de répondre spécifiquement à ce problème en se concentrant sur le cache.
— Une fois que la mémoire a été réallouée, il est difficile voire impossible de la récupérer sans *swapper* [9, 6], ce qui dégrade les performances globales des applications.

Pour montrer à quel point le *ballooning* peut impacter les performances, nous avons mesuré le



(a) Référence (b) Auto-ballooning

FIGURE 3 – Latence des allocations de mémoire

temps nécessaire pour effectuer des allocations de mémoire (*malloc*) sur une MV alors qu'une autre MV effectuait des E/S, en utilisant le *ballooning* automatique [3] de l'hyperviseur KVM. Ces mesures sont représentées dans la figure 3b. Pour comparer, nous reportons dans la figure 3a le temps des allocations sur une MV seule.
Alors qu'en temps normal, une allocation mémoire prend moins d'1ms, avec le *ballooning* automatique les allocations prennent en moyenne 20ms, avec un important écart type (certaines valeurs non représentées sur cette figure dépassent 1s).

3 Automatisation du partage de la mémoire dans PUMA

Dans cette section, nous étudions le comportement de PUMA en présence de *workloads* sur les nœuds client et serveur, et nous proposons et évaluons des mécanismes permettant d'améliorer la gestion dynamique de la mémoire dans PUMA. La section 3.1 présente les applications que nous avons utilisées pour analyser PUMA. Ensuite, la section 3.2 se concentre sur les variations de charge entre le client et le serveur, puis la section 3.3 s'intéresse à l'effet de *workloads* simultanés sur le client et le serveur.

3.1 Méthodologie

Pour évaluer PUMA en présence de *workloads* sur le client et le serveur, nous avons utilisé 2 MVs, la première (client) possède 512 Mo de mémoire, et la seconde (serveur) possède 700 Mo de mémoire.
Pour simuler une application intensive en E/S, nous avons utilisé le benchmark de lectures aléatoires de Filebench [1] et nous monitorons le nombre d'E/S complétées par seconde. Nous avons utilisé un fichier de 500 Mo, ainsi le client aura besoin de cache supplémentaire pour améliorer ses performances, alors que le serveur aura besoin de toute sa mémoire.
Pour simuler une application intensive en allocations de mémoire (*malloc*), nous avons développé un micro-benchmark qui consiste en l'allocation de petits fragments de mémoire à chaque seconde, ce qui permet d'émuler le besoin en mémoire d'une application. La taille de ces fragments est choisie de façon à ce que toute la mémoire de la MV soit allouée en 10min. Nous écrivons des données dans chaque fragment alloué pour s'assurer que les pages ont réellement été allouées à la MV par l'hôte.
Dans le reste de cette section, nous représentons dans les résultats de nos expériences la période d'activité de l'activité de la MV client par ⟷, et la période de l'activité de la MV serveur par

1. `http://filebench.sourceforge.net`

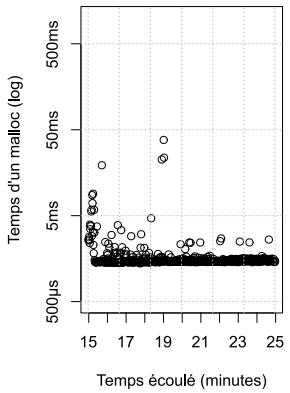


FIGURE 4 – Latence des allocations de mémoire avec PUMA

↔.

3.2 Prêter et récupérer efficacement la mémoire d'un nœud PUMA

Dans cette section, nous nous intéressons d'abord à l'efficacité de la récupération de la mémoire de PUMA lorsqu'un processus tente d'allouer de la mémoire sur le serveur. Ensuite, nous analysons le comportement de PUMA lorsqu'un client arrête de faire des E/S alors que le serveur commence une activité intensive en E/S, puis lorsque le serveur arrête son activité intensive en E/S et que le client recommence à faire des E/S.

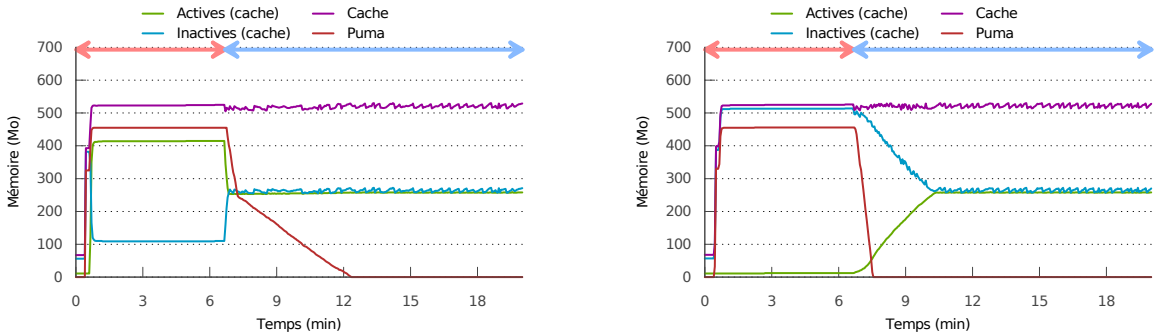
3.2.1 Récupération efficace de la mémoire pour des pages anonymes

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 nous avons observé avec l'auto-ballooning.
Pour rendre ce mécanisme à la fois automatique et efficace, nous avons choisi d'intégrer les pages *distantes* de PUMA (c.-à-d. les pages hébergées) directement dans les listes LRU du noyau Linux. Ainsi, la récupération de la mémoire occupée par les pages distantes s'effectue de la même manière que les pages locales, via le *PFRA*. Dans le cas où le *PFRA* choisi l'une de ces pages, PUMA évince la page immédiatement [2], et en informe le nœud client pour qu'il maintienne ses métadonnées.
Pour mesurer l'efficacité de cette solution, nous avons reproduit l'expérience de la figure 3, dont les résultats sont présentés dans la figure 4.
Contrairement à l'auto-ballooning, PUMA embarque la logique de récupération de la mémoire directement dans les MVs, ce qui nous permet de récupérer la mémoire utilisée pour du cache sans l'aide de l'hyperviseur. En moyenne, nous avons mesuré 1,8ms de latence pour les allocations de mémoire, avec un écart type de 2,2, ce qui est 10 fois plus rapide que l'auto-ballooning.

3.2.2 Récupération de la mémoire pour du cache

Le principe même d'un cache fait que PUMA stocke les pages des clients y compris lorsqu'ils arrêtent de faire des E/S. Cependant, lorsque le nœud PUMA serveur commence une activité intensive en E/S, il doit être capable d'utiliser efficacement sa propre mémoire, sans être pénalisé par les pages laissées par les clients.

2. 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.



(a) Mémoire du serveur PUMA en utilisant la totalité du page cache

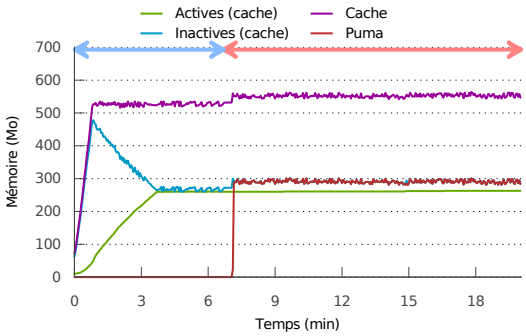
(b) Mémoire du serveur PUMA en utilisant seulement les pages inactives

FIGURE 5 – 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

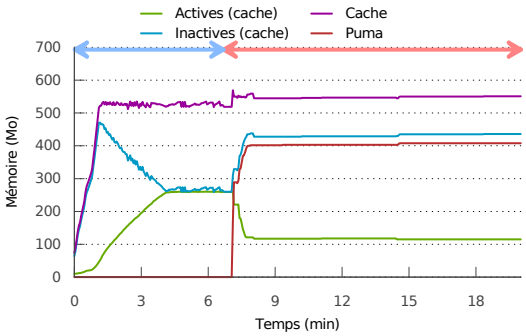
Pour évaluer le comportement de PUMA dans cette situation, nous lançons le benchmark de lectures aléatoires sur le client pendant 400s, puis nous lançons à nouveau ce même benchmark sur le serveur pendant 800s. Comme la figure 5a le montre, on remarque que le serveur récupère lentement les pages du client : lorsque le benchmark du serveur démarre, les pages *actives* sont récupérées à hauteur de 50%, puis elles sont progressivement récupérées. Le problème ici est que la *désactivation* d'une page 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 5b montre l'évolution de la mémoire du serveur 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 du serveur d'atteindre sa performance maximale plus rapidement.

3.2.3 Détection de l'inactivité d'un nœud PUMA

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. Pour évaluer ce comportement, nous avons lancé le benchmark de lectures aléatoires sur le serveur pendant 400s, puis sur le client. La figure 6a montre l'évolution de la mémoire du serveur. On remarque que le client n'est pas capable d'utiliser plus de 50% de la mémoire disponible du serveur. En effet, comme les pages distantes sont maintenant stockées exclusivement dans la liste inactive, il est impossible de récupérer plus de 50% de la mémoire puisque le reste contient les pages (anciennement) actives du serveur.
Pour permettre à un client d'utiliser plus de mémoire sans pénaliser le serveur, nous avons modifié le noyau Linux pour que des pages actives puissent être désactivées plus souvent. Pour cela, 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 6b montre l'évolution de la mémoire du serveur une fois ce mécanisme activé. On remarque que PUMA est maintenant capable d'utiliser tout le page cache, dans les limites que nous avons fixé.

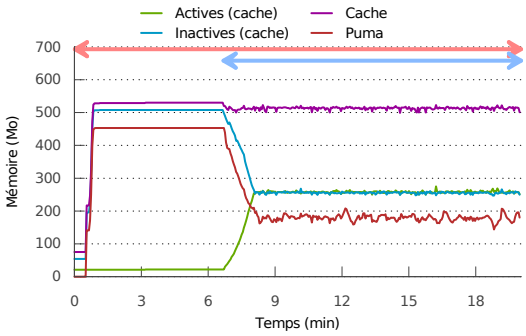


(a) Mémoire du serveur PUMA en utilisant seulement les pages inactives

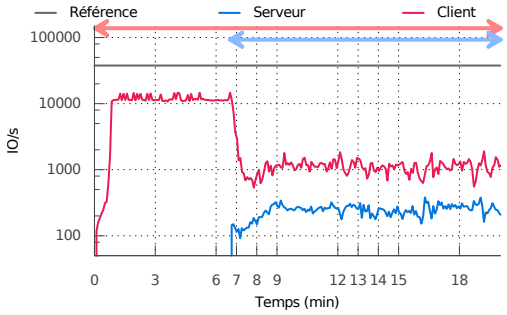


(b) Mémoire du serveur PUMA en désactivant les pages actives

FIGURE 6 – 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



(a) Mémoire du serveur PUMA



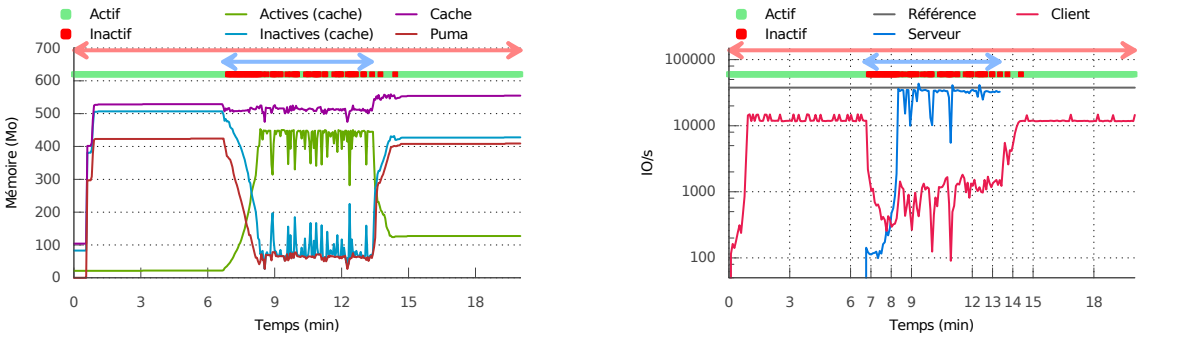
(b) Performance des benchmarks de lectures aléatoires sur les 2 nœuds PUMA

FIGURE 7 – Comportement de PUMA en présence de workloads concurrents sur les nœuds client et serveur.

3.3 Gestion de la concurrence des accès au cache

Nous avons montré comment PUMA est capable de détecter et récupérer la mémoire inutilisée au profit d'un autre nœud ou pour lui-même. Cependant, il est possible qu'une charge concurrente s'exécute à la fois sur le nœud client et sur le nœud serveur ; dans ce cas il est nécessaire de ne pas dégrader les performances des applications intensives en E/S qui s'exécutent sur le serveur.

Pour analyser le comportement de PUMA dans cette situation, nous exécutons le benchmark de lectures aléatoires en continu sur le nœud client, puis nous exécutons en parallèle le même benchmark sur le nœud serveur. Les résultats de cette expérience sont présentés dans la figure 7. La figure 7a montre l'évolution de la mémoire du serveur. On remarque que, grâce aux mécanismes introduits dans la section 3.2, PUMA récupère rapidement les pages distantes pour son propre usage. Cependant, l'activité concurrente diminue la quantité de mémoire utilisable par le serveur, ce qui ne permet pas aux données du serveur de tenir dans le cache pour atteindre les performances maximales. En effet, la figure 7b montre la performance mesurée pour ce benchmark : la performance du client chute, puisqu'il accède à moins de mémoire distante, mais celle du serveur reste extrêmement basse parce qu'il prête trop de mémoire au client.



(a) Mémoire du serveur PUMA

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

FIGURE 8 – Comportement de PUMA en présence de workloads concurrents sur les nœuds client et serveur et en désactivant PUMA lors d'un hit dans les *shadow pages* du serveur.

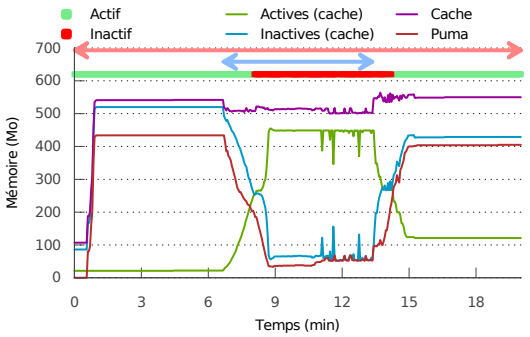
Afin de corriger ce problème, nous proposons de désactiver PUMA lorsqu'une activité intensive en E/S est détectée côté serveur, ce qui lui permettrait de récupérer sa mémoire sans être dérangé par un client qui continue de lui envoyer des pages.

3.3.1 Détection d'une activité cache via les *shadow pages*

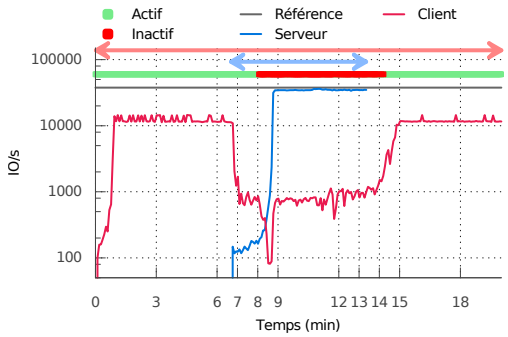
Une première approche pour détecter une activité intensive en E/S est d'utiliser l'algorithme d'estimation du *working set* présenté dans la section 2.1. En effet, lorsque cet algorithme détecte un *shadow hit*, cela veut dire qu'une page *chaude* a été évincée parce que la liste inactive est trop petite. 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 MV subit une forte charge en terme d'E/S qui nécessite plus de cache. Ainsi, nous décidons de désactiver PUMA pendant 3 secondes à chaque fois que cet algorithme détecte un *shadow hit* pour faire en sorte que le client n'envoie plus de pages au serveur. Une fois ce délai passé, s'il n'y a pas eu de *shadow hit* PUMA peut être réactivé. Cette valeur a été définie expérimentalement et permet d'ajouter de l'hystérésis dans le cycle d'activation/de désactivation de PUMA.
Pour évaluer ce mécanisme, nous reproduisons la précédente expérience dont les résultats sont présentés dans la figure 8. Sur ces figures, nous avons indiqué les périodes pendant lesquelles PUMA se désactivait. Sur la figure 8a, 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 8b on observe que la performance du benchmark sur le serveur est proche 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 la performance mesurée. Une seconde métrique pour détecter l'activité est donc nécessaire.

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

Une seconde méthode pour détecter l'activité du serveur de façon plus précise repose sur le réequilibrage des listes actives/inactives tel que décrit dans la section 2.1. Pour cela, nous considérons que le serveur possède une activité intensive en E/S lorsque 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 le serveur possède une activité suffisamment importante en E/S au point d'activer des pages *et* d'en réclamer.

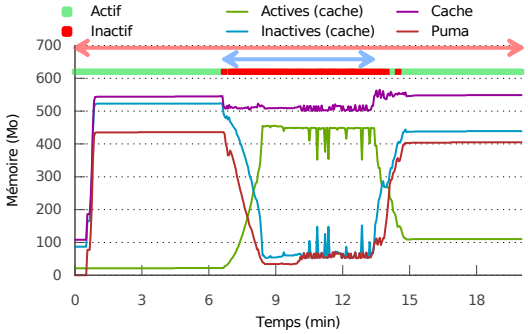


(a) Mémoire serveur PUMA

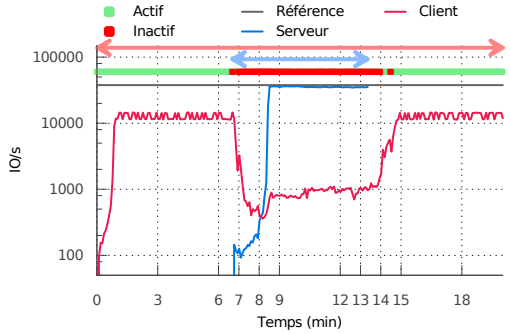


(b) Performance des benchmarks de lectures aléatoires sur les 2 nœuds PUMA

FIGURE 9 – Comportement de PUMA en présence de workloads concurrents sur les nœuds client et serveur et en désactivant PUMA lorsqu'une page active du serveur est désactivée.



(a) Mémoire du nœud PUMA serveur



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

FIGURE 10 – Comportement de PUMA en présence de workloads concurrents sur les nœuds client et serveur et en combinant les 2 approches.

La figure 9 montre les résultats de l'expérience précédente en désactivant PUMA pendant 3 secondes lorsque que ce 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 les rééquilibrer. En effet, on peut remarquer qu'il s'écoule environ 1 minute entre le début du benchmark du serveur et l'arrêt de PUMA, alors que l'approche précédente désactivait PUMA immédiatement.

3.3.3 Combinaison des deux approches

Afin de bénéficier des avantages des deux approches, nous proposons de les combiner pour détecter l'activité du serveur à la fois rapidement (1re approche) et de façon stable (2e approche). Nous avons reproduit l'expérience précédente et nous présentons les résultats dans la figure 10. Comme nous pouvions nous y attendre, la combinaison des deux approches permet de détecter l'activité du serveur rapidement, ce qui permet de désactiver PUMA pour que le serveur puisse utiliser son cache sans être pénalisé par le client.

4 Conclusion

Dans les architectures de type *cloud computing*, l'utilisation intensive de la virtualisation entraîne une fragmentation importante de la mémoire. Si des solutions comme PUMA permettent de mutualiser la mémoire *inutilisée* des MVs pour faire du cache, leur configuration est souvent statique et manuelle.
Dans ce papier, nous présentons et évaluons plusieurs mécanismes, intégrés à PUMA et au noyau Linux. Ces mécanismes permettent d'automatiser la récupération de la mémoire inutilisée des MVs. La quantité de mémoire prêtée évolue en fonction des besoins et en cas de forte pression mémoire côté serveur, le service est automatiquement suspendu.
Nos évaluations montrent que les MVs sont capables de prêter efficacement leur mémoire inutilisée, et qu'elles peuvent la récupérer rapidement sans dégrader les performances.

Bibliographie

1. 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*, *SOSP '03*, pp. 164–177. ACM, 2003.
2. Birke (R.), Podzimek (A.), Chen (L. Y.) et Smirni (E.). – 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 '13*, pp. 1–12. IEEE, 2013.
3. Capitulino (L.). – *Automatic Memory Ballooning*. – 2013. KVM Forum.
4. Chen (P. M.) et Noble (B. D.). – When virtual is better than real. – In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, *HOTOS '01*, pp. 133–138. IEEE, 2001.
5. Corbet (J.). – Better active/inactive list balancing. – `http://lwn.net/Articles/495543`, 2012.
6. Hwang (J.), Uppal (A.), Wood (T.) et Huang (H.). – Mortar : Filling the gaps in data center memory. – In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *VEE '14*, pp. 53–64. ACM, 2014.
7. 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, 2007.
8. Lorrillere (M.), Sopena (J.), Monnet (S.) et Sens (P.). – PUMA : Un cache distant pour mutualiser la mémoire inutilisée des machines virtuelles. – In *ComPAS'2014 : Conférence d'informatique en Parallélisme, Architecture et Système*, *ComPAS'2014*, avril 2014.
9. Salomie (T.-I.), Alonso (G.), Roscoe (T.) et Elphinstone (K.). – Application level ballooning for efficient server consolidation. – In *Proceedings of the 8th ACM European Conference on Computer Systems*, *EuroSys '13*, p. 337–350. ACM, 2013.
10. Schopp (J. H.), Fraser (K.) et Silbermann (M. J.). – Resizing memory with balloons and hotplug. – In *Proceedings of the Linux Symposium*, pp. 313–319, 2006.
11. Waldspurger (C. A.). – Memory resource management in VMware ESX Server. – In *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, *OSDI '02*, pp. 181–194. ACM, 2002.
12. Weiner (J.). – Thrash detection-based file cache sizing. – Linux kernel commit a528910e12ec7ee203095eb1711468a66b9b60b0, avril 2014.
13. Zhao (W.) et Wang (Z.). – Dynamic memory balancing for virtual machines. – In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *VEE '09*, pp. 21–30. ACM, 2009.