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

Optimisation mémoire dans une architecture NUMA : comparaison des gains entre natif et virtualisé

Gauthier Voron[1], Gaël Thomas[2], Pierre Sens[1], Vivien Quéma[3]

[1]Laboratoire d'informatique de Paris 6, {gauthier.voron,pierre.sens}@lip6.fr
[2]Telecom SudParis, gael.thomas@telecom-sudparis.eu
[3]Grenoble INP, vivien.quema@imag.fr

Résumé
L'exécution d'applications dans une architecture NUMA nécessite la mise en œuvre de politiques adaptées pour pouvoir utiliser efficacement les ressources matérielles disponibles. Différentes techniques permettant à un système d'exploitation d'assurer une bonne latence mémoire sur de telles machines ont déjà été étudiées. Cependant, dans le cloud, ces systèmes d'exploitation s'exécutent dans des machines virtuelles sous la responsabilité d'un hyperviseur, qui est soumis à des contraintes qui lui sont propres. Dans cet article, nous nous intéressons à ces contraintes et à la manière dont elles affectent les politiques NUMA déjà existantes. Nous étudions pour cela les effets d'une technique d'optimisation mémoire connue dans un système virtualisé et les comparons avec ceux obtenus dans un système d'exploitation.

Mots-clés : système, virtualisation, cloud, NUMA, comparaison de performances

1. Introduction

Dans le cloud, l'utilisation d'architectures fortement multicœur permet de réduire le nombre de machines utilisées, et ainsi, de réaliser des économies énergétiques et financières. Cependant, pour des applications fortement multi-threadées, comme des applications scientifiques [19], d'analyse de données [12, 17, 18] ou pour des serveurs [8, 15], il est difficile d'exploiter efficacement ce matériel car il repose généralement sur une architecture NUMA (Non-Uniform Memory Access). Dans une architecture NUMA, l'espace d'adressage physique est partitionné entre les différents nœuds NUMA, reliés par un réseau interne, chaque nœud disposant de plusieurs cœurs et d'un banc mémoire local.
Dans le contexte du cloud, l'hyperviseur, qui simule plusieurs machines virtuelles sur une seule machine physique, masque la topologie NUMA aux systèmes invités. Par conséquent, ces derniers allouent de la mémoire depuis des nœuds choisis au hasard, ce qui amène les threads applicatifs à accéder principalement à de la mémoire distante. Ces accès distants provoquent des congestions sur le réseau interne et font apparaître des goulots d'étranglement.
Plusieurs systèmes ont été proposés pour éviter ce problème de congestion dans les architectures NUMA [2, 9]. Pour fonctionner, ces systèmes ont besoin de connaître la topologie NUMA de la machine. Cependant, dans le cloud, il n'est pas souhaitable que les systèmes invités aient accès à la topologie de la machine hôte. En effet, pour équilibrer la charge à l'échelle du centre de calcul, une machine virtuelle est susceptible de migrer vers un hôte avec une topologie

NUMA différente [13, 16]. Les systèmes d'exploitation actuels ne sont pas capables de s'adapter dynamiquement à une topologie NUMA qui pourrait se modifier à chaud et de manière imprévisible.
Dans cet article, nous étudions comment les heuristiques de placement mémoire proposées dans le contexte des systèmes d'exploitation se comportent dans un contexte virtualisé. Ce nouveau contexte pose de nouvelles difficultés du fait de l'aspect boîte noire des machines virtuelles, interdisant des politiques d'allocation fines, par processus et à la demande.
En particulier, cette étude montre que :

— L'équilibrage de la charge mémoire est un facteur de performance aussi bien dans un système virtualisé que natif.
— Les algorithmes d'équilibrage de charge conçus pour des systèmes natifs fonctionnent dans des systèmes virtualisé.
— Si le fait d'améliorer la localité des données a une importance dans les systèmes natifs, soit il n'en va pas de même avec les systèmes virtualisés, soit les méthodes utilisées pour améliorer ce critère ne fonctionnent pas dans un hyperviseur.

La suite de cet article présente en section 2 les raisons qui nous poussent à étudier cette différence comportementale entre systèmes natifs et virtualisés et en section 3 un état de l'art sur la virtualisation. La section 4 décrit les moyens par lesquels nous portons un système de gestion mémoire NUMA dans un contexte de virtualisation. Enfin la section 5 détaille les résultats obtenus par ce système et propose une explication puis la section 6 conclue.

2. Motivation

L'utilisation efficace des ressources dans une architecture NUMA par un système d'exploitation est une problématique déjà largement étudiée [6, 7, 11]. Ces études montrent que le temps d'accès à une donnée en mémoire dépend principalement du niveau de saturation des bus, une congestion de ces bus provoquant une dégradation importante de la latence d'accès aux données. Plus spécifiquement, une utilisation efficace de la mémoire est conditionnée par :

— Une répartition de la charge sur les différents contrôleurs mémoire.
— Un placement des données maximisant les accès locaux.

Un système peut répartir équitablement sa charge mémoire en déplaçant les données de manière aléatoire sur les nœuds NUMA exploités. De même, ce système peut détecter, via différentes techniques de profiling, à quelles données accèdent les cœurs d'un nœud NUMA et déplacer ces données dans la mémoire locale de ce nœud.
Les hyperviseurs diffèrent cependant des systèmes dans la gestion de la mémoire du fait des contraintes suivantes :

— Les volumes de mémoire traités sont plus importants, là où un système supervise des applications, un hyperviseur supervise des systèmes et toutes leurs applications. Ces quantités de mémoire plus importantes rendent plus difficiles la mise en œuvre de politiques basées sur le déplacement de pages mémoire puisqu'il est nécessaire de migrer de plus grandes quantités de données pour avoir un impact visible, ce qui entraîne un surcoût plus important.
— L'allocation de la mémoire se fait une seule fois au démarrage de la machine virtuelle et n'est désallouée qu'à son extinction. L'utilisation du ballooning ou de mécanismes similaires n'est par ailleurs pas envisageable car trop intrusif. Il n'est donc pas possible pour l'hyperviseur d'utiliser des politiques d'allocation « NUMA friendly » qui permettent de placer les données mémoire à la meilleure place possible dès leur allocation.

Ces différences peuvent affecter les stratégies jusque là étudiées. Cet article se propose d'étu-

dier ces différences.

3. Contexte

La virtualisation consiste à donner l'illusion à un système « invité » qu'il s'exécute en mode privilégié sur une machine alors que ce n'est pas le cas, dans un but d'isolation. La virtualisation logicielle consiste à exécuter le code du système invité en remplaçant les instructions privilégiées par des traitements de substitution. Les instructions privilégiées sont détectées soit quand elles génèrent une faute de protection dans le système hôte, soit par traduction binaire. Dans le cas de la virtualisation matérielle [1, 14], le processeur est doté d'un mode d'exécution « hôte » , classiquement utilisé lorsque le processeur n'est pas en mode virtualisé, et dans lequel s'exécute l'hyperviseur, et d'un mode d'exécution « invité ». Quand un système en mode invité tente d'exécuter une instruction privilégiée, le processeur le détecte et exécute un traitement de substitution préalablement spécifié par l'hyperviseur. Le système invité peut aussi être averti qu'il est virtualisé et coopérer avec l'hyperviseur pour obtenir de meilleures performances, on parle alors de paravirtualisation [3]. La virtualisation matérielle étant à la fois efficace et indépendante du système invité, c'est elle qui sera étudiée dans cet article. La plupart des techniques présentées peuvent toutefois s'appliquer à la virtualisation logicielle.

3.1. Virtualisation matérielle

Les processeurs supportant la virtualisation matérielle ajoutent un mode d'exécution invité, orthogonal aux niveaux de privilèges. Un système en mode invité peut ainsi exécuter des instructions privilégiées, qui sont détectées par le processeur et remplacées par le traitement associé indiqué par le système hôte. Les interruptions matérielles sont systématiquement redirigées vers le système hôte. Ce dernier peut donc régulièrement, grâce à l'interruption horloge, reprendre la main et commuter entre plusieurs systèmes invités. A chaque système invité est associé un état virtuel de la machine, stocké en mémoire. Quand le système invité tente de d'accéder ou de modifier une ressource privilégiée, c'est cet état virtuel qui est utilisé.
Le processeur permet également d'associer à chaque machine virtuelle un espace d'adressage différent, au moyen d'un second niveau de table des pages (voir Figure 1). Lorsqu'un système invité accède à la mémoire via une adresse virtuelle, la MMU (Memory Management Unit) traduit cette dernière via la table des pages maintenue par le système invité, pointée par le registre CR3 virtuel. Le résultat de cette traduction est une adresse physique, uniquement valide pour le système invité responsable de la traduction. La MMU traduit ensuite cette adresse physique en adresse machine via une seconde table des pages associée à la machine virtuelle, et maintenue par le système hôte. Cette adresse machine est compréhensible par le contrôleur mémoire.

3.2. Échantillonage et virtualisation

Indépendamment de la virtualisation, les processeurs modernes supportent différents types de compteurs matériels, utilisables à des fins d'introspection. Le type le plus simple à utiliser est incrémenté par le matériel à chaque fois qu'un évènement se produit. Le type d'évènement est programmable et la valeur du compteur peut être relevée régulièrement ou sur interruption. Ce type de compteur n'engendre qu'un faible surcoût. Les processeurs x86 offrent également une capacité d'échantillonage matériel (sampling) : régulièrement, une instruction en cours d'exécution est sélectionnée, décodée, puis le processeur transmet le résultat de ce décodage au système via une interruption. Les informations ainsi transmises peuvent renseigner sur l'adresse mémoire accédée, si la donnée est présente dans le cache L2, etc... Cette méthode permet de

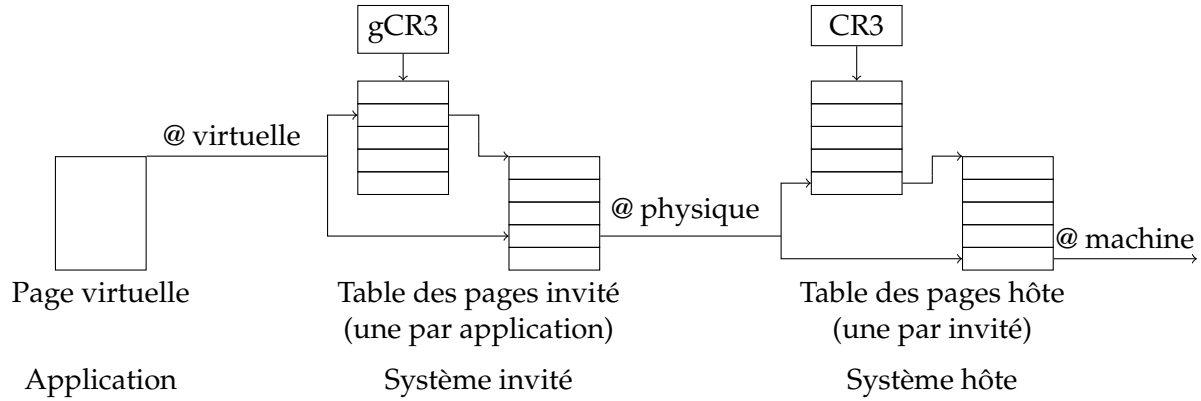

FIGURE 1 – Schéma d'une traduction d'adresse en virtualisation matérielle. L'adresse virtuelle d'une page de l'application est traduite via la table des pages du système invité, pointée par gCR3 en adresse physique qui est à son tour traduite via la table des pages du système hôte, pointée par CR3, en adresse machine. Par soucis de clareté, les tables des pages sont représentées avec seulement deux niveaux.

construire une vue précise et statistiquement valide du comportement de la machine, mais peut provoquer un nombre important d'interruptions, et donc générer un surcoût important.
Ces technologies d'échantillonage sont pour l'instant difficilement compatibles avec la virtualisation matérielle. La technologie IBS [10] (Instruction Based Sampling) d'AMD par exemple, fournit les adresses virtuelle et machine d'une instruction mémoire, en revanche, l'obtention de l'adresse physique nécessite un parcours logiciel de la table des pages du système invité. La technologie PEBS [14] (Precise Event Based Sampling) d'Intel est quant à elle inapplicable à la virtualisation matérielle. En effet, le résultat du décodage d'instructions est stocké dans une zone mémoire spécifiée par une adresse virtuelle, qui sera traduite différemment pour chaque système invité et pour l'hyperviseur. Il est donc difficile pour système hôte d'assurer où le résultat du décodage d'une instruction en mode invité sera écrit, sans la coopération du système invité.

3.3. L'hyperviseur Xen

Xen est un hyperviseur open source, utilisé dans l'industrie, initialement conçu pour la paravirtualisation mais capable d'utiliser la virtualisation matérielle. Un hyperviseur Xen s'exécute directement au dessus du matériel (type 1) et est administré via une machine virtuelle privilégiée, appelée dom0, par opposition aux autres machines virtuelles appelées domU. Lorsque l'administrateur en dom0 démarre un nouveau domU, il spécifie la quantité de mémoire attribuée et sur quels cœurs physiques peut s'exécuter le nouveau domaine. Xen pré-alloue alors l'espace physique de la machine virtuelle dans la mémoire machine et remplit intégralement la table des pages hôte du nouveau domaine avant de le démarrer. Dans le cas d'une architecture NUMA, l'administrateur peut restreindre le nouveau domaine à un sous-ensemble de nœuds NUMA. Si plusieurs nœuds NUMA sont possibles pour un domU, la mémoire allouée est répartie équitablement entre ces nœuds, en tourniquet avec une granularité d'allocation de 1Gio. Bien que Xen soit capable de reconnaître et de s'adapter à une architecture NUMA, il en masque la topologie aux systèmes invités, leur donnant l'illusion de s'exécuter sur une architecture SMP (Symetric Multi-Processor).

4. Conception et mise en œuvre

Carrefour [9] est un système implanté dans Linux, permettant de réduire la contention sur les liens mémoire d'une architecture NUMA et d'augmenter la proportion d'accès locaux. Nous proposons de porter Carrefour dans l'hyperviseur Xen puis d'étudier son comportement. Carrefour est composé d'un module noyau et d'un programme en espace utilisateur, communiquant via le procfs. Le programme utilisateur mesure, à l'aide de compteurs matériels, la quantité d'accès mémoire, le taux d'accès locaux et le rapport lecture/écriture. A partir de ces mesures, le programme décide et informe le module noyau de la politique globale à adopter : ne rien faire, activer la réplication, l'équilibrage ou la collocation. Sauf dans le cas de la première politique, le module noyau utilise IBS pour déterminer quelles sont les pages chaudes et par quels cœurs elles sont accédées. Selon le type d'accès et la ou les politiques globales activées, ces pages seront ensuite répliquées, réparties ou déplacées vers les nœuds accesseurs.

4.1. Le module noyau

La notion de module n'existe pas dans Xen, aussi le module noyau est-il intégré de manière monolithique dans l'hyperviseur. Le module original, pour déplacer les données de manière transparente pour l'application, les copie d'une adresse physique à une autre, puis associe via la table des pages l'adresse virtuelle de ces données à la nouvelle adresse physique. La page ainsi déplacée est interdite en écriture durant l'opération. Notre module modifié effectue la même séquence d'opérations mais en travaillant sur la seconde table des pages, qui associe adresses physiques et adresses machines. Les données sont ainsi transférées de manière transparente pour le système invité et ses applications utilisateur, indépendamment du grain d'allocation utilisé par Xen. Xen aussi bien que Linux fournit une API pour IBS. Le module original utilise les résultats d'échantillonage pour obtenir les adresses virtuelles et physiques d'une donnée accédée. Cependant, quand la virtualisation matérielle est activée, un résultat d'échantillonage fournit uniquement l'adresse virtuelle et l'adresse machine de la donnée accédée, or notre module doit connaître l'adresse physique pour pouvoir modifier la table des pages. La solution adoptée est, dans le traitement de l'interruption IBS, de parcourir la table des pages du système invité à partir du registre CR3 virtuel pour obtenir l'adresse physique à partir de l'adresse virtuelle. Ce parcours ne peut se faire que dans le traitement de l'interruption, autrement le système invité pourrait commuter et changer de table de page. Pour permettre à l'interruption de se terminer plus rapidement, le résultat d'échantillonage est ignoré et l'interruption se termine immédiatement si le verrou d'une section critique ne peut être obtenu. Dans les expériences présentées, cet évènement se produit une fois pour mille interruptions.

4.2. Le programme utilisateur

Le programme utilisateur d'origine se sert des compteurs matériels via des appels systèmes pour choisir quelle politique globale appliquer, puis communique cette décision au module via le procfs. De la même manière qu'un système fournit des appels systèmes, l'hyperviseur Xen fournit des appels hyperviseurs, utilisables depuis le domaine privilégié dom0, déclenchables depuis le mode utilisateur et relayés par le système invité. Notre version modifiée de Xen fournit deux nouveaux appels hyperviseur, le premier permet de lire et modifier les registres qui contrôlent le comptage matériel, la sélection et la configuration du compteur étant des paramètres. Le second appel hyperviseur permet de communiquer avec le module Carrefour embarqué, en offrant la même interface qu'une écriture dans un fichier du procfs. Notre programme modifié, exécuté dans un processus utilisateur du dom0, utilise ces appels hyperviseur pour assurer un comportement identique au programme d'origine.

5. Évaluation de performance

Par une évaluation d'un système de placement mémoire comme Carrefour dans des environnements virtualisés, nous cherchons à comprendre quels sont les facteurs importants de performance dans de tels environnements et en quoi le problème de placement mémoire diffère par rapport à un environnement non virtualisé. Les mesures de performance sont obtenues à partir des suites de test Parsec [5] v2.1 et NAS Parallel Benchmark [4] v3.3.1. Les applications choisies dans ces suites de test sont limitées par la latence d'accès à la mémoire. Les applications de la suite Parsec manipulent des données structurées, offrant des motifs d'accès réguliers avec des granularités et des taux de partage entre threads différents. Les applications de la suite NAS Parallel Benchmark au contraire effectuent des accès mémoire aux motifs irréguliers. Ces applications sont exécutées sur les machines aux caractéristiques suivantes :

— Les deux machines disposent de 8 nœuds NUMA répartis sur 4 sockets, chaque nœud a 6 cœurs (48 cœurs au total). Les nœuds sont interconnectés avec un lien HyperTransport 3.0.
— La machine A dispose de 32 Gio de mémoire par nœud (256 Gio au total), et de cœurs AMD Opteron 6172 2.2 GHz.
— La machine B dispose de 16 Gio de mémoire par nœud (128 Gio au total), et de cœurs AMD Opteron 6174 2.2 GHz.

Nous utilisons un noyau Linux 3.9 et l'hyperviseur Xen 4.5. Pour les tests avec virtualisation, les applications sont exécutées dans un domU avec 48 cœurs virtuels, chacun attaché à un cœur physique différent, une mémoire totale de 110 Gio. Sur cette machine virtuelle, nous utilisons le même noyau Linux 3.9 et les mêmes bibliothèques logicielles que pour les tests sans virtualisation.

5.1. Résultats

5.1.1. Expérience 1 : grain d'allocation

On cherche dans un premier temps à mesurer l'impact de la répartition de la charge mémoire sur les nœuds NUMA. Par défaut, Xen alloue la mémoire qu'il attribue aux systèmes invités par bloc continus de 1 Gio, chaque bloc étant alloué en tourniquet sur chaque nœud NUMA autorisé. Avec ce mode d'allocation, les applications dont les données occupent moins de 1 Gio de mémoire résident entièrement dans un nœud NUMA, qui supporte toute la charge mémoire. Notre version de Xen modifiée autorise des allocations avec des grains de 2 Mio ou de 4 Kio. Ces modes d'allocation permettent de répartir sur l'ensemble des nœuds NUMA, la charge des applications invitées.

La figure 2 montre le temps d'exécutions de ces applications dans Xen en fonction du grain d'allocation. Selon les applications, on observe deux cas : soit les temps sont faiblement dégradés par un grain d'allocation plus fin, soit ils sont sensiblement améliorés. Les applications du second cas utilisent une faible quantité de mémoire, relativement au grain d'allocation de base de 1 Gio et on peut observer qu'une allocation à grain plus fin, qui permet de répartir la charge, permet de meilleures performances. Cette observation confirme l'importance de la répartition de la charge mémoire dans les systèmes virtualisés. Les autres applications, qui occupent un espace mémoire plus importants bénéficient déjà d'un bon équilibrage avec un grain d'allocation de 1 Gio. La dégradation est due à une augmentation du nombre de TLB miss.

5.1.2. Expérience 2 : localité dans Xen

On cherche dans un second temps à savoir quels gains peut apporter un système comme Carrefour et à quelles conditions. L'expérience précédente montre que plus le grain d'allocation

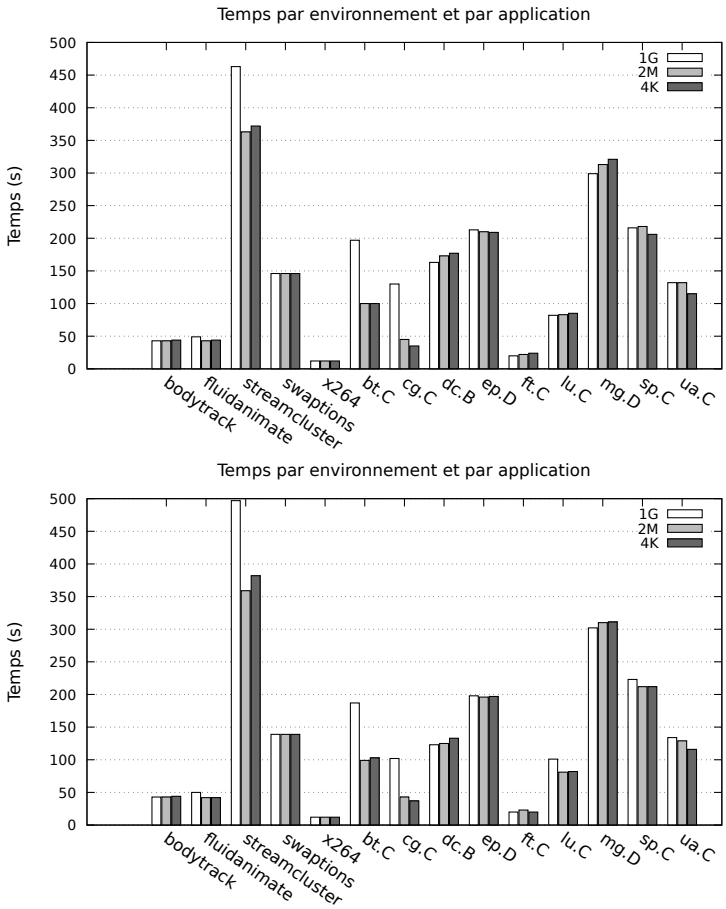

FIGURE 2 – Temps par application et par grain d'allocation sur les machines A (en haut) et B (en bas)

est fin, mieux la charge est équilibrée. Aussi plus le grain d'allocation est-il fin, moins les gains obtenus avec notre Carrefour modifié dépendent-ils de l'équilibrage de la charge.
La figure 3 montre le gain obtenu avec le système Carrefour pour Xen, pour les mêmes applications et en fonction du grain d'allocation. La formule pour calculer le gain obtenu est la même qu'utilisée par Dashti et al. :

$$\frac{\mathrm{XenBaseline}_{\mathrm{temps}} - \mathrm{XenCarrefour}_{\mathrm{temps}}}{\mathrm{XenCarrefour}_{\mathrm{temps}}} * 100\%$$

On observe le phénomène suivant : avec un grain d'allocation de 1 Gio, Carrefour améliore le temps d'exécution de certaines applications. Dans l'expérience précédente, ces mêmes applications (streamcluster, bt.C, cg.C, lu.C et ua.C) sont celles dont les temps d'exécution sont améliorés par l'utilisation d'un grain d'allocation plus fin. Le gain apporté par Carrefour a donc pour cause un meilleur équilibrage de la mémoire. D'autre part, avec des grains d'allocation plus fin, de 2 Mio et 4 Kio, la charge est déjà équilibrée, et on observe que Carrefour, qui ne peut alors qu'améliorer la localité, dégrade systématiquement les performances des applications. On peut en conclure que dans un système virtualisé, soit Carrefour est incapable d'améliorer la localité, soit cette amélioration n'apporte pas de gain significatif.

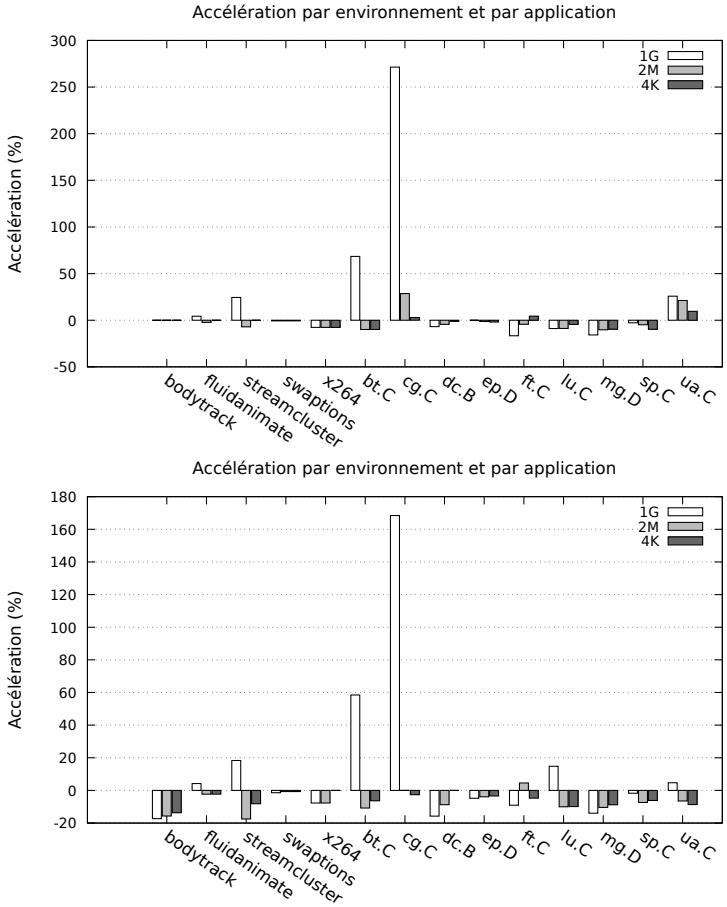

FIGURE 3 – Accélération apportée par Carrefour par application et par grain d'allocation sur les machines A (en haut) et B (en bas)

5.1.3. Expérience 3 : comparaison avec Linux

L'objectif de cette expérience est de vérifier si l'impossibilité de Carrefour à gagner en performance via la localité est due à la virtualisation. Linux permet aux applications utilisateur de spécifier sur quels nœuds NUMA allouer la mémoire. Dans une première politique, « first-touch », une page est allouée sur le nœud NUMA qui fait la demande d'allocation, dans une seconde, « interleave », les pages sont allouées en tourniquet avec un grain de 4 Kio. Dans la plupart des applications considérées, les données sont allouées par le thread maître, par conséquent ces données se trouvent sur le même nœud NUMA. La politique « first-touch » cause donc un mauvais équilibrage de charge, à la différence de la politique « interleave ».

La figure 4 montre le gain obtenu avec le système Carrefour pour Linux sans virtualisation, pour les mêmes applications et pour les politiques « first-touch » et « interleave ». On observe que le système Carrefour améliore le temps des applications dont la charge est déséquilibrée en « first-touch » (streamcluster, bt.C, ep.D, ft.C et mg.D). Cela montre que sans virtualisation, Carrefour améliore bien l'équilibrage de charge, de la même manière que dans un système virtualisé. En revanche avec une politique « interleave », même si certaines applications sont impactées négativement par Carrefour, cette dégradation est toujours inférieure à 10%. De plus, d'autres applications (streamcluster, ep.D, ft.C, mg.D) ont de meilleurs temps d'exécution avec

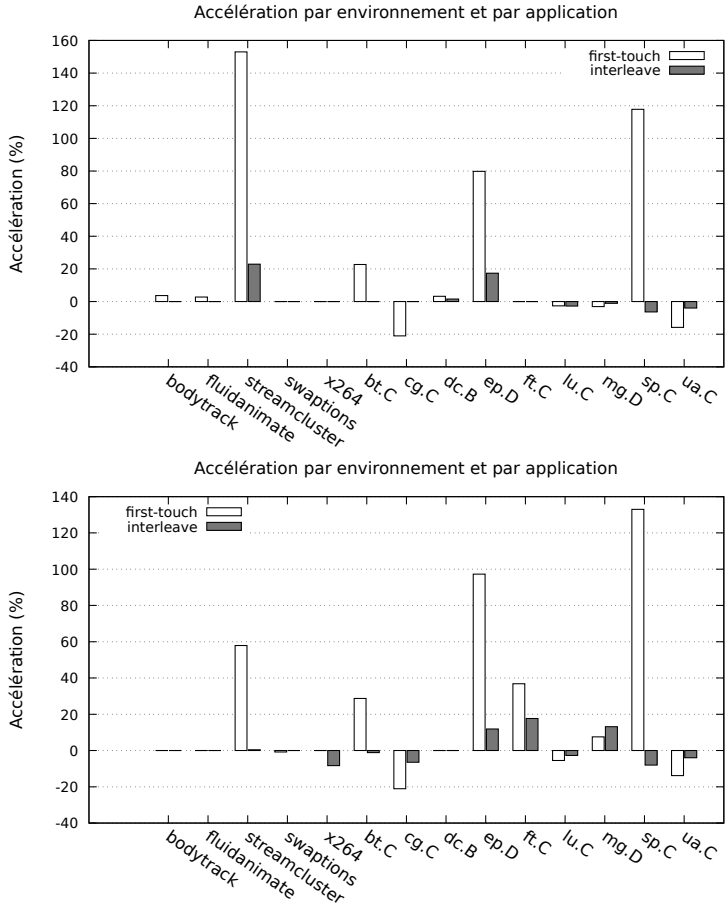

FIGURE 4 – Accélération apportée par Carrefour par application dans un système Linux avec et sans répartition mémoire sur les machines A (en haut) et B (en bas)

Carrefour, malgré une allocation en tourniquet. On peut en conclure que dans un système sans virtualisation, Carrefour est capable d'améliorer les performances via la localité, même si le gain apporté est faible en comparaison de l'équilibrage de charge.

6. Conclusion

Dans cet article, nous présentons un comparatif de gains obtenus entre un environnement natif et un environnement virtualisé dans une architecture NUMA avec le système de placement mémoire Carrefour, correspondant à l'état de l'art. Ce comparatif est réalisé via un portage du système Carrefour dans Xen, dont les difficultés techniques, liées à la virtualisation sont listées. Ce comparatif montre que si l'équilibrage de la charge mémoire est un facteur aussi important avec et sans virtualisation, celui-ci est déjà géré correctement par les algorithmes existants. En revanche, la localité des données, bien qu'étant non négligeable dans un environnement natif est soit sans importance, soit mal géré dans un contexte de virtualisation. Dans nos futurs travaux, nous souhaitons étudier si une bonne répartition de la charge mémoire est l'unique critère à prendre en compte pour améliorer les performances dans un système virtualisé. Si tel est le cas, y a-t-il une politique plus efficace qu'une allocation en tourniquet, si non, quels sont les autres critères et comment les améliorer.

Bibliographie

1. *AMD64 Architecture Programmer's Manual Volume 2 : System Programming*. – 2013.
2. Autonuma - the other approach to numa scheduling. – `http://lwn.net/Articles/488709/`, 2012.
3. 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. *SIGOPS Oper. Syst. Rev.*, vol. 37, n5, octobre 2003, pp. 164–177.
4. Nas parallel benchmarks. – `http://www.nas.nasa.gov`.
5. Parsec benchmark suite. – `http://parsec.cs.princeton`.
6. Blagodurov (S.), Zhuravlev (S.), Dashti (M.) et Fedorova (A.). – A case for numa-aware contention management on multicore systems. – In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, *USENIXATC'11*, USENIXATC'11, pp. 1–1, Berkeley, CA, USA, 2011. USENIX Association.
7. Bolosky (W.), Fitzgerald (R.) et Scott (M.). – Simple but effective techniques for numa memory management. *SIGOPS Oper. Syst. Rev.*, vol. 23, n5, novembre 1989, pp. 19–31.
8. Apache cassandra. – `http://cassandra.apache.org/`, 2008.
9. Dashti (M.), Fedorova (A.), Funston (J.), Gaud (F.), Lachaize (R.), Lepers (B.), Quema (V.) et Roth (M.). – Traffic management : A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, vol. 41, n1, mars 2013, pp. 381–394.
10. Drongowski (P. J.) et Center (B. D.). – Instruction-based sampling : A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices, Inc*, 2007.
11. Gidra (L.), Thomas (G.), Sopena (J.), Shapiro (M.) et Nguyen (N.). – Numagic : A garbage collector for big data on big numa machines. – In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS '15*, ASPLOS '15, pp. 661–673, New York, NY, USA, 2015. ACM.
12. Apache hadoop. – `https://hadoop.apache.org/`, 2009.
13. Hermenier (F.), Lorca (X.), Menaud (J.-M.), Muller (G.) et Lawall (J.). – Entropy : A consolidation manager for clusters. – In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *VEE '09*, VEE '09, pp. 41–50, New York, NY, USA, 2009. ACM.
14. *Intel® 64 and IA-32 Architectures Software Developer's Manuals, Volume 3*. – 2014.
15. Memcached - a distributed memory object caching system. – `http://memcached.org/`, 2004.
16. Nagin (K.), Hadas (D.), Dubitzky (Z.), Glikson (A.), Loy (I.), Rochwerger (B.) et Schour (L.). – Inter-cloud mobility of virtual machines. – In *Proceedings of the 4th Annual International Conference on Systems and Storage*, *SYSTOR '11*, SYSTOR '11, pp. 3 :1–3 :12, New York, NY, USA, 2011. ACM.
17. Neo4j - the world's leading graph database. – `http://www.neo4j.org/`, 2010.
18. Apache spark - lightning-fast cluster computing. – `http://spark.apache.org/`, 2010.
19. Yin (J.), Wang (J.), Feng (W.-c.), Zhang (X.) et Zhang (J.). – Slam : Scalable locality-aware middleware for i/o in scientific analysis and visualization. – In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, *HPDC '14*, HPDC '14, pp. 257–260, New York, NY, USA, 2014. ACM.