Compas'2018 : Parallélisme/ Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Scalevisor : Un pilote CPU et mémoire pour les gros multicœurs

Alexis Lescouet[1], Nicolas Derumigny[2], Gaël Thomas[1]

[1]Télécom SudParis
[2]ENS Lyon

Résumé
Ces dernières années, le besoin de puissance de calcul a conduit à l'apparition de nouvelles architectures complexes utilisant le parallélisme pour gagner en puissance.
Or, ces machines ne produisent que des performances médiocres si la gestion de ces ressources (mémoires, CPUs) ne permet pas de tirer profit du parallélisme. Malheureusement, l'introduction de nouvelles heuristiques de gestion de la mémoire dans les noyaux existants est un travail complexe qui requiert la modification de nombreuses parties du code.
Plutôt que de modifier en profondeur le noyau, nous proposons de mettre en œuvre un pilote de périphériques dédié à la gestion de ces ressources et d'utiliser des techniques de virtualisation pour rendre ce pilote transparent pour le noyau.
Ce pilote permettra la mise en œuvre de nouvelles heuristiques qui seront adaptables selon les spécificités du matériel et des applications.

Mots-clés : système, virtualisation, NUMA

---

1. Introduction

Avec la croissance de domaines comme l'analyse de données ou la simulation, le besoin en puissance de calcul a grandement augmenté ces dernières années. Les machines modernes utilisées pour ce type d'application sont des machines complexes composées d'un ensemble de nœuds connectés par un réseau rapide appelé "interconnect". Chacun de ces nœuds est constitué de plusieurs CPUs, d'un banc mémoire et de sa hiérarchie de caches. En raison de cette topologie, ces machines sont qualifiées de mémoires à accès non uniformes (non uniform memory access, NUMA).
Bien qu'intrinsèquement puissantes en raison de la grande quantité de ressources dont elles disposent, un mauvais placement des tâches ou de la mémoire cause une forte dégradation des performances. En effet, le temps d'accès à de la mémoire située sur un nœud distant possède un coût supérieur aux accès locaux [3], mais qui reste du même ordre de grandeur. En revanche, certains travaux récents ont montré que les performances s'écroulaient lorsque trop d'accès distants conduisent à la saturation des contrôleurs mémoire et de l'interconnect en raison de l'importante latence qui en découle [2].
De plus, le dernier niveau de la hiérarchie de cache de ces machines qui permet un accès rapide aux données récemment accédées est partagé entre plusieurs CPUs. Or lorsqu'un CPU remplace de façon abusive les données du cache partagé il oblige les autres CPUs à accéder à la mémoire de façon directe, ce qui participe d'autant plus à saturer les contrôleurs mémoire.

Puisque le placement sur ce genre de machines est critique, il est primordial d'avoir de bonnes heuristiques proposées par le système.

— Pour éviter ces dégradations de performances, le placement doit prendre un compte plusieurs ressources (mémoire, CPU, I/Os) [4]. Ce problème à déjà été abordé dans des travaux existants mais seulement en tenant compte de certaines ressources considérées statiques pour déterminer le placement d'une unique ressource mobile.
Il serait donc possible d'améliorer grandement ces performances en mettant en œuvre des heuristiques prenant en compte simultanément le placement de la mémoire et des tâches tout en tenant compte des entrées/sorties.
— De plus, une même politique de placement ne peut convenir à tous les types d'applications [3]. Les systèmes d'exploitation actuels proposent des heuristiques basiques pour la gestion des ressources, mais du fait de leur trop grande généralité, celles-ci ne permettent pas une amélioration suffisamment importante.
— Enfin, les systèmes d'exploitation comme Linux sont des programmes complexes, qui contiennent des millions de lignes de code. Il est donc impossible d'y intégrer de nouvelles politiques car cela impliquerait d'effectuer des modifications dans un trop grand nombre de structures internes [4].

Dans ce papier, nous proposons de modifier le design de la pile logicielle communément utilisée en système pour mieux exploiter les ressources des architectures multicœurs.
Pour commencer, nous proposons de résoudre le problème de maintenablilité en effectuant le placement des ressources en dehors du système d'exploitation. Ensuite, nous proposons de prendre en compte différents types de ressources et de les gérer grâce aux techniques de virtualisation qui permettent de dissimuler la répartition de ces ressources au système. Enfin, nous proposons de satisfaire les besoins différents de multiples applications en utilisant un système de modules permettant de déployer des politiques spécifiques adaptées à des applications spécifiques.
Outre le design, nous proposons une mise en oeuvre nommée Scalevisor. Scalevisor, développé sans base de code existante, est en cours de développement et est actuellement capable de démarrer un Linux monocœur. Nous sommes actuellement dans la phase de mise en oeuvre des premières politiques. La mise en œuvre proposée cible des machines NUMA, dotées de processeurs x86 compatibles avec les avancées technologiques récentes en matière de virtualisation.
La suite du papier est organisée comme suit. La section 2 expose le design du pilote, la section 3 présente une mise en œuvre de ce design, Scalevisor. La section 4 décrit le système de module permettant à des applications d'influencer la politique de placement. La section 5 discute différents détails liés à la virtualisation et aux performances. Enfin, la section 6 conclut ce papier.

2. Design de Scalevisor

Habituellement, la gestion du placement des processeurs et de la mémoire revient exclusivement au système d'exploitation. Pour permettre à notre pilote de prendre le contrôle de ces ressources nous avons modifié le design de la pile logiciel classique qui place le système d'exploitation directement au-dessus du matériel.
Dans cette nouvelle pile logiciel, Scalevisor prend le contrôle des ressources matérielles et pour permettre la compatibilité avec le système d'exploitation expose une architecture abstraite simple (fig. 1). Cette topologie abstraite permet aussi que le placement des ressources effectué par la suite par le pilote soit totalement transparent pour le système.
Lorsque nous aurons mis en œuvre une version fonctionnelle de notre système de placement,

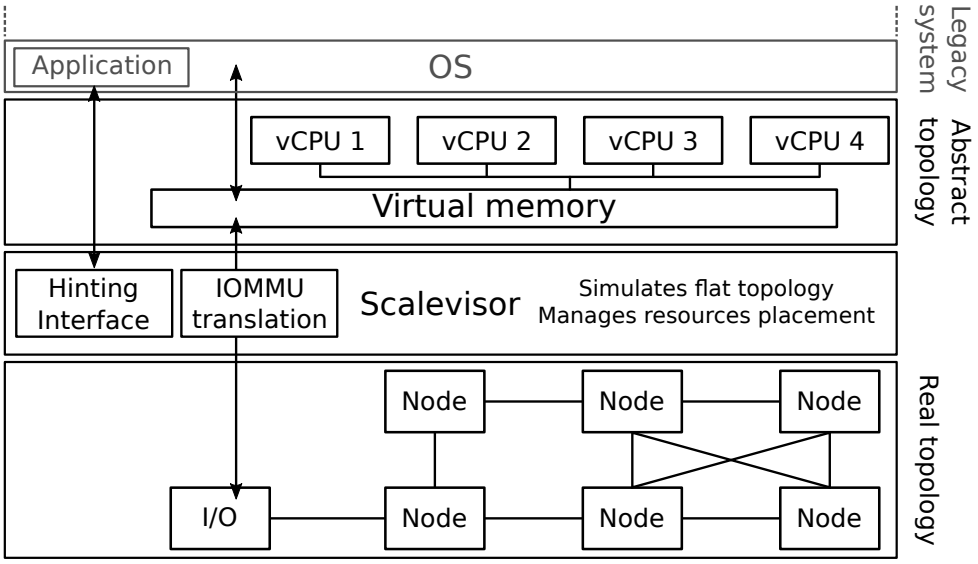

FIGURE 1 – Une pile logiciel incluant Scalevisor

nous comptons donner la possibilité pour les applications du système invité de fournir des indices de placement.
Ces indices décrirons dans l'ensemble combien de tâches sont utilisées en parallèle et comment ces tâches accèdent à la mémoire allouée à l'application. Grâce à la connaissance du fonctionnement de l'application, nous pourrons améliorer sensiblement le comportement de notre algorithme de placement en tenant compte des besoins spécifiques des applications qui auront fourni des informations.

3. Mise en œuvre de Scalevisor

Les avancées matérielles récentes permettent de nombreuses améliorations en terme de virtualisation, parmi lesquelles, celle d'exécuter plusieurs systèmes invités simultanément et de façon isolée. Dans les faits, toutes ces avancées se résument à ajouter un niveau d'abstraction sur les différentes ressources (CPU, mémoire, périphériques) afin que les systèmes invités puissent partager les ressources physiques tout en ayant le total contrôle des ressources virtuelles.
Les performances des machines NUMA dépendent grandement de la gestion des ressources et donc des spécificités de l'architecture complexe de la machine. Pour améliorer ces performances, nous utilisons le niveau d'abstraction supplémentaire fourni par la virtualisation pour exposer une vue linéaire simplifiée de la topologie au système et contrôler en arrière-plan les ressources matérielles.
Les techniques de virtualisation matérielle récentes permettent à un processeur de fonctionner dans deux modes, "root" et ("non-root"). Dans le mode *root*, le logiciel a accès à toutes les fonctionnalités classiques ainsi qu'à plusieurs instructions supplémentaires qui permettent de contrôler la virtualisation. Le mode *non-root* est un mode très proche du mode classique, où seules quelques instructions ne sont pas accessibles. Le gestionnaire de machine virtuelle (Scalevisor) s'exécute dans le mode *root* tandis que le système invité (Linux) s'exécute dans le mode *non-root*.

Les nouvelles instructions introduites dans le mode *root* permettent d'interagir avec une structure de contrôle qui représente un CPU virtuel (vCPU) :

— Modifier le comportement d'un vCPU en mode *non-root*
— Exécuter un vCPU sur un CPU physique

De plus, les extensions de virtualisation fournissent un système de traduction mémoire supplémentaire ("Extended Page Table") qui permet au gestionnaire de machine virtuelle de mettre en place une traduction entre les adresses physiques accédées par le système en mode *non-root* et les adresses physiques de la mémoire machine.

3.1. Ordonnancement des tâches

L'ordonnancement des tâches utilise la possibilité d'exécuter un vCPU arbitraire sur un CPU physique, puis d'interrompre cette exécution et de la reprendre sur un autre CPU physique.
En effet, le mode *non-root* permet au système d'exécuter le code qu'il souhaite sur le CPU jusqu'à ce que le gestionnaire de machine virtuelle interrompe cette exécution, retournant ainsi en mode *root*. Il est alors possible, en déplaçant la structure de contrôle, d'ordonnancer ce CPU virtuel sur un autre CPU. Le retour en mode *root* peut être la conséquence de plusieurs événements, parmi lesquels, un compteur spécialement conçu qui peut être configuré par le gestionnaire de machine virtuelle pour interrompre l'exécution après un certain intervalle de temps.

3.2. Placement de la mémoire

Pour gérer le placement de la mémoire de façon transparente pour le système d'exploitation nous utilisons le mécanisme de traduction supplémentaire mis à disposition par les extensions de virtualisation.
En mode natif, les systèmes d'exploitation n'utilisent pas la mémoire physique de la machine de façon directe. Cela permet d'isoler la mémoire des différents processus et de simplifier la mise en œuvre de ceux-ci. Le système utilise donc une table des pages pour effectuer la traduction entre les adresses physiques, celles de la mémoire machine, et virtuelles, celles qu'utilisent les processus. L'étape de traduction, qui a lieu extrêmement souvent lors de l'exécution est réalisée par la MMU (memory management unit) qui utilise la table des pages construite par le système.
Le mécanisme de double pagination proposé par les processeurs récents nous permet donc d'exposer au système d'exploitation une topologie mémoire plate constituée des adresses physiques avant traduction, et de placer cette mémoire à l'adresse physique machine souhaitée. Pour déplacer de la mémoire lors de l'exécution, il suffit de modifier la correspondance entre l'adresse mémoire physique avant traduction qui est le choix du système d'exploitation et l'adresse finale qui sera choisie en fonction du placement optimal. Ce mécanisme est totalement transparent pour le système d'exploitation et un système de cache permet d'accélérer les traductions.
En ce qui concerne les entrées/sorties, les périphériques qui prennent en charge les requêtes émises par le système utilisent habituellement l'adressage machine pour interagir avec la mémoire. Or en raison du système de double pagination, les adresses physiques qu'utilisent le système ne sont pas les adresses physiques de la mémoire machine. Heureusement, l'IOMMU permet d'étendre le système de traduction des adresses aux périphériques. Cette assistance matérielle est donc précieuse puisqu'elle permet, entre autres avantages, d'utiliser de larges portions de mémoires non contiguës (qui apparaissent comme tel grâce à l'adressage virtuel) mais aussi de donner au système invité un accès direct et transparent aux périphériques ce qui permet d'éviter un surcoût important de prise en charge des entrées/sorties par Scalevisor.

4. Politiques de placement configurables

4.1. Heuristiques générales

Dans un premier temps nous nous proposons de mettre en œuvre des politiques de placement statiques utilisées dans Linux, "first touch", "interleaved", puis la politique de placement "Carrefour" [2].
Dans un second temps, nous mettrons en œuvre une heuristique prenant un compte simultanément le placement des tâches et de la mémoire. Cette heuristique tendra à satisfaire trois contraintes :

1. Augmenter au maximum l'équilibrage entre les différents nœuds NUMA
2. Maximiser la localité des accès mémoire et des entrées/sorties
3. Améliorer l'utilisation du cache partagé (cache de dernier niveau)

Plus précisément, nous utiliserons diverses techniques d'échantillonnage matériel afin de déterminer quels vCPUs accèdent à quelles ressources, aussi bien les accès à la mémoire que les entrées/sorties.
Grâce à ces informations, nous pourrons ensuite décider s'il est nécessaire de modifier le placement. Si tel est le cas, il s'agira alors de migrer la mémoire vers le nœud exécutant le vCPU ayant effectué l'accès ou l'inverse selon le plus avantageux.
Pour la mémoire, le coût de la migration est relativement simple, il est fonction de la taille de mémoire à migrer. En revanche, le coût de migration d'un vCPU dépend de plusieurs facteurs. Premièrement, chaque vCPU est représenté par sa structure de contrôle en mémoire qu'il faut donc également rapprocher au maximum du CPU physique exécutant le vCPU. Deuxièmement, migrer un vCPU nécessite de le stopper puis de le redémarrer, ce qui a un coût non négligeable. Étant donné ces deux coûts potentiels, il faut de surcroît prendre en compte la charge supplémentaire qu'engendre la migration de la mémoire sur l'interconnect afin de ne pas provoquer de saturation tandis que le système continue son exécution.
En temps normal, le système d'exploitation pourra exploiter la totalité des CPU physiques de la machine. Grâce au compteur évoqué à la section 3.1 un des CPU virtuels interrompra son exécution régulièrement pour évaluer la situation et effectuer des modifications. Ce système permet de ne retirer qu'un minimum de ressources au système d'exploitation. Le temps entre chaque réveil de Scalevisor pourra être modifié dynamiquement en fonction des besoins, fréquents ou non, de modifier le placement.

4.2. Politiques sur mesure

L'heuristique proposé à la section précédente constituera la base de notre système et donc la politique appliquée par défaut. Cependant, pour résoudre les problèmes de performances liés aux comportements spécifiques de certaines applications, nous proposons de mettre en œuvre une interface permettant aux applications d'informer directement Scalevisor de leur comportement.
Ce fonctionnement possède deux principaux avantages :

- Les coûts de migration seront diminués puisque le placement sera meilleur en premier lieu et nécessitera donc moins de modifications futures, il en va de même pour le coût de l'échantillonnage dont la fréquence pourra être diminuée pour cette application donnée.
- Il est raisonnable de penser qu'une application fournissant des informations sur son fonctionnement nous permettra d'obtenir des informations plus fines et plus justes que l'échantillonnage.

Contrairement à la politique par défaut qui s'interrompt de façon régulière pour évaluer le

placement et le modifier au besoin, dans ce cas, l'application aura à sa disposition des voies de communication directes permettant de notifier Scalevisor lors d'allocations mémoire pour donner des indications sur la façon dont cette mémoire sera utilisée, et par quelles tâches.

5. Discussion

5.1. Le coût de la virtualisation

Utiliser la virtualisation permet de fournir suffisamment d'abstraction pour pouvoir mettre en œuvre de nouvelles techniques de placement indépendemment du système d'exploitation. Cependant, malgré les avancées récentes dans le domaine, cette technologie est une source de surcoûts potentiels lors de l'exécution.

Nous pensons que ce surcoût sera en grande partie absorbé par le gain important de performances que permettra le placement efficace sur des machines NUMA. Par ailleurs nous nous affranchissons de plusieurs sources de pertes de performances classiques en raison de l'architecture de notre projet.

Premièrement, puisque nous n'exécutons qu'un seul système invité nous ne paierons aucun surcoût dû à l'ordonnancement de plusieurs invités. De plus, le coût du réveil de Scalevisor, évoqué aux sections 3.1 et 4.2, bien que non négligeable est faible. En effet, il suffira alors de consulter les informations fournies par les différentes techniques d'échantillonnage et de prendre une décision. Dans le cas, idéal, où le système d'exploitation est stable, n'effectue quasiment aucune allocation mémoire et aucun déplacement de tâches, nous ne déplacerons aucune ressource non plus.

Deuxièmement, la quasi-totalité des entrées/sorties seront faites directement par l'unique invité, évitant ainsi de multiplexer ces accès au niveau logiciel. Enfin, lors de la présence de plusieurs systèmes invités, l'hyperviseur doit intercepter et router les différentes interruptions vers les machines virtuelles auxquelles celles-ci sont destinées, un problème que nous n'aurons pas à traiter dans le cadre d'un seul système invité.

5.2. Le coût de l'échantillonnage

Les diverses techniques d'échantillonnage possèdent également un coût, il s'agira par la suite de trouver un bon rapport entre la quantité d'information nécessaire pour prendre des décisions de placement efficaces et le surcoût induit. Il est par ailleurs possible d'utiliser une fréquence d'échantillonnage adaptative selon l'évolution du système.

6. Travaux connexes

Les nouvelles contraintes liées aux machines NUMA ont déjà donné lieu à de nombreux travaux. Ces travaux montrent tous le fort impact de ces architectures sur les performances et proposent des solutions pour améliorer ces performances.

Certains proposent des heuristiques de placement mono-ressources. C'est le cas de Voron et al. [6] qui décrivent une politique de placement dynamique utilisant les compteurs de performances du processeur pour augmenter la localité de la mémoire. De même, Rao et al. [5] présentent une heuristique de placement des vCPUs basée sur les latences d'accès aux composants externes au cœur (cache L3 par exemple) qui consiste à minimiser les coûts de ces accès. Nous avons en commun avec ces travaux l'approche dynamique basée sur les compteurs matériels, cependant, nous pensons qu'il est nécessaire de prendre en compte plusieurs types de ressources. Par ailleurs ces deux travaux ne proposent qu'une seule politique qui ne peut donc pas être efficace pour toutes les applications. Nous comptons résoudre ce problème grâce aux

indices fournis par les applications.
Bugnion et al. [1] présentent également une heuristique mono-ressource qui tend à augmenter la localité des accès mémoires par des migrations ou des duplications de mémoire selon le type d'accès (lecture ou écriture). De même, cette approche ne permet pas de s'adapter à diverses applications.
D'autres travaux présentent des heuristiques multi-ressources comme Wu et al. [7] qui prennent en compte simultanément le placement des vCPUs et de la mémoire. Cependant, la politique de placement mise en œuvre est très généraliste puisqu'elle consiste à répartir équitablement entre les nœuds les vCPU exécutant les tâches les plus intensives pour la mémoire et à voler un vCPU lorsqu'un pCPU est inoccupé.

7. Conclusion

Ce papier propose d'améliorer la gestion des ressources sur des architectures multicœurs complexes grâce à un pilote pour les processeurs et la mémoire. Ce pilote emploie des technologies de virtualisation récentes pour agir de façon transparente du point de vue du système d'exploitation. Nous réalisons la mise en œuvre du design proposé, et travaillons actuellement sur la virtualisation d'un système utilisant plusieurs processeurs.

Bibliographie

1. Bugnion (E.), Devine (S.) et Rosenblum (M.). – DISCO : running commodity operating systems on scalable multiprocessors. – In *SOSP*, pp. 143–156. ACM, 1997.
2. Dashti (M.), Fedorova (A.), Funston (J. R.), Gaud (F.), Lachaize (R.), Lepers (B.), Quéma (V.) et Roth (M.). – Traffic management : a holistic approach to memory placement on NUMA systems. – In *ASPLOS*, pp. 381–394. ACM, 2013.
3. Lachaize (R.), Lepers (B.) et Quéma (V.). – Memprof : A memory profiler for NUMA multicore systems. – In *USENIX Annual Technical Conference*, pp. 53–64. USENIX Association, 2012.
4. Lozi (J.), Lepers (B.), Funston (J. R.), Gaud (F.), Quéma (V.) et Fedorova (A.). – The linux scheduler : a decade of wasted cores. – In *EuroSys*, pp. 1 :1–1 :16. ACM, 2016.
5. Rao (J.), Wang (K.), Zhou (X.) et Xu (C.). – Optimizing virtual machine scheduling in NUMA multicore systems. – In *HPCA*, pp. 306–317. IEEE Computer Society, 2013.
6. Voron (G.), Thomas (G.), Quéma (V.) et Sens (P.). – An interface to implement NUMA policies in the xen hypervisor. – In *EuroSys*, pp. 453–467. ACM, 2017.
7. Wu (S.), Sun (H.), Zhou (L.), Gan (Q.) et Jin (H.). – vprobe : Scheduling virtual machines on NUMA systems. – In *CLUSTER*, pp. 70–79. IEEE Computer Society, 2016.