RenPar'20 / SympA'14 / **CFSE 8**
Saint-Malo, France, du 10 au 13 mai 2011

Optimisations applicatives pour multi-cœurs NUMA : un cas d'étude avec le serveur web Apache

Fabien Gaud[1], Renaud Lachaize[2], Baptiste Lepers[3], Gilles Muller[1], Vivien Quéma[3]

[1]INRIA, [2]Université de Grenoble, [3]CNRS,
{prenom.nom}@inria.fr

Résumé
Les machines multi-cœurs à accès mémoire non uniforme (NUMA) sont de plus en plus répandues. Il est donc nécessaire de comprendre comment les exploiter efficacement. La plupart des travaux menés dans ce domaine choisissent de résoudre ce problème au niveau du noyau du système d'exploitation, en améliorant les abstractions fournies aux applications ou en proposant de nouvelles architectures logicielles pour améliorer le passage à l'échelle des applications sur ce type de machines.
Dans ce papier, nous adoptons un point de vue complémentaire : nous étudions comment améliorer le couple d'applications Apache-PHP au dessus d'un noyau Linux standard. Nous mettons en lumière trois différents problèmes de performance à différents niveaux du système : (i) un nombre d'accès mémoire distants trop important, (ii) un équilibrage de charge inefficace entre les processeurs et (iii) de la contention sur des structures de données du noyau. Nous proposons et implantons des solutions au niveau applicatif pour chacun de ces problèmes. Notre version optimisée des applications Apache et PHP a un débit 33 % plus important que ces applications non modifiées sur une machine à 16 cœurs. Nous concluons en explicitant les leçons tirées lors de ces travaux sur l'amélioration de la performance des serveurs sur les architectures multi-cœurs NUMA.

Mots-clés : serveur, performances, parallélisme, passage à l'échelle, architecture mémoire

1. Introduction

Les machines multi-cœurs à accès mémoire non uniforme (NUMA) deviennent la norme et l'exploitation efficace de leurs ressources est toujours un domaine de recherche ouvert. Cette question a été principalement abordée au niveau du système d'exploitation. Des architectures novatrices pour les environnements d'exécution [22, 18, 23] et pour les noyaux de systèmes d'exploitation [3, 2] ont notamment été proposées. Des travaux récents sur l'adaptation de l'architecture des systèmes d'exploitation montrent des résultats prometteurs [4].
Ce papier adopte un point de vue complémentaire. Nous examinons ce qui peut être fait au niveau applicatif pour améliorer le passage à l'échelle des applications parallèles au dessus des systèmes d'exploitation actuels. Notre approche est motivée par plusieurs facteurs. Premièrement, cela permet aux applications de surmonter rapidement les problèmes de passage à l'échelle des noyaux courants, tout en attendant des corrections de ceux-ci à plus long terme. Deuxièmement, tous les problèmes de passage à l'échelle de l'application ne peuvent pas être entièrement résolus sans intervention du développeur de celle-ci. Dans tous les cas, une connaissance approfondie du comportement de l'application est nécessaire pour le développement de politiques efficaces au niveau du système d'exploitation.
Nous choisissons d'étudier un serveur Web, pour lequel une exploitation efficace des architectures multi-cœurs est cruciale. Plus précisément, nous étudions la performance du couple Apache-PHP [1, 17] sur une machine 16 cœurs NUMA (4 processeurs) avec le banc de test reconnu SPECweb2005 [20]. Nous avons choisi ces deux applications car elles sont les plus utilisées actuellement dans leur domaine [1], sont

1. Une étude récente [12] montre qu'Apache représente 58 % des serveurs Web sur Internet.

complexes et sont connues pour leurs bonnes performances. Sur cette configuration, nous observons une baisse de 26 % de la performance lorsque le nombre de processeurs utilisés passe de 1 à 4. Dans ces travaux, nous identifions différentes causes d'inefficacité et proposons un ensemble d'optimisations pour les réduire. Nos solutions s'étalent sur plusieurs dimensions : configuration de déploiement, stratégie d'équilibrage de charge et contournement des problèmes de passage à l'échelle du noyau. Grâce à ces solutions, nous améliorons les performances de 33 % par rapport à la configuration initiale du serveur et arrivons à seulement 5 % du passage à l'échelle idéal.
Ce papier apporte les contributions suivantes : (i) il fournit des réponses sur les problèmes de passage à l'échelle d'un serveur populaire sur une architecture multi-cœur NUMA avec une charge réaliste. (ii) il décrit des stratégies au niveau applicatif pour résoudre ces problèmes de performances (iii) il décrit la méthodologie qui nous a permis de localiser précisément ces problèmes non triviaux et en tire un ensemble de leçons à partir de notre expérience.
Le reste du papier est organisé comme suit. La section 2 présente l'état de l'art. La section 3 fournit des informations sur le matériel et sur les applications utilisés. La section 4 décrit notre plate-forme de test ainsi que les performances de référence. Dans les sections 5, 6 et 7, nous analysons les problèmes de performances, ainsi que leurs solutions. La section 8 résume les leçons méthodologiques retenues de ce travail. Finalement, la section 9 conclut ce papier.

2. État de l'art

De nombreux projets de recherche ont eu pour objectif de rendre les systèmes d'exploitation plus adaptés pour les architectures multi-cœurs. L'idée principale consiste à minimiser la contention sur les ressources matérielles et logicielles en éliminant le partage inutile entre les cœurs. K42 [10] repense les structures internes pour améliorer le passage à l'échelle. Corey [3] fournit aux programmeurs d'applications la possibilité de contrôler finement le partage des données ainsi que le placement des services du noyau. Le modèle *multi-kernel*, mis en œuvre dans le système d'exploitation Barrelfish [2], va plus loin dans cette approche en architecturant le système d'exploitation comme un système distribué. Durant les dix dernières années, de nombreuses améliorations de passage à l'échelle ont également été introduites dans des noyaux plus communs, tels que Linux (e.g. [5]). Dans une analyse très récente [4], les chercheurs du MIT ont étudié le passage à l'échelle du noyau Linux sur une architecture massivement multi-cœur et ont conclu que, sur les matériels actuels, les architectures de systèmes d'exploitation traditionnelles ne présentent pas d'obstacles majeurs pour le passage à l'échelle.
Un ensemble de travaux a considéré l'étude et l'amélioration de la performance des serveurs de données (e.g. [11, 15]) mais seule une faible partie d'entre eux ont pris en compte les spécificités des architectures multi-cœurs. Choi et al. [7] comparent la performance de différentes architectures de serveurs sur une architecture parallèle. Leur étude utilise plusieurs traces réelles, mais est seulement effectuée sur un simulateur avec un matériel et un modèle applicatif simples. De plus, ils choisissent d'étudier des charges statiques provoquant un nombre important d'E/S disque. Pour notre part, nous choisissons d'étudier le comportement du serveur Web sur une architecture réelle, avec une distribution de fichiers tenant en mémoire et avec une partie des requêtes concernant des fichiers générés dynamiquement. Nos travaux sont proches de l'étude menée par Veal et Foong [21], qui mesure le passage à l'échelle de la pile logicielle Linux-Apache-PHP sur une machine Intel à 8 cœurs. Ils concluent que le bus d'adresses est le principal obstacle au passage à l'échelle des performances et masque les problèmes logiciels. Toutefois, notre configuration matérielle est sensiblement différente avec deux fois plus de cœurs, une architecture NUMA et une capacité réseau sensiblement augmentée. De plus, nous fournissons des solutions aux problèmes de performances rencontrés.
Finalement, remarquons que les auteurs de Corey et de Barrelfish ont utilisé des serveurs simples pour leurs tests. Apache est également l'une des applications considérées par les travaux du MIT sur PK [4] et DProf [16]. Dans ces quatre cas, le serveur a été testé avec une charge simpliste, car l'objectif de l'expérimentation était de stresser le système d'E/S du noyau. Notre étude fournit un point de vue différent, en se concentrant sur l'impact combiné (i) des interactions du serveur avec un système complexe tel que PHP et (ii) d'une charge plus réaliste.

3. Contexte

Dans cette section, nous présentons l'architecture matérielle que nous avons utilisée pour l'évaluation de performances. Nous décrivons ensuite l'architecture du serveur Web Apache et expliquons comment le traitement d'une requête s'effectue sur notre architecture matérielle.

3.1. Architecture matérielle étudiée

Les mesures ont été effectuées sur une machine Dell PowerEdge R905. L'architecture de cette machine est présentée dans la figure 1. Elle est équipée de quatre processeurs AMD Opteron 8380 disposant chacun de quatre cœurs, soit un total de 16 cœurs. Chaque cœur a une fréquence de 2.5 GHz, un cache L1 et L2 privé et un cache L3 partagé avec les trois autres cœurs du processeur. La machine est équipée de 32 Go de mémoire découpée en 4 nœuds, soit 8 Go par processeur. Nous avons mesuré un débit maximum de 61 Gb/s entre le contrôleur mémoire (MCT) et la mémoire locale (DRAM). Les processeurs communiquent (pour les E/S, les requêtes mémoire et de cohérence de caches) grâce aux liens HyperTransport 1.0, avec un débit maximum mesuré de 24 Gb/s. Les messages sont routés statiquement entre les processeurs (par exemple, les messages envoyés du processeur 0 au processeur 3 transitent toujours par le processeur 1) grâce au *crossbar*. Les accès mémoire sont effectués par le contrôleur mémoire, qui est également responsable du mécanisme de cohérence de cache. Finalement, la machine a 20 cartes réseau Ethernet 1 Gb/s (Intel Pro/1000 PT), connectées soit au nœud 0 soit au nœud 3.

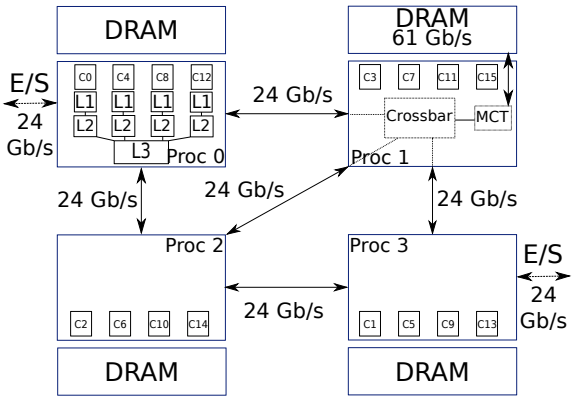

FIGURE 1 – Architecture du serveur évalué.

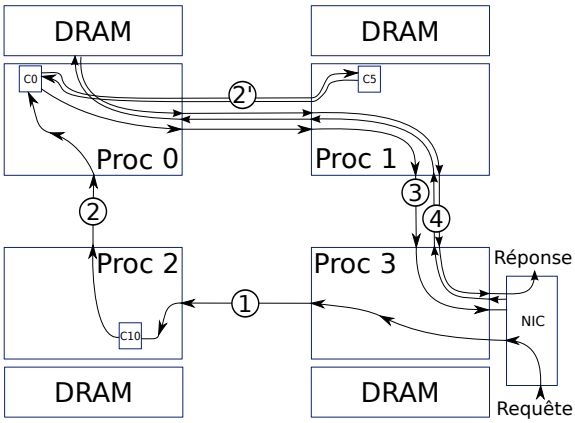

FIGURE 2 – Étapes nécessaires au traitement d'une requête dynamique.

Notre architecture n'est pas complètement symétrique en terme de connexions mémoires. Les processeurs 0 et 3 sont plus proches des périphériques d'E/S mais ont une connexion moins directe vers certains bancs mémoire. Plus précisément, les processeurs 1 et 2 ont accès à la mémoire au maximum en un saut (*processeurs rapides*) alors que les processeurs 0 et 3 accèdent à la mémoire en au plus deux sauts (*processeurs lents*).

3.2. Architecture des serveurs Apache et PHP

Apache peut utiliser trois modes différents d'exécution : *Worker*, *Prefork* et *Event*. Dans nos expériences, nous utilisons Apache dans sa configuration la plus répandue : *Worker*. Cette architecture utilise à la fois des processus et des threads. Tous les processus se comportent de la même façon, exception faite du premier processus créé (*maître*). Chaque processus déploie un thread pour l'acceptation des nouvelles connexions et maintient un ensemble de threads pour servir les requêtes entrantes. Le processus maître est responsable de la création et de la destruction des processus en fonction de la charge sur le serveur.

Les threads traitent les requêtes statiques et dynamiques de manière différente. Les requêtes statiques sont servies en utilisant l'appel système `sendfile`, qui permet de transmettre un fichier sur le réseau

sans copies. Les requêtes dynamiques sont traitées par un nombre fixe de processus PHP [2] démarrés à la création du serveur. Les processus PHP interagissent avec Apache en utilisant le protocole FastCGI.
La figure 2 donne un exemple de traitement d'une requête dynamique sur notre architecture [3]. Le paquet TCP entrant est traité par le système d'exploitation sur un cœur, ce qui peut nécessiter de transiter par un ou plusieurs liens HyperTransport (étape 1). Une fois traitée par le noyau, la requête HTTP est prise en charge par un thread Apache s'exécutant potentiellement sur un processeur différent (étape 2). Le traitement d'une requête dynamique nécessite ensuite une interaction avec un processus PHP éventuellement placé sur un autre processeur (étape 2'). L'étape 3 correspond à l'appel système `write` qui lance une requête DMA, effectuée à l'étape 4. Notons que pour les requêtes statiques, l'interaction est identique à l'exception de l'étape 2' et de l'utilisation de `sendfile` à la place de `write`. Dans ce cas, le placement des fichiers en mémoire conditionne la localité des accès mémoire effectués par la carte réseau.

4. Configuration de test et performances de référence

Dans cette section, nous décrivons l'injection de charge SPECweb2005. Nous présentons ensuite notre configuration expérimentale et évaluons enfin le passage à l'échelle de cette configuration.

4.1. Injection utilisée

Notre évaluation de performance est basée sur le banc d'essai SPECweb2005 qui fourni un module d'injection en boucle fermée ainsi que des scripts et des fichiers permettant de modéliser un site Internet réaliste. SPECweb2005 inclue trois types de charge : *Support*, *E-commerce* et *Banking*. Nous avons choisi d'étudier la charge *Support* pour plusieurs raisons. Premièrement, cette charge passe plus de temps dans le serveur Web que les autres charges (les autres passent plus de temps, par exemple, dans les bibliothèques de chiffrement). Ainsi, elle est la plus à même de soulever les problèmes de passage à l'échelle du serveur. Deuxièmement, cette charge nécessite moins de traitement CPU et plus d'E/S que les autres, ce qui rend l'utilisation efficace des multi-cœurs moins aisée car le noyau est plus sollicité.
Pour éviter les E/S disque, nous avons limité la taille de la distribution de fichiers à 12 Go (via l'option `dirscaling`). Ce choix est en phase avec les tendances actuelles dans les fermes de serveurs [14] et a également été effectué par Veal et Foong dans leur étude [21].
Nous utilisons la métrique de performance définie par SPECweb2005. Cette métrique considère le nombre de clients pouvant être traités par seconde, tout en garantissant une qualité de service (QoS) définie : 95 % des requêtes doivent être traitées en moins de 3 secondes (*bonnes* requêtes), 99 % des requêtes en moins de 5 secondes (requêtes *tolérables*), moins de 0.5 % d'erreurs et un débit minimum pour les pages statiques.

4.2. Configuration matérielle

Nous exécutons le serveur Web Apache sur la machine PowerEdge R905 décrite précédemment dans la section 3. Pour les expériences nécessitant moins de 16 cœurs, nous désactivons, grâce aux mécanismes du noyau, les cœurs inutilisés. L'injection de charge est effectuée grâce à 22 machines double cœurs équipées de 2 Go de mémoire et d'une carte réseau 1 Gb/s. Ces machines sont interconnectées grâce à un commutateur 48 ports non bloquant à 1 Gb/s. Chaque machine exécute deux instances du module SPECweb2005, ce qui permet de simuler jusqu'à 2000 clients par machine, et peut envoyer des requêtes sur chacune des interfaces du serveur. SPECweb2005 utilise un module pour simuler une base de données (BeSim). Tout comme Veal et Foong [21], nous exécutons une instance du BeSim par client, pour éviter les goulots d'étranglement sur ce module.

4.3. Configuration logicielle

Système d'exploitation. Nous utilisons le noyau Linux 2.6.34 avec le support des architectures NUMA. Nous avons soigneusement réglé les paramètres du noyau pour assurer que le réseau n'est jamais un

2. Nous avons observé que le gestionnaire de processus PHP diminue de manière sensible les performances du serveur Web. Par conséquent, nous utilisons une configuration statique pour PHP.

3. Notons que les interactions décrites supposent que les fichiers sont présents en mémoire et ne provoquent ainsi pas d'E/S disque.

goulot d'étranglement des performances. Plus précisément, nous avons configuré les IRQs de manière à ce que chaque processeur gère 5 cartes réseau et avons fixé soigneusement différents paramètres TCP, tels que ceux décrits par Veal et Foong [21]. Nous avons également augmenté sensiblement le nombre maximum de processus pouvant être créés sur le système de 32768 à 262144.

Apache/PHP. Nous utilisons la version 2.2.14 du serveur Apache et la version 5.2.12 de PHP. Tel que mentionné précédemment, tous les fichiers sont préchargés en mémoire. Pour équilibrer la charge entre les nœuds mémoire, nous répartissons la distribution de fichiers sur les quatre bancs mémoire.
Nous avons soigneusement réglé les paramètres des serveurs Apache et PHP pour obtenir la meilleure performance possible. Nous avons notamment augmenté le nombre de threads par processus à 64 et utilisons `sendfile` pour effectuer des envois de fichiers sans copie. Comme expliqué précédemment, nous fixons statiquement le nombre de processus PHP à 500. Finalement, nous utilisons eAccelerator [8], un cache de *byte-code* qui permet de ne compiler les scripts PHP qu'une seule fois et améliore ainsi très sensiblement les performances du serveur.

4.4. Performances de référence

Nous avons évalué la pile logicielle Linux-Apache-PHP configurée comme décrite précédemment. Cette configuration est appelée *Apache non modifié* dans le reste de ce papier. Comme unité de passage à l'échelle, nous considérons le processeur et non pas le cœur. La raison est que, comme expliqué dans [4], utiliser une configuration monocœur comme référentiel de performance est trompeur. En effet, les cœurs situés sur le même processeur partagent un cache, ce qui empêche le système d'atteindre un passage à l'échelle optimal (nous observons que le taux de fautes de cache L3 passe de 8 à 14 % entre 1 et 4 cœurs).
Les résultats obtenus pour la version non modifiée d'Apache sont présentés dans le tableau 1. Lorsqu'un processeur est utilisé, tous les cœurs de ce processeur sont actifs. Notons que, comme les processeurs ne sont pas symétriques, lorsque moins de 4 processeurs sont utilisés, nous présentons la moyenne des résultats obtenus sur les différentes combinaisons. Le tableau 1 montre que lorsque 4 processeurs sont utilisés, le nombre de clients pouvant être traités par un processeur (6600 clients) est 26 % moins important que lorsqu'un unique processeur est utilisé (8900 clients).

		1 processeur		*2 processeurs*		*4 processeurs*	
	Configuration	**Nb clients**	**Gain (%)**	**Nb clients**	**Gain (%)**	**Nb clients**	**Gain (%)**
1	Apache sans modif.	8900	N/A	7900	N/A	6600	N/A
2	Apache + modif. 1	8900	0	7700	-2,5	7200	9,1
3	Apache + modif 1 et 2	8900	0	8100	2,5	8000	21,2
4	Apache + modif 1, 2 et 3	9300	4,5	8800	11,4	8800	33,3

TABLE 1 – Résultat de performance des différentes configurations d'Apache avec une charge SPECweb2005 et gain de performance par rapport à la version non modifiée d'Apache à 1, 2 et 4 processeurs.

Dans les sections suivantes, nous localisons trois problèmes de performance, expliquant la baisse de débit entre 1 et 4 processeurs, et présentons une solution pour chacun d'eux. Nous considérons premièrement une baisse dans l'efficacité des accès mémoire et réduisons ce problème en colocalisant les processus communicants (Section 5). Ensuite, nous améliorons l'équilibrage de charge en prenant en compte l'asymétrie de notre architecture NUMA (Section 6). Enfin, nous identifions et contournons des problèmes de passage à l'échelle dans le noyau Linux (Section 7).

5. Première optimisation : réduction du nombre d'accès distants

Dans cette section, nous présentons notre première optimisation pour améliorer la performance du serveur Apache sur les architectures NUMA. Nous expliquons comment nous avons diagnostiqué une première source de baisse de performance que sont les accès mémoire distants. Nous introduisons ensuite une solution à ce problème et montrons qu'elle améliore les performances à 4 processeurs de 9 %.

5.1. Analyse du problème

Le problème de passage à l'échelle observé dans la section précédente ne vient pas d'un problème d'injection de charge : les processeurs sont complètement chargés, les liens réseau et les injecteurs ne sont pas saturés. La première métrique que nous regardons est le nombre d'instructions exécutées par cycle (IPC). Cette métrique permet de mettre en lumière la performance des accès mémoires. Le tableau 2 présente l'IPC moyen sur chacun des cœurs. Ce tableau montre que l'IPC baisse sensiblement à 4 processeurs (-21 %).

La baisse d'IPC observée peut être provoquée par trois principales causes : (i) une baisse de l'efficacité des caches due à du vrai ou du faux partage entre les processeurs, (ii) un nombre plus important d'accès mémoire distants par octet transmis et (iii) une pression plus importante sur les liens HyperTransport ou sur les bus d'accès mémoire locaux qui provoque une augmentation des latences d'accès mémoire. Nous étudions chacune de ces possibilités dans la suite de cette section.

Efficacité des caches. Nous avons mesuré le nombre de fautes de cache L1, L2 et L3 par octet transmis ainsi que leur taux de fautes de cache. Nous n'avons pas observé une réduction dans l'efficacité des caches : le nombre de fautes de cache par octet transmis reste constant entre 1 et 4 processeurs, ainsi que le taux de fautes de cache. Par conséquent, la baisse d'IPC n'est pas attribuable à une baisse de l'efficacité des caches.

Accès mémoire distants. Nous avons mesuré le ratio entre le nombre d'accès mémoire total et le nombre d'accès mémoire distants ainsi que ce dernier par octet transmis. Les résultats sont présentés dans le tableau 2. Nous observons que le nombre d'accès distants par octet transmis augmente très sensiblement à 4 processeurs (+250 %). De plus, à 4 processeurs, le nombre d'accès mémoire distants est plus important que le nombre d'accès mémoire locaux (67 % des accès mémoire sont distants). Nous concluons de ces observations que l'augmentation des accès mémoire distants est une cause possible de la baisse de l'IPC.

Utilisation des liens HyperTransport et des bus mémoire. Nous avons mesuré le taux d'utilisation des liens HT. Le taux maximum d'utilisation des liens HT est présenté dans le tableau 2. Nous pouvons observer qu'aucun bus mémoire n'est saturé, mais que le taux d'utilisation augmente très sensiblement entre 1 et 4 processeurs (+200 %). Ceci peut être une des raisons de la baisse observée de l'IPC à 4 processeurs.

Il n'est pas possible sur notre machine de déterminer précisément le taux d'utilisation des bus mémoire locaux. Toutefois, nous pensons que ces bus ne sont pas la source principale de baisse de l'IPC, car ils ont une capacité plus importante que les liens HT et la majeure partie (67 %) des accès mémoire transite par liens HT.

Configuration	IPC moyen	Accès distants (%)	Accès distants/Ko	Taux Max. HT (%)
1 processeur	0,38	21	4	25
4 processeurs	0,30	67	14	75

TABLE 2 – **Apache non modifié :** IPC moyen, ratio d'accès mémoire distant, nombre d'accès mémoire distants et taux maximum d'utilisation des liens HyperTransport.

5.2. Solution

Dans la section 5.1, nous avons identifié deux raisons possibles pour la baisse de l'IPC à 4 processeurs : une augmentation importante du nombre d'accès mémoire distants ainsi qu'une augmentation du taux d'utilisation des liens HT. Les causes des accès mémoire distants sont multiples. Premièrement, ils résultent du partage de ressources (par exemple, les *pipes* de communication) au sein des processus Apache. Deuxièmement, ils sont provoqués par les communications entre les différents composants du serveur Web (illustrés par la figure 2). Troisièmement, ils proviennent des interactions entre l'application et les services du noyau, notamment la pile réseau.

Une première solution pour limiter le nombre d'accès mémoire distants est de forcer Apache à ne communiquer qu'avec les processus PHP situés sur le même processeur. Toutefois, cette approche ne permet de résoudre qu'une partie du problème : la localisation sur un même processeur des communications entre les composants du serveur Web. Par conséquent, nous avons choisi une autre solution qui consiste à déployer une instance du serveur par processeur, plutôt qu'une unique instance pour tous les processeurs. Les processus Apache d'une instance, leurs threads ainsi que les processus PHP associés sont fixés sur le même processeur. Pour cela, nous utilisons les possibilités offertes par Linux (i.e. `taskset`) sans modifier le serveur Web. Chaque instance d'Apache gère 5 cartes réseau, de façon à colocaliser le traitement de la pile réseau avec le traitement de la requête HTTP.
Grâce à cette solution, les étapes 2 et 2' (présentées sur la figure 2) sont maintenant effectuées sur le même processeur. Les seuls accès distants possibles sont ceux effectués par les DMA des cartes réseau (étapes 1, 3 et 4).

5.3. Évaluation

Le tableau 1 présente les résultats obtenus avec la solution présentée précédemment (deuxième ligne). Cette optimisation augmente les performances du serveur Web de 9 % à 4 processeurs par rapport à la version non modifiée. Cette optimisation ne change pas les performances à 1 processeur. De manière plus étonnante, les performances à 2 processeurs sont légèrement réduites (-2.5 %). La raison est donnée dans la section 6.
Nous avons vérifié l'impact de cette optimisation sur les métriques décrites dans la section 5.1. Les résultats sont présentés dans le tableau 3. Comme attendu, l'IPC à 1 processeur n'est pas modifié. Dans la configuration à 4 processeurs, nous observons une augmentation importante d'IPC (+19 %) par rapport à la version non modifiée. Cette augmentation est principalement due à une baisse sensible du nombre d'accès à la mémoire distante (-64 %). Grâce à cette solution, la plupart des accès sont faits localement (73%). En ce qui concerne le taux d'utilisation des liens HT, le gain est moins significatif (-13%). La raison est que le trafic sur ces liens n'est pas uniquement dû aux accès mémoire distants, mais également aux accès effectués par le DMA ainsi qu'à la cohérence de cache.

Configuration	IPC moyen	Accès distants (%)	Accès distants/Ko	Taux Max. HT (%)
1 processeur	0,38	21	4	25
4 processeurs	0,36	27	5	65

TABLE 3 – **Apache + optim. 1 :** IPC moyen, ratio d'accès mémoire distants, nombre d'accès mémoire distants et taux maximum d'utilisation des liens HyperTransport.

6. Deuxième optimisation : répartition équitable de la charge

Dans cette section, nous présentons notre deuxième optimisation visant à améliorer les performances d'Apache. Nous commençons par diagnostiquer le problème : la charge n'est pas équitablement répartie entre les processeurs, et les processeurs rapides sont inactifs 15% du temps. Nous proposons ensuite une solution à ce problème et montrons qu'Apache, modifié avec nos deux optimisations, est 21.2% plus efficace à 4 processeurs qu'un Apache non modifié.

6.1. Diagnostic

Les résultats présentés dans le tableau 1 montrent qu'Apache, avec notre première optimisation, est toujours 19% moins efficace à 4 processeurs qu'à 1 processeur. Comme nous l'avons fait dans la section 5, nous commençons par vérifier que les CPUs ne sont pas inactifs, que le réseau n'est pas saturé et que les injecteurs ne sont pas surchargés. Nous observons que les processeurs ne travaillent pas tous au maximum de leur capacité : les processeurs 1 et 2 sont inactifs 15% du temps, alors qu'ils sont tous utilisés à 100% de leur capacité avec une configuration d'Apache non modifiée.
Pour comprendre les causes d'inactivité sur certains processeurs, rappelons que la première optimisation oblige tous les processus chargés de traiter une requête à être localisés sur un même processeur. A

contrario, avec une version non modifiée d'Apache, tous les processeurs peuvent participer au traitement d'une requête. L'ordonnanceur de Linux peut également équilibrer la charge sur tous les processeurs. Ce n'est pas le cas avec notre première optimisation, puisque tous les processus impliqués dans le traitement d'une requête sont placés sur le processeur en charge de la carte réseau ayant reçu la requête. Obliger les requêtes à être traitées sur un unique processeur induit de l'inactivité sur certains processeurs si la charge n'est pas répartie de manière adéquate. Comme nous l'avons vu en section 4, SPECweb2005 utilise une injection en boucle fermée. Chaque client choisit une nouvelle carte réseau, en *round-robin*, à chaque fois qu'il se connecte au serveur. Cette méthode fonctionne parfaitement si tous les processeurs traitent les requêtes à la même vitesse. Si un processeur est plus lent que les autres, le débit va être asservi à la vitesse de ce processeur [4].

Comme nous l'avons expliqué dans la section 3, nous avons une différence de performance entre nos processeurs due à l'interconnexion entre ceux-ci. Nous avons vérifié que certains de nos processeurs sont effectivement plus lents que d'autres. Les résultats SPECweb2005 sont présentés dans le tableau 4. Ces résultats montrent qu'avec 6800 clients, les processeurs lents et rapides se comportent de manière similaire. Cependant, plus la charge augmente, plus le ratio de requêtes traitées en un temps bon ou tolérable sur les processeurs lents diminue. En haute charge, les processeurs lents sont surchargés et les requêtes attendent dans les files d'attente du serveur, ce qui augmente la latence perçue par les clients. Le ratio des processeurs rapides, au contraire, reste constant. En effet, la charge est injectée à une vitesse constante, asservie au débit des processeurs lents. La ligne grise représente le nombre maximum de clients qui peuvent être traités tout en respectant la QoS de SPECweb2005.

	Processeurs lents			*Processeurs rapides*		
Nb clients	**Bonnes (%)**	**Tolérables (%)**	**Processeur (%)**	**Bonnes (%)**	**Tolérables (%)**	**Processeur (%)**
6800	99,9	100	95	99,9	100	82
7200	99,6	100	100	99,9	100	85
8000	15,4	90	100	99,9	100	85
8800	8,7	8,5	100	99,9	100	85

TABLE 4 – **Apache + Optim. 1 :** résultats SPECweb2005 et utilisation du processeur avec une injection round-robin sur les processeurs lents et rapide (configuration 16 processeurs).

Une approche simple, pour résoudre ce problème consiste à remplacer le choix des cartes réseau : passer d'une sélection en roud-robin, à une sélection fixe (i.e. chaque client virtuel injecte sur une carte donnée pendant toute la durée du test). Les résultats obtenus avec cette approche sont présentés dans le tableau 5. Le nombre maximum de clients traités avec cette méthode est toujours égal à 7200 (ligne grise). De manière intéressante, nous pouvons observer que la charge n'est plus asservie à la vitesse des processeurs lents : les processeurs rapides sont de plus en plus chargés. Cependant, cette solution n'est pas satisfaisante parce qu'elle répartit la charge de manière homogène sur tous les processeurs, sans tenir compte de leur puissance.

	Processeurs lents			*Processeurs rapides*		
Nb clients	**Bonnes (%)**	**Tolérables (%)**	**Processeur (%)**	**Bonnes (%)**	**Tolérables (%)**	**Processeur (%)**
6800	100	100	92	100	100	85
7200	99,9	100	99	100	100	95
8000	69,6	99	100	99,7	100	100
8800	24,4	85,2	100	53,4	97,8	100

TABLE 5 – **Apache + Optim 1 :** résultats SPECweb2005 et utilisation du processeur avec une injection fixe sur les processeurs lents et rapides (configuration 16 processeurs).

4. C'est une conséquence de l'injection en boucle fermée : les clients attendent d'avoir reçu une réponse avant d'envoyer une nouvelle requête.

6.2. Solution

La majorité des infrastructures Web utilisent un ou plusieurs mécanismes d'équilibrage de charge pour répartir les requêtes entre plusieurs serveurs [6]. Nous avons décidé de tirer parti de ce mécanisme pour répartir la charge entre les processeurs d'une même machine. Comme chaque instance d'Apache est responsable d'un ensemble de cartes réseau, le mécanisme d'équilibrage de charge considère chaque ensemble comme une machine différente. Il répartit donc la charge entre les processeurs comme il le ferait avec des machines physiques différentes.

Comme nous n'avons pas d'autre machine capable de traiter 20Gb/s, nous avons implanté le mécanisme d'équilibrage de charge sur les clients. Cette implantation est similaire à un routeur IP prenant en compte la charge des serveurs [6] : le choix du serveur n'est basé que sur la charge estimée des machines, pas sur le contenu des requêtes. Nous estimons la charge des processeurs en comptant le nombre de requêtes en attente sur chaque ensemble de cartes réseau.

6.3. Évaluation

Les résultats obtenus avec cette solution sont présentés dans le tableau 6. Le nombre maximal de clients servis est maintenant égal à 8000 (ligne grise) et les processeurs lents et rapides sont pleinement chargés. Notre mécanisme d'équilibrage de charge est donc efficace. Les résultats à 1, 2 et 4 processeurs sont présentés dans le tableau 1 (troisième ligne). Nous pouvons observer que, à 4 processeurs, cette optimisation améliore les performances de 11% par rapport à la première optimisation seule et de 21% par rapport à Apache non modifié. Nous pouvons également remarquer que l'absence d'équilibrage de charge était la cause de la légère baisse de performance à 2 processeurs observée avec l'optimisation précédente.

	Processeurs lents			*Processeurs rapides*		
Nb clients	**Bonnes (%)**	**Tolérables (%)**	**Processeur (%)**	**Bonnes (%)**	**Tolérables (%)**	**Processeur (%)**
6800	99,9	100	90	100	100	90
7200	99,9	100	98	100	100	98
8000	97,8	100	100	98,9	100	100
8800	25,4	97,8	100	54	98	100

TABLE 6 – **Apache + Optim. 1 et 2 :** résultats SPECweb2005 et utilisation du processeur sur les processeurs lents et rapide (configuration 16 processeurs).

7. Troisième optimisation : suppression des goulots d'étranglements logiciels

Les deux premières optimisations présentées dans ce papier permettent d'utiliser au mieux les ressources de la machine. Dans cette section, nous nous intéressons aux goulots d'étranglement logiciels. Nous introduisons tout d'abord une métrique simple, le potentiel de gain (PPG, *Potential Performance Gain*), et montrons comment elle nous a permis de trouver une fonction dans le coeur d'Apache qui provoque de la contention dans le noyau. Ensuite, nous présentons une solution et montrons qu'Apache, modifié avec nos trois optimisations, est 33% plus performant à 4 processeurs qu'un Apache non modifié.

7.1. Diagnostic

Les goulots d'étranglement logiciels sont des fonctions moins efficaces à 4 processeurs qu'à 1 processeur. La performance d'une fonction peut se mesurer en utilisant le nombre de cycles passés dans la fonction par octet transmis aux clients. Une fonction est plus efficace à 1 processeur si elle passe moins de cycles par octet transmis qu'à 4 processeurs. Notons $\mathrm{NCPB}_{\mathtt{f}}^{1\_\mathrm{proc}}$ (resp. $\mathrm{NCPB}_{\mathtt{f}}^{4\_\mathrm{procs}}$) le nombre de cycles passés dans une fonction `f` par octet transmis à 1 processeur (resp. 4 processeurs). Notons $\mathrm{PC}_{\mathtt{f}}^{4\_\mathrm{procs}}$ le pourcentage des cycles totaux passés dans cette fonctions à 4 processeurs. Nous définissons le potentiel de gain de la fonction `f`, noté $\mathrm{PPG}_{\mathtt{f}}$, comme l'amélioration de performance qui serait observée si la

fonction `f` était aussi efficace à 4 processeurs qu'à 1 processeur. Ce potentiel de gain est défini comme :

$$\mathrm{PPG}_f = \frac{\mathrm{NCPB}_f^{4\_procs} - \mathrm{NCPB}_f^{1\_proc}}{\mathrm{NCPB}_f^{4\_procs}} \times \mathrm{PC}_f^{4\_procs}$$

Pour illustrer cette métrique, considérons le cas d'une fonction `f` qui représente 2 cycles par octet transmis à 1 processeur et 10 cycles à 4 processeurs. Cette fonction est 5 fois moins efficace à 4 processeurs. Si le pourcentage de cycles passés dans la fonction `f` est de 20%, alors $\mathrm{PPG}_f = \frac{10-2}{10} \times 20\% = 16\%$. On peut donc réduire le temps passé dans cette fonction de 16%. Il est important de noter que le potentiel de gain d'une fonction `f` ne représente que les cycles réellement passés dans la fonction `f`, par opposition aux cycles passés dans les fonctions appelées par `f`. Le potentiel de gain a été défini de cette manière volontairement, sinon les fonctions racines d'un programme (`main`, etc.) auraient toujours un potentiel de gain proche de 100%, masquant ainsi les causes réelles d'inefficacité.
Nous avons calculé le potentiel de gain de toutes les fonctions utilisées par Linux-Apache-PHP. Les fonctions possédant le plus haut potentiel de gain sont présentées dans le tableau 7. La plupart de ces fonctions appartiennent au noyau et à des bibliothèques système. Notre objectif étant de résoudre les problème au niveau applicatif, nous nous sommes intéressés aux fonctions d'Apache et de PHP qui provoquent le plus d'appels aux fonctions incriminées. Pour ce faire, nous avons créé un *profiler*, basé sur de l'échantillonnage, qui sauvegarde, pour chaque échantillon, la pile d'exécution complète. En analysant ces piles d'exécution, nous avons constaté que 45% des appels aux fonctions incriminées proviennent majoritairement d'une seule fonction : `ap_directory_walk`.

Fonction	**PPG (%)**
__d_lookup	2,49%
_atomic_dec_and_lock	2,32%
lookup_mnt	1,41%
copy_user_generic_string	0,83%
memcpy	0,76%

TABLE 7 – **Apache + optim. 1 et 2 :** PPG les plus importants.

7.2. Solution

Pour améliorer l'efficacité d'Apache, la première étape consiste à comprendre le comportement de la fonction `ap_directory_walk`. Cette fonction est responsable de (i) la récupération des métadonnées des fichiers (taille, date de modification) et (ii) la vérification des droits d'accès aux fichiers. Pour améliorer les performances d'Apache, nous avons introduit un cache des résultats de cette fonction. Avant d'appeler cette fonction, Apache vérifie qu'une requête similaire n'a pas déjà été traitée. Si c'est le cas, alors Apache retourne le résultat déjà présent en cache sans appeler `ap_directory_walk`.
Nous utilisons l'interface `inotify` pour remonter les changements du système de fichiers à Apache. Les entrées modifiées sont invalidées dans le cache. Les fichiers servis par SPECweb sont très rarement modifiés et les permissions des dossiers ne changent jamais. Le cache des résultats de `ap_directory_walk` est donc très efficace. Il en est de même dans la majorité des serveurs en production, où la plupart des fichiers ne sont jamais modifiés et les données mutables sont stockées en base de données.

7.3. Évaluation

Nous avons évalué le gain de performance apporté par cette troisième optimisation. Les résultats sont présentés dans le tableau 1 (quatrième ligne). À quatre processeurs, cette optimisation améliore les performance de 10% par rapport à Apache modifié avec nos deux premières optimisations. Au total, nous obtenons une amélioration de 33% par rapport à un Apache non modifié. Notons que ce cache améliore légèrement les performances à 1 processeur (+4.5%), puisqu'il permet d'éviter quelques appels inutiles à `ap_directory_walk`.

8. Discussion

Comme d'autres chercheurs l'ont souligné [16, 9], la compréhension fine des problèmes de performance sur les architectures multi-cœurs est souvent plus difficile que l'élaboration de solutions. Dans nos travaux, nous avons fait face à un ensemble de difficultés que nous résumons dans cette section.

Obtenir une analyse fine est difficile. En essayant d'analyser le serveur avec Oprofile [13], nous nous sommes heurtés à différents problèmes. Premièrement, la calibration précise de cet outil est ardue : un seuil de déclenchement trop haut provoque la perte d'une partie des événements, un seuil trop bas rend les résultats peu représentatifs. Deuxièmement, sous une charge importante, il est difficile de démarrer et d'arrêter le mécanisme de profilage à des points précis de l'exécution, ce qui provoque des résultats imprécis. L'utilisation de priorités ou de groupes d'ordonnancement ne résout pas ce problème.
Pour résoudre ces problèmes, nous avons créé *lightProf*, un outil de profilage léger qui récupère périodiquement la valeur des compteurs d'événements matériels voulus (e.g. le nombre de fautes de cache depuis le début de l'expérience). Ainsi, aucune calibration n'est nécessaire. De plus, comme des informations temporelles sont stockées avec les valeurs de compteurs échantillonnées, il est possible de n'observer qu'une partie de l'exécution.

Une analyse exhaustive n'est pas possible. Les problèmes matériels sont particulièrement difficiles à dépister à cause du nombre de causes possibles. Notre machine permet d'observer 123 événements de performance différents, chacun pouvant être raffiné. Il n'est possible d'observer que 4 événements à la fois et multiplexer l'observation n'est pas une solution satisfaisante, car, par expérience, cela réduit fortement la précision des mesures. Par conséquent, tester et analyser les résultats pour chacun de ces évènements n'est pas viable, surtout si l'expérience requiert plusieurs dizaines de minutes.
Nous avons choisi de commencer avec une métrique de performance globale qui est l'IPC, pour nous convaincre que le matériel est une source de problèmes de passage à l'échelle. Nous avons ensuite déterminé un ensemble d'événements de performance pour chacun des composants de la machine (caches, liens HT, ...). Au total, nous avons mesuré 37 événements de performance différents pour avoir une vue globale de la machine. À partir de ces événements, nous calculons les métriques présentées dans ce papier (utilisation des liens HT, IPC, . . .) pour détecter une utilisation inefficace des ressources matérielles.

Détecter les problèmes nécessite plusieurs niveaux d'observation. L'étape précédente permet de produire une vue synthétique des problèmes matériels et considère l'application comme une boite noire. Cette étape n'est pas nécessairement suffisante, soit parce que l'utilisation pathologique des ressources ne peut pas être expliquée simplement, soit parce que le problème est principalement logiciel. Dans cette situation, il est nécessaire de déterminer précisément quelles sont les portions du code qui sont fautives et améliorables. Cette recherche peut s'avérer parfois difficile si on ne considère que l'évolution du temps passé dans les fonctions. Deux autres indicateurs sont d'une aide précieuse dans cette recherche : le potentiel de gain (PPG), défini dans la section 7.1, et l'analyse des appelants de la fonction.

9. Conclusion et ouverture

Dans ce papier, nous avons mis en avant le rôle des optimisations applicatives sur les performances des architectures NUMA. Nous avons présenté une méthodologie permettant de diagnostiquer trois goulots d'étranglement dans le serveur Apache ainsi que trois solutions simples à mettre en oeuvre. Notre version d'Apache est 33% plus performante que la version non modifiée. Elle passe quasiment idéalement à l'échelle de 1 à 4 processeurs. Nous imputons les 5.4% de performance manquants à la légère baisse de l'IPC de 1 à 4 processeurs et à quelques inefficacités mineures des algorithmes du noyau. Dans cette section, nous discutons de la généralité de nos observations.

Portée de nos observations. Nous pensons que les problèmes d'accès distants observés sur notre machine sont également présents sur d'autres architectures. Baumann et al. [2], par exemple, décrivent une machine 32 cœurs dans laquelle certains processeurs ont besoin de 4 sauts pour accéder à la mémoire alors que d'autres n'ont besoin que de deux sauts. Les derniers prototypes de processeurs (tels que le SCC d'Intel) introduisent également de l'asymétrie entre les cœurs d'un même processeur : cer-

tains cœurs n'ont pas d'accès direct à la mémoire. Nous avons également l'intention de tester d'autres types de charge, en particulier des charges en boucle ouverte ou semi-ouverte [19]. Nous pensons que l'asymétrie des architectures NUMA aura toujours un impact significatif dans ces différents contextes.

Applicabilité du N-Copy. L'utilisation de plusieurs instances d'un serveur Web pour améliorer les performances n'est pas nouvelle [16, 4]. Dans ce papier, nous mettons en évidence un nouvel avantage de cette approche : réduire l'impact du NUMA sur les performances. Nous montrons également qu'une telle approche nécessite une répartition de charge intelligente entre les différentes instances du serveur.

Point de vue du système d'exploitation. Une étude récente de Boyd et al. sur le noyau Linux[4] a également mis en avant des goulots d'étranglement dans le système de fichiers virtuel et proposé des solutions pour les contourner. Il serait intéressant de comparer l'impact de ces modifications à notre solution. Nous comptons également étudier la manière dont les compteurs de performance pourraient être utilisés par l'ordonnanceur pour améliorer dynamiquement la gestion des ressources de la machine.

Bibliographie

1. Apache. The Apache HTTP Server Project. `http://httpd.apache.org`.
2. Baumann (Andrew) et al. – The multikernel : a new OS architecture for scalable multicore systems. *In : SOSP*. – October 2009.
3. Boyd-Wickizer (Silas) et al. – Corey : An Operating System for Many Cores. *In : OSDI*. – December 2008.
4. Boyd-Wickizer (Silas) et al. – An Analysis of Linux Scalability to Many Cores. *In : OSDI*. – October 2010.
5. Bryant (Ray) et al. – Scaling Linux to the Extreme : From 64 to 512 Processors. *In : Linux Symposium*. – July 2004.
6. Cardellini (Valeria) et al. – The State of the Art in Locally Distributed Web-Server Systems. *ACM Computing Surveys*, 2002, pp. 263–311.
7. Choi (Gyu Sang) et al. – A Multi-Threaded PIPELINED Web Server Architecture for SMP/SoC Machines. *In : WWW*. – 2005.
8. eAccelerator. `http://eaccelerator.net/`.
9. Hardesty (L.). – Linux and Many-Core Scalability ? No Problem, Say Researchers., 2010. `http://www.drdobbs.com/open-source/227501025`.
10. Krieger (Orran) et al. – K42 : building a complete operating system. *In : EuroSys*. – 2006.
11. Nahum (Erich) et al. – Performance issues in www servers. *IEEE/ACM Transaction on Networking*, vol. 10, n1, 2002.
12. Netcraft. – Web Server Survey, 2011. `http://news.netcraft.com/`.
13. OProfile. A System Profiler for Linux `http://oprofile.sourceforge.net`.
14. Ousterhout (John) et al. – The case for RAMClouds : Scalable High-Performance Storage Entirely in DRAM. *SIGOPS OSR*, vol. 43, n4, 2010, pp. 92–105.
15. Pariag (David) et al. – Comparing the Performance of Web Server Architectures. *In : EuroSys*. – June 2007.
16. Pesterev (Aleksey) et al. – Locating Cache Performance Bottlenecks Using Data Profiling. *In : EuroSys*. – 2010.
17. PHP. – Hypertext Preprocessor. `http://www.php.net/`.
18. Saha (Bratin) et al. – Enabling Scalability and Performance in a Large Scale CMP Environment. *In : EuroSys*. – June 2007.
19. Schroeder (Bianca) et al. – Open Versus Closed : a Cautionary Tale. *In : NSDI*. – May 2006.
20. Standard Performance Evaluation Corporation, 2005. `http://www.spec.org/web2005/`.
21. Veal (Bryan) et Foong (Annie). – Performance Scalability of a Multi-Core Web Server. *In : ANCS*. – December 2007.
22. Zeldovich (Nickolai) et al. – Multiprocessor Support for Event-Driven Programs. *In : USENIX ATC*. – June 2003.
23. Zhuravlev (Sergey) et al. – Addressing shared resource contention in multicore processors via scheduling. *In : ASPLOS*. – 2010.