RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Vol de Tâches Efficace pour Systèmes Événementiels Multi-cœurs

Fabien Gaud[1], Sylvain Genevès[1], Fabien Mottet[2]

[1]Université de Grenoble, [2]INRIA Rhône-Alpes
{prenom.nom}@inria.fr

Résumé
De nombreux serveurs de données à hautes performances utilisent le paradigme de programmation événementiel. Comme les plates-formes multi-cœurs sont désormais omniprésentes, il devient crucial de pouvoir efficacement tirer parti du parallélisme matériel avec ce modèle. Pour cela, les supports d'exécution événementiels multi-cœurs utilisent le principe du vol de tâches. Ce papier étudie l'impact du mécanisme de vol de tâches sur les performances globales du système, dans le cas du support d'exécution Libasync-smp. Nous montrons tout d'abord que l'algorithme original implémenté dans Libasync-smp n'est pas toujours efficace. Par exemple, nous montrons qu'il peut réduire de 10% les performances d'un serveur Web s'exécutant sans vol de tâches. Nous présentons ensuite un nouvel algorithme de vol de tâches que nous évaluons avec des micro-tests et des applications réelles. L'évaluation montre que notre algorithme permet d'améliorer systématiquement les performances de Libasync-smp. En particulier, les performances obtenues sur le serveur Web sont améliorées jusqu'à 15% par rapport à la solution d'origine.

Mots-clés : architectures multi-cœurs, programmation événementielle, vol de tâches, serveurs de données, performance

1. Introduction

La programmation événementielle est une approche fréquemment employée pour le développement d'applications réparties efficaces [13, 11, 2, 12]. Les points forts de ce modèle résident dans son expressivité (gestion à grain fin de la concurrence, notamment pour les E/S disque et réseau asynchrones), la portabilité et dans certains cas une empreinte mémoire plus faible et de meilleures performances que les autres modèles de programmation.

Cependant, l'approche événementielle traditionnelle n'est pas en mesure d'exploiter efficacement les plates-formes multiprocesseurs récentes. En effet, le support d'exécution associé est généralement articulé autour d'un seul processus léger (*thread*), restreignant ainsi le traitement des événements à un seul cœur. L'approche la plus adoptée pour utiliser les différents cœurs, connue sous le nom de *N-Copy*, consiste à avoir plusieurs instances de la même application. Cette solution a l'avantage d'être rapide à déployer, mais peut cependant réduire l'efficacité de l'application et est difficile à mettre en œuvre si les instances doivent partager un état [18].

Dans cet article, nous nous intéressons à l'approche adoptée par Libasync-smp [18] parce qu'elle préserve l'expressivité du modèle événementiel classique, qu'elle offre un modèle de programmation permettant de gérer les problèmes de concurrence d'accès aux données et qu'elle est applicable à un grand nombre d'applications existantes. Nous étudions notamment le mécanisme de vol de tâches, utilisé pour distribuer efficacement les événements sur les cœurs disponibles et proposons de nouvelles heuristiques pour améliorer son comportement. Notre travail révèle que des modifications simples de l'algorithme de vol de tâches conduisent à des changements notables sur les débits des serveurs de données. Plus précisément, nos contributions principales sont les suivantes : (i) nous montrons qu'activer le mécanisme de vol de tâches patrimonial peut réduire les performances dans certains cas (jusqu'à -10%), (ii) nous introduisons trois nouvelles heuristiques de vol de tâches pour améliorer les performances globales du

système, (iii) nous évaluons ces heuristiques au moyen d'un ensemble de micro-tests ainsi que de deux applications : un serveur de fichiers sécurisé et un serveur Web simplifié. Nous montrons, par exemple, que notre algorithme de vol de tâches permet d'améliorer le débit du serveur Web de 15% comparé au mécanisme patrimonial de vol de tâches.
La suite de l'article est organisée comme suit. La section 2 résume les aspects principaux du support d'exécution original de Libasync-smp et présente ses limitations. La section 3 introduit les heuristiques proposées pour améliorer l'efficacité du mécanisme de vol de tâches. L'évaluation de notre contribution est exposée dans la section 4, en utilisant à la fois des micro-tests et des applications réelles. La section 5 situe notre approche par rapport aux travaux existants. Enfin, la section 6 présente nos conclusions ainsi que les directions futures de nos travaux.

2. Libasync-smp

Cette section décrit le support d'exécution Libasync-smp. Une application événementielle est structurée autour de traitants réagissant à des événements. Un événement peut être, par exemple, une E/S ou un événement interne engendré par un autre traitant. Nous nous intéressons ici particulièrement à la façon dont les événements sont répartis sur les cœurs d'exécution disponibles.

Architecture logicielle. Chaque cœur dispose d'un thread et d'une file d'événements (également appelés *tâches*). Les événements sont des structures de données contenant un pointeur vers le traitant et un ensemble de paramètres. Le thread a pour rôle d'exécuter le traitant associé à chacun des événements de sa file. L'approche événementielle fait l'hypothèse que les traitants ne sont pas bloquants, ce qui explique pourquoi un unique thread par cœur est suffisant.
L'exécution simultanée de plusieurs traitants sur différents cœurs nécessite une prise en compte d'eventuels problèmes de concurrence d'accès à des données partagées. Par exemple, si deux traitants ont besoin de mettre à jour la même donnée, il est nécessaire qu'ils s'exécutent en exclusion mutuelle. Pour ce faire, le support d'exécution Libasync-smp ne se base pas sur des mécanismes de verrous dans le code des traitants. À l'inverse, la concurrence est gérée à l'échelle d'un traitant : le programmeur doit spécifier les contraintes sur l'exécution d'événements simultanés en utilisant des couleurs. Deux événements avec différentes couleurs pourront être exécutés en concurrence alors que l'exécution des événements de la même couleur sera sérialisée. Cela est obtenu en assignant ces événements au même processeur. Par défaut, tous les événements ont la même couleur, ce qui permet ainsi d'ajouter du parallélisme de manière incrémentale.
De façon intéressante, le mécanisme de coloration permet la mise en œuvre de différentes formes de parallélisme. Par exemple, il est possible d'autoriser différentes instances d'un même traitant à travailler sur des données disjointes (coloriage par flot) en les coloriant avec des couleurs différentes. Pour un serveur de données, les événements sont généralement coloriés avec le numéro de connexion. On peut ainsi traiter simultanément plusieurs connexions. Il est aussi possible de garantir que toutes les instances d'un même traitant seront exécutées en exclusion mutuelle (coloriage par traitant) en leur assignant la même couleur[1]. Ceci est utilisé lorsqu'un traitant maintient un état global (par exemple, le nombre de clients connectés).
Les événements colorés sont distribués sur les cœurs en utilisant une simple fonction de *modulo*. Cette répartition de charge simple ignore le fait que certaines couleurs nécessitent plus de temps pour être traitées que d'autres (par exemple, lorsqu'il y a beaucoup d'événements d'une couleur ou lorsque différents événements ont des coûts de traitement différents). Pour ajuster dynamiquement la charge sur les multiples cœurs, la bibliothèque Libasync-smp implémente un mécanisme d'équilibrage basé sur un algorithme de vol de tâches.

Algorithme de vol de tâches. L'algorithme de vol de tâches de Libasync-smp fonctionne de la manière suivante : lorsqu'un thread n'a plus de tâche à exécuter, il vérifie si l'un des autres threads a encore des événements dans sa file. Si c'est le cas, le thread inoccupé essaye de voler certains d'entre eux : il commence par choisir une couleur à voler (en excluant celles en cours d'exécution) et déplace dans sa file

[1] Dans ce cas précis, le modèle d'exécution ressemble à celui mis en place dans SMP Click [10].

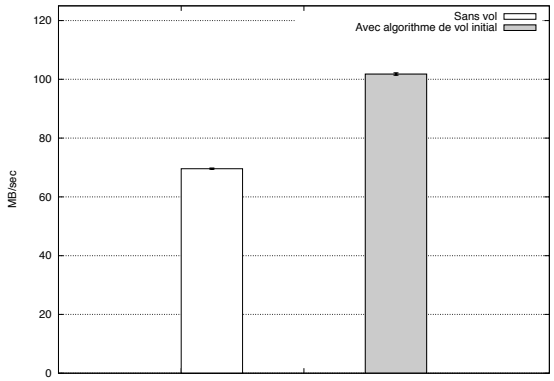

FIG. 1 – Performance du serveur de fichiers SFS avec et sans vol de tâches.

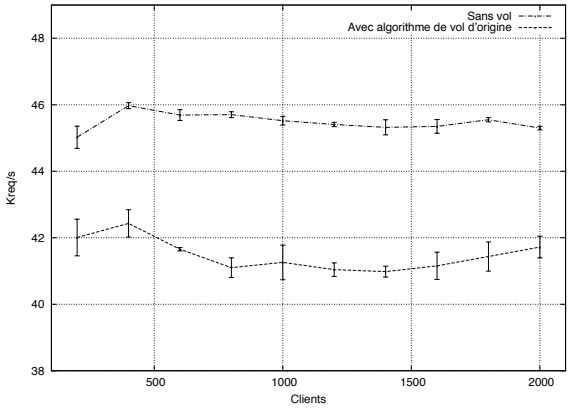

FIG. 2 – Performance du serveur Web SWS avec et sans vol de tâches.

tous les événements de cette couleur. De cette façon, l'exclusion mutuelle, garantie pour les événements de même couleur, est préservée. Lorsque plusieurs threads sont potentiellement volables, l'algorithme de vol choisit celui qui a le plus d'événements dans sa file.

Performance. Une évaluation de Libasync-smp a été faite par Zeldovich et al. [18]. Elle montre que des applications réelles tirent avantageusement parti des multiprocesseurs. L'évaluation utilise deux applications différentes : le serveur de fichiers sécurisé SFS [18] et un serveur Web. Néanmoins, l'impact du vol de tâches n'est pas étudié dans le contexte de ces deux applications.
Nous avons testé le serveur de fichiers SFS et un serveur Web en activant et en désactivant le mécanisme de vol de tâches. La figure 1 présente le débit atteint par le serveur de fichiers lorsque 15 clients émettent des requêtes. Nous pouvons clairement voir que le vol de tâches permet d'améliorer significativement le débit du serveur (+30%). La figure 2 présente le débit soutenu par le serveur Web lorsque le nombre de clients varie de 200 à 2000. Les résultats montrent que, dans ce cas, le vol de tâches de Libasync-smp dégrade les performances jusqu'à -10%.
Comme mentionné par Zeldovich et al. [18], SFS est un bon candidat pour la parallélisation à l'aide du vol de tâches car il exécute essentiellement des opérations de cryptographie à gros grain très coûteuses. Le serveur Web est lui composé de traitants à grain fin qui sont moins consommateurs de ressources processeur. Cela implique donc qu'il est plus difficile d'obtenir un bon passage à l'échelle d'une telle application sur une machine multi-cœur. Nous nous intéressons donc à améliorer les performances de cette classe d'applications pour laquelle le vol de tâches présent dans Libasync-smp est pénalisant.
La section suivante présente les optimisations que nous avons intégrées au mécanisme de vol de tâches, dans le but d'améliorer son comportement dans de telles circonstances.

3. Un nouvel algorithme de vol de tâches

Dans cette section, nous présentons les changements faits à l'algorithme de vol de tâches pour améliorer son comportement sur les applications où les coûts induits réduisent les gains potentiels. Nous introduisons trois heuristiques différentes, nommées *cache-aware stealing*, *batch stealing*, et *handler pinning*, qui sont décrites en détails dans les sous-sections suivantes.

3.1. Cache-aware stealing

La hiérarchie des caches (L1, L2, L3) a une grande importance sur les performances des applications. Certains de ces caches sont dédiés à un seul cœur, d'autres sont partagés par plusieurs cœurs. Par exemple, dans les récents processeurs quadri-cœurs Intel Xeon, les cœurs sont découpés en deux groupes de deux cœurs. Chaque cœur dispose d'un cache L1 privé de 64kB et partage un cache L2 de 6MB avec l'autre cœur de son groupe (appelé *voisin*).

Il devient donc crucial d'architecturer les algorithmes pour prendre en compte ces hiérarchies de caches hétérogènes. Le vol de tâches est un mécanisme central et doit donc tenir compte de la hiérarchie de caches. Dans la solution d'origine de Libasync-smp, la victime est choisie uniquement en fonction du nombre d'événements dans sa file et ne prend pas en compte la hiérarchie mémoire. Lorsqu'une couleur est déplacée vers un cœur non voisin, les données associées aux événements déplacés doivent être rechargées dans le cache du nouveau cœur possesseur de la couleur, provoquant ainsi de coûteuses fautes de cache.
L'heuristique *cache-aware stealing* essaye donc de favoriser le vol sur un cœur voisin plutôt que sur un cœur plus distant pour favoriser le maintien des données en cache.

3.2. Batch stealing

La version d'origine du vol de tâches utilise des verrous pour assurer l'exclusion mutuelle sur les files d'événements des différents cœurs. Plus précisément, l'environnement d'exécution Libasync-smp repose sur des verrous à attente active pour éviter des changements de contexte, qui ne sont pas nécessaires puisqu'il n'y a qu'un seul thread par cœur. Cependant, malgré cette optimisation, les verrous peuvent devenir un facteur limitant.
La seconde heuristique que nous proposons a pour but d'amortir les coûts d'acquisition de verrous. Dans la version d'origine du vol de tâches, il n'y a pas de limite sur le nombre minimum d'événements à voler (le voleur ne choisissant qu'une seule couleur et donc uniquement les événements associés). Nos expériences ont montré que les événements étaient souvent volés en très petites quantités, impliquant donc des vols fréquents[2]. L'heuristique *batch stealing* permet de voler plus de travail. Plus précisément, plutôt que de voler une seule couleur, un thread essaye de voler un pourcentage du nombre d'événements présents dans la file de la victime (fixé à 50% dans nos évaluations). Pour atteindre ce pourcentage, un thread peut être amené à voler plusieurs couleurs. Pour garantir la cohérence des données (basée sur les couleurs), tous les événements associés aux couleurs choisies sont volés

3.3. Handler pinning

La version d'origine de Libasync-smp ne permet pas d'assigner un traitant sur un cœur donné (i.e. assurant que tous les événements associés à ce traitant seront exécutés sur le même cœur)[3]. Néanmoins, nous pensons que cette possibilité peut être utile pour certains traitants particuliers. C'est par exemple le cas des traitants effectuant des opérations réseau telles qu'accepter des connexions TCP. Il a été observé que, pour soutenir un débit important, les serveurs doivent accepter les connexions à un taux suffisamment important [9]. Nous pensons qu'assigner les traitants à un coeur acceptant les connexions TCP est une manière simple de garantir un taux d'acceptation important, et ainsi d'obtenir de bonnes performances. En effet, les cœurs qui possèdent des traitants assignés vont dédier plus de temps à l'exécution de ces traitants. Cela vient du fait que ces cœurs vont se faire voler les autres couleurs. De plus, s'ils ont suffisamment d'événements associés avec ces traitants dans leur files, ils vont voler moins fréquemment que les autres cœurs. Par conséquent, assigner le traitant acceptant les connexions TCP devrait améliorer le taux auquel les connexions sont acceptées et augmenter ainsi les performances globales du serveur.
Le nombre de traitants assignés est généralement faible (dans nos expérimentations nous n'assignons que les traitants acceptant et fermant des connexions), une majorité d'événements pouvant donc être volés. Cela implique que les cœurs auront toujours du travail (en volant potentiellement des événements sur un autre cœur). Ainsi, assigner quelques traitants ne réduit pas le parallélisme de manière significative.

4. Évaluation

Nous avons mis en œuvre nos heuristiques au sein du support d'exécution Libasync-smp. Le fonctionnement de base de notre algorithme de vol de tâches est exactement le même que celui de la version

[2] La fréquence de vol importante est aussi conditionnée par le fait que le traitement d'un événement est généralement de courte durée. Cela est en contraste avec la granularité des tâches manipulées par les supports d'exécution pour la programmation à base de threads (tels que Cilk [8]).

[3] La version d'origine de Libasync-smp utilise en fait une forme limitée d'assignation : elle assure que tous les événements de la couleur 0 seront exécutés sur le premier cœur de la machine. Cette caractéristique n'est pas documentée dans le papier original [18] ni dans le code.

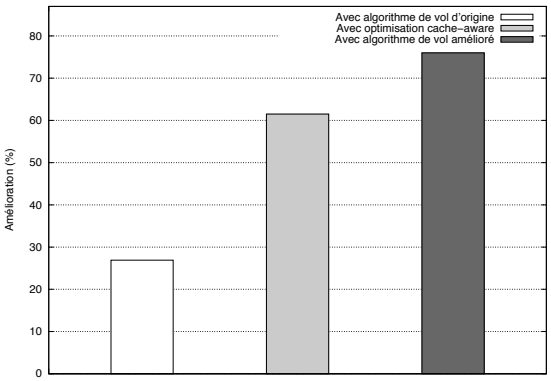

FIG. 3 – Évaluation de l'heuristique *cache-aware*.

Avec algorithme de vol d'origine	71%
Avec optimisation *cache-aware*	44%
Avec algorithme de vol amélioré	36%

FIG. 4 – Fautes de cache L2 observées sur les différentes configurations.

d'origine (décrit en Section 2).
Nous avons évalué nos heuristiques en utilisant à la fois des micro et des macro-tests. L'objectif des micro-tests est d'isoler l'impact des trois heuristiques complémentaires, présentées en Section 3. Le but des macro-tests est de montrer que notre vol de tâches amélioré augmente les performances d'applications réelles. Nous avons choisi deux applications : un serveur Web de contenu statique et un serveur de fichiers sécurisé. Nos résultats montrent que notre algorithme de vol de tâches amélioré dépasse constamment la version d'origine du support d'exécution avec ou sans vol de tâches.
Nous décrivons premièrement nos paramètres expérimentaux, puis nous présentons les résultats obtenus sur les micro-tests. Enfin, nous donnons les résultats obtenus sur les applications réelles.

4.1. Paramètres expérimentaux

Matériel. Le test SFS a été réalisé sur un bi-Xeon E5120 *Woodcrest*. Chaque processeur est composé de deux cœurs, partageant un cache L2 de 2MB. Les micro-tests et le test serveur Web ont été réalisés sur un Intel bi-Xeon E5410 *Harpertown*. Chaque processeur est composé de quatre cœurs. Pour aider à la comparaison avec les résultats obtenus sur SFS, nous n'activons que quatre cœurs au total (deux sur chaque processeur). Les cœurs actifs d'un même processeur partagent un cache L2 de 6MB. Chacune des machines est équipée de 8GB de mémoire et d'une carte réseau 1Gb/s.
Logiciel. Toutes les machines utilisent un noyau Linux 2.6.24. L'application est compilée avec GCC-2.95 (pour des raisons de compatibilité avec la version d'origine de Libasync-smp) en utilisant le niveau d'optimisation -O2 et utilise la bibliothèque Glibc-2.3.6. De plus, le jeu d'instructions 32 bits est utilisé pour des raisons de compatibilité avec GCC-2.95.

4.2. Micro-tests

L'objectif des micro-tests est de valider l'impact des trois heuristiques mises en œuvre dans notre algorithme de vol de tâches. Dans tous nos micro-tests, les traitants sont organisés en pipeline (i.e. tous les événements sont traités par tous les traitants successivement). Les tests varient par le traitement qui est fait des événements et par les données qui sont traitées.
Dans tous les tests (à l'exception du dernier, *handler pinning*), les événements sont alloués périodiquement de manière à ce qu'il y en ait toujours à traiter. De plus, les événements alloués ne sont pas répartis uniformément sur les cœurs. La répartition est choisie de manière à ce que la probabilité d'avoir des files vides soit importante, déclenchant ainsi le mécanisme de vol de tâches. Dans ces micro-tests, les événements sont majoritairement coloriés par flots (voir section 2) pour obtenir le maximum de parallélisme. Tous ces micro-tests ont été réalisés sur 100 itérations avec un écart type très faible.

4.2.1. Cache-aware stealing

Ce micro-test utilise cinq traitants. Le premier alloue un gros tableau (80kB) qu'il ajoute au message. Les trois traitants suivants font des opérations arithmétiques (addition, multiplication et modulo) sur tous les éléments du tableau. Le quatrième compte simplement le nombre d'événements qu'il reçoit pour

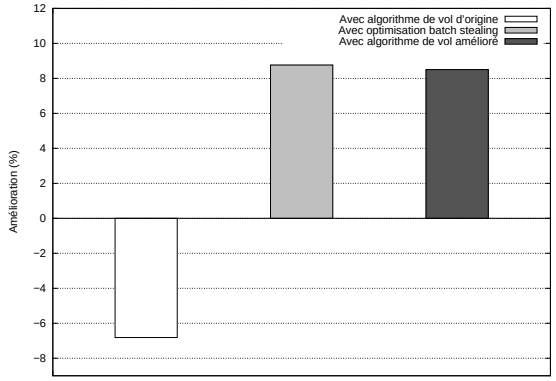

FIG. 5 – Évaluation de l'heuristique *batch stealing*.

Sans vol de tâches	1,2%
Avec algorithme de vol d'origine	15,4%
Avec optimisation *batch stealing*	5,1%
Avec algorithme de vol amélioré	5,8%

FIG. 6 – Pourcentage du temps total passé dans les fonctions de verrouillage.

déterminer la fin du test. Cette configuration est intéressante pour évaluer le vol de tâches *cache-aware* car dans ce cas déplacer un événement implique de déplacer les données associées (qui sont d'une taille importante dans ce test).
La figure 3 montre les résultats obtenus sur le micro-test. L'axe des ordonnées représente le gain amené par les différentes politiques de vol de tâches comparé à une version du support d'exécution sans vol de tâches. Nous comparons trois algorithmes de vol de tâches différents : l'algorithme d'origine, une version de notre algorithme où seule l'optimisation *cache-aware* est activée et la version où toutes les optimisations sont présentes (algorithme de vol amélioré).
La première observation que nous pouvons faire est que le vol de tâches améliore significativement les performances de l'application. Cette amélioration est de 27% avec le vol de tâches de Libasync-smp, de 62% avec notre optimisation cache-aware et de 76% avec toutes nos heuristiques activées. Nous pouvons ensuite observer que la politique *cache-aware* augmente significativement les performances (de 27% par rapport au vol de tâches de Libasync-smp). Finalement, nous observons que l'algorithme amélioré est légèrement meilleur que le vol de tâches *cache-aware* (10%). Cela montre notamment que l'amélioration apportée par la politique cache-aware n'est pas impactée négativement par les autres politiques.
Pour expliquer ces résultats, nous avons observé le nombre de fautes de cache L2 dans les différentes configurations grâce à la bibliothèque PAPI [4]. Nous n'avons observé que les caches L2 puisque ce sont les seuls partagés entre les cœurs. Les résultats sont présentés dans le tableau 4. Ils montrent que le vol de tâches d'origine entraîne un fort taux d'échecs dans le cache L2. D'un autre coté, les algorithmes *cache-aware* et amélioré ne provoquent plus que respectivement 44% et 36% de fautes de cache. Ce comportement confirme donc nos attentes.

4.2.2. Batch stealing

Ce micro-test est composé de cinq traitants. Quatre d'entre eux effectuent un travail léger (50 concaténations de chaînes de caractères). Le dernier compte simplement le nombre d'événements qu'il reçoit. Il est intéressant d'utiliser des traitants courts pour évaluer l'heuristique *batch stealing* car avec des traitant court le support d'exécution (et donc les mécanismes de verrouillage associés) est plus sollicité.
La figure 5 présente les performances du support d'exécution avec les différents algorithmes de vol de tâches (version d'origine, *batch stealing* uniquement, amélioré avec toutes les optimisations) comparées à une version sans vol de tâches. Nous pouvons observer premièrement que le vol de tâches d'origine a de moins bonnes performances (-7%) qu'une version sans vol. Deuxièmement, nous constatons que l'heuristique *batch stealing* améliore les performances significativement. Le gain est de 18% comparé au vol de tâches d'origine et de 10% comparé à la version sans vol. Enfin, nous remarquons que la version complète a sensiblement les mêmes performances que la version *batch stealing* ce qui montre que les autres heuristiques n'ont pas un impact négatif.

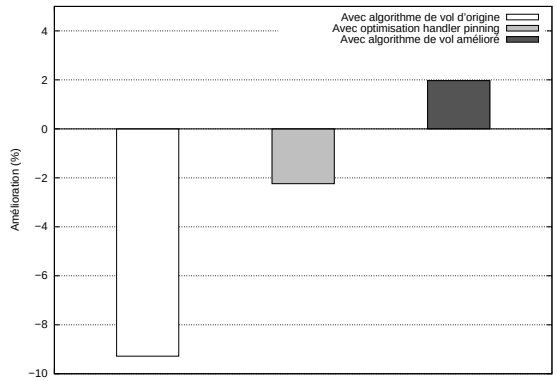

FIG. 7 – Évaluation de l'heuristique *handler pinning*.

Avec algorithme de vol d'origine	-8,5%
Avec optimisation *handler pinning*	+1,6%
Avec algorithme de vol amélioré	+2,4%

FIG. 8 – Pourcentage d'augmentation du taux d'acceptation des connexions TCP, comparé à la version sans vol de tâches.

Pour expliquer les résultats obtenus dans la figure 5, nous avons observé le temps passé dans les fonctions de verrouillage. Les résultats sont présentés dans le tableau 6. Nous observons, premièrement, que sans vol de tâches le temps passé dans les fonctions de verrouillage est très faible. Deuxièmement, la version *batch stealing* et la version améliorée du vol de tâches passent beaucoup moins de temps dans les fonctions de verrouillage que la version d'origine, ce qui est conforme à nos attentes.

4.2.3. Handler pinning

Ce micro-test consiste en un serveur simple comprenant cinq traitants ; un traitant acceptant les connexions TCP[4], un traitant lisant une requête d'un octet, un autre envoyant une réponse d'un octet et un dernier fermant la connexion. Le traitant acceptant les connexions et celui fermant les connexions sont coloriés de manière identique et assignés au premier cœur (pour les raisons présentées en section 3.3). Les autres événements sont coloriés de manière à ce que les requêtes faites par différents clients puissent être traitées en parallèle. Nous réalisons l'injection en utilisant un ensemble de clients connectés au serveur avec un commutateur Ethernet à 1Gb/s. Chaque client envoie un grand nombre de requêtes et calcule le débit auquel les réponses sont fournies par le serveur.

La figure 7 présente les résultats obtenus avec les différents algorithmes de vol de tâches (version d'origine, *pinning* uniquement, améliorée avec toutes les optimisations) comparé à une version sans vol. Nous pouvons faire différentes observations. Premièrement, le vol de tâches a des performances significativement moins bonnes que toutes les autres configurations (-9%). Nous attribuons ceci au fait que c'est la seule configuration avec laquelle le traitant acceptant les connexions TCP est déplaçable. Deuxièmement, la version avec ce traitant assigné au premier cœur a de légèrement moins bonnes performances que la version sans vol de tâches (-2.2%) et a de meilleures performances que la version d'origine du vol de tâches (+8%). Enfin, nous observons que notre version améliorée dépasse légèrement celle avec uniquement l'assignation de l'acceptation des connexions sur un coeur (+4%) et celle sans vol de tâches (+2%). Nous attribuons cette différence de performance à l'impact de l'heuristique de *batch stealing* (cf. 4.2.2). En effet, le traitement des événements est très court et donc, dans ce cas, le *batch stealing* permet d'amortir les coûts de verrouillage.

Pour expliquer les résultats observés dans la figure 7, nous avons tracé le taux d'acceptation des connexions TCP (puisque dans ce cas particulier, assigner le traitant à un coeur a pour but d'augmenter ce taux). Le tableau 8 présente l'évolution de ce taux (en pourcentage) comparé à la version sans vol. Le comportement observé est conforme aux attentes : les versions avec le traitant acceptant les connexions assigné à un cœur ont un taux d'acceptation plus élevé que la version d'origine du vol de tâches (+12%).

[4] Notons que ce traitant accepte les connexions par lot pour améliorer les performances.

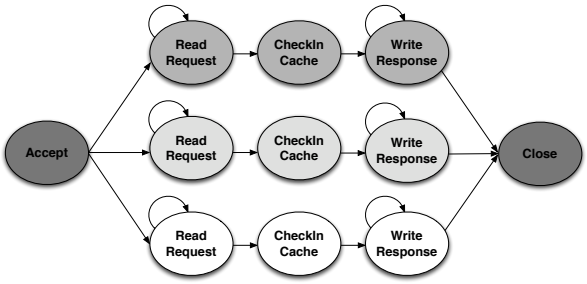

FIG. 9 – Architecture du serveur Web.

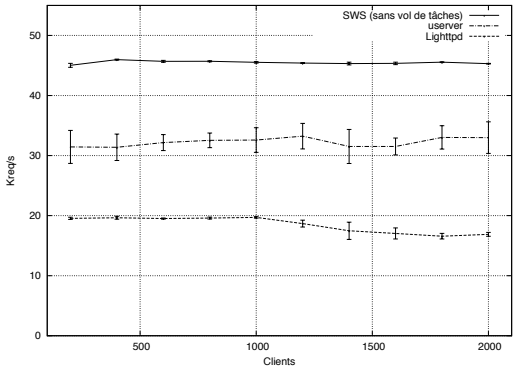

FIG. 10 – Comparaison de différents serveurs Web.

4.3. Macro-tests

Dans cette section, nous décrivons les deux applications précédemment introduites en section 2. La première est un serveur Web, un exemple d'application intensive au niveau du réseau. La seconde est un serveur de fichiers sécurisé (SFS), qui fait une utilisation intensive du processeur pour le chiffrement de la communication avec le client. Dans les deux cas, notre vol de tâches amélioré dépasse les performances de la version d'origine du support d'exécution avec ou sans vol de tâches. Dans ces tests, nous comparons trois différentes configurations : avec le vol de tâches d'origine, avec notre vol amélioré et sans vol.

4.3.1. Serveur Web

Dans cette partie, nous évaluons un serveur Web simplifié (SWS) que nous avons développé en utilisant la Libasync-smp. Nous n'avons pu utiliser ni celui initialement évalué avec Libasync-smp, car il n'est pas fourni avec Libasync-smp, ni OKWS [12] qui est un serveur Web développé avec une nouvelle version de Libasync, non compatible avec Libasync-smp et non multiprocesseur. SWS est un serveur Web simplifié ne gérant qu'un sous-ensemble du protocole HTTP, construisant les réponses au démarrage (une optimisation déjà décrite dans Flash [13]) et ne gérant que des erreurs basiques. L'architecture de SWS est illustrée par la figure 9. Les événements liés aux traitants Accept et Close sont coloriés avec la même couleur et assignés au premier cœur. Les autres événements sont coloriés de manière à ce que des requêtes venant de clients distincts puissent être traitées en parallèle.

Nous avons développé un injecteur de charge événementiel en boucle fermée similaire à celui décrit dans [6]. Les clients sont synchronisés grâce à un mécanisme maître/esclave. Chaque client établit une connexion, demande un ficher de 1kB, attend la réponse et ferme la connexion, avant de réitérer ce processus. Les clients sont déployés sur 5 machines Intel T2300 double cœur et connectés au serveur par un commutateur à 1Gb/s. Nous déployons sur chaque machine entre 40 et 400 clients.

Pour valider l'architecture de SWS, nous avons tout d'abord comparé ses performances à celles de deux serveurs Web efficaces : Lighttpd [3] et µserver [2]. Le serveur µserver utilise 4 processus pour exploiter les 4 cœurs disponibles (*N-Copy*). La figure 10 présente les résultats obtenus. Nous pouvons voir que SWS dépasse constamment les performances des autres serveurs. Nous attribuons ceci au fait que SWS est plus basique que les deux autres. Cette différence se réduirait probablement si SWS offrait plus de fonctionnalités. Cependant, ce test montre que SWS est efficace.

La figure 11 présente les débits observés. Comme nous l'avons dit en section 2, nous observons que le vol de tâches d'origine fait baisser les performances entre 7% et 10%. À l'inverse, notre algorithme est significativement meilleur que la version d'origine (jusqu'à 15%) et légèrement meilleur que la version sans vol de tâches (jusqu'à 4%). Ce test montre que les améliorations observées sur les micro-tests sont aussi valables avec une application réelle.

4.3.2. Serveur de fichiers sécurisé (SFS)

La seconde application que nous considérons est un serveur de fichiers sécurisé (SFS) [18]. Les clients

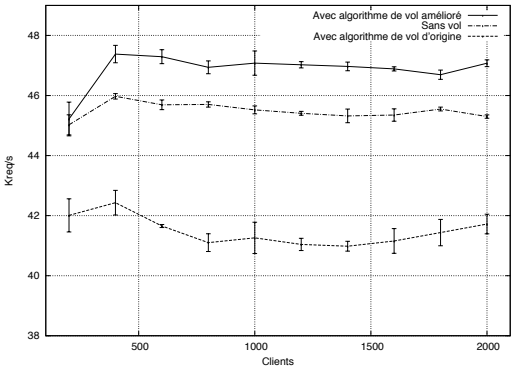

FIG. 11 – Impact des heuristiques sur le serveur Web.

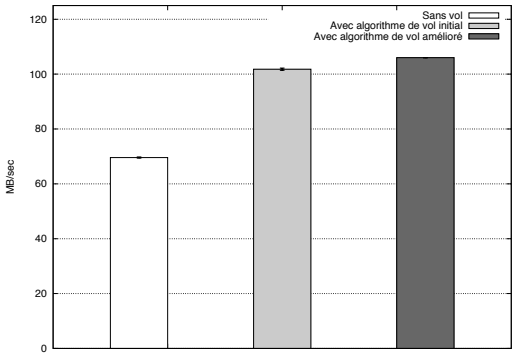

FIG. 12 – Impact des heuristiques sur SFS.

dialoguent avec le serveur en utilisant une communication TCP. De plus, toutes les communications avec le serveur sont chiffrées et authentifiées, rendant SFS consommateur de processeur. Nos expérimentations (et les observations initiales de Libasync-smp) ont montrées que 60% du temps est passé dans ces fonctions de chiffrement. Tout comme dans l'évaluation initiale de SFS, seuls ces traitants sont effectivement parallélisés.

L'injection de charge est faite en utilisant 15 machines Intel T2300 double cœur connectées au serveur par un commutateur 1Gb/s. Nous avons utilisé la suite de tests Multio [1] comme suit : chaque machine fait tourner un seul client lisant un fichier de 200MB à plusieurs reprises[5]. Le cache du client est vidé après chaque requête pour être sûr que la suivante sera bien envoyée au serveur. Chaque client évalue le débit du serveur et un maître est en charge de collecter ces données et de les agréger.

La figure 12 présente les débits observés. Comme mentionné dans la section 2, nous pouvons constater que le vol de tâches améliore significativement les performances (environ 30%). Enfin, nous observons que notre algorithme de vol de tâches est légèrement meilleur que la version d'origine (environ 3%). Conformément à nos attentes (voir section 2), le gain est plus faible que dans le cas du serveur Web.

5. État de l'art

Cet article n'a ni pour objectif de relancer le débat concernant les mérites du modèle à base de threads et du modèle événementiel [11, 16], ni celui de proposer une approche plus simple pour la gestion de la concurrence [5], mais a en revanche celui d'étudier les performances des applications événementielles s'exécutant sur des plates-formes multiprocesseurs.

Les performances des supports d'exécution multiprocesseurs basés sur des files d'événements ont été étudiées dans des travaux antérieurs, avec différentes hypothèses sur le modèle de programmation fourni (SEDA [17]) ou sur le domaine d'application et la granularité des tâches (SMP Click [10]). Avec SEDA, la répartition des tâches sur les unités d'exécution est laissée à l'ordonnanceur du système d'exploitation et, à notre connaissance, cet aspect n'a pas été étudié en détail. À cause des contraintes spécifiques mentionnées par ses auteurs, SMP Click ne peut pas utiliser un mécanisme de vol de tâches et utilise une technique spécifique. La mise en place d'une telle technique dans Libasync-smp est limitée par le fait que les deux supports d'exécution n'implémentent pas la même forme de parallélisme.

La performance des serveurs Web sur les plates-formes multi-cœurs a fait l'objet de récentes études. Veal et al. [15] ont identifié des goulots d'étranglement matériels empêchant le passage à l'échelle du serveur Web (à base de threads) Apache sur une plate-forme à 8 cœurs.

Des recherches antérieures sur les supports d'exécution événementiels monoprocesseurs ont montré les bénéfices de politiques d'ordonnancement avancées. Bhatia et al. [7] ont révélé que l'interaction entre le support d'exécution et l'allocateur mémoire conduit à une amélioration importante de l'utilisation des caches processeurs. Nous sommes en train d'étudier comment ces politiques d'ordonnancement

[5] Ce test est similaire à celui effectué dans l'évaluation de Zeldovich et al. [18]

local à un processeur peuvent être efficacement combinées avec les mécanismes de répartition de tâches présentés dans cet article.
Le vol de tâches à été introduit et étudié dans le contexte des applications à base de threads [8] et adapté au contexte du traitement d'événements dans la version d'origine de Libasync-smp [18]. McRT [14], le support d'exécution parallèle développé par Intel, peut aussi utiliser un mécanisme de vol de tâches pour répartir les threads ordonnancés de manière coopérative. Ses auteurs ont également exploré une variante *cache-aware* de l'algorithme. Cependant, l'évaluation de McRT s'est uniquement concentrée sur des modèles de charge liés aux applications bureautiques et multimédia.

6. Conclusion

La programmation événementielle est un paradigme populaire dont l'efficacité a été prouvée pour la conception de serveurs de données à hautes performances. Libasync-smp est un support d'exécution événementiel permettant de tirer parti des architectures multi-cœurs.
Dans ce papier, nous avons étudié le mécanisme de vol de tâches mis en œuvre dans Libasync-smp et proposé trois améliorations. Ces optimisations sont simples, faciles à mettre en œuvre et améliorent les performances significativement. Bien que nous ne nous soyons intéressés qu'au contexte de Libasync-smp, les optimisations proposées sont plus générales et nous pensons qu'elles sont applicables aisément à d'autres systèmes événementiels.
Nous envisageons différentes directions pour nos travaux futurs. Premièrement, nous pensons améliorer la mise en œuvre de l'heuristique *cache-aware*, notamment pour supporter différentes hiérarchies de caches. Deuxièmement, nous pensons fournir un support pour le choix des traitants assignés. Enfin, nous souhaitons étudier l'intérêt de notre approche pour d'autres domaines d'application ainsi que sur d'autres types d'architectures matérielles.

Bibliographie

1. The multio benchmark, 2004. http ://www.cisl.ucar.edu/css/software/multio/.
2. The µserver project, 2007. http ://userver.uwaterloo.ca.
3. The lighttpd project, 2008. http ://www.lighttpd.net.
4. The performance application programming interface (papi), 2008. http ://icl.cs.utk.edu/papi/.
5. Adya (Atul), Howell (Jon), Theimer (Marvin), Bolosky (William J.) et Douceur (John R.). – Cooperative Task Management Without Manual Stack Management. *In : Proceedings of the the 2002 USENIX Annual Technical Conference*. – Monterey, CA, USA, June 2002.
6. Banga (Gaurav) et Druschel (Peter). – Measuring the Capacity of a Web Server. *In : Proceedings of USITS'97*. – Monterey, CA, USA, 1997.
7. Bhatia (Sapan), Consel (Charles) et Lawall (Julia L.). – Memory-Manager/Scheduler Co-Design : Optimizing Event-Driven Servers to Improve Cache Behavior. *In : Proceedings of ISMM'06*. – Ottawa, Ontario, Canada, June 2006.
8. Blumofe (Robert D.), Joerg (Christopher F.), Kuszmaul (Bradley C.), Leiserson (Charles E.), Randall (Keith H.) et Zhou (Yuli). – Cilk : An Efficient Multithreaded Runtime System. *J. Parallel Distrib. Comput.*, vol. 37, n1, 1996, pp. 55–69.
9. Brecht (Tim), Pariag (David) et Gammo (Louay). – Acceptable Strategies for Improving Web Server Performance. *In : Proceedings of the 2004 USENIX Annual Technical Conference*. – Boston, MA, USA, july 2004.
10. Chen (Benjie) et Morris (Robert). – Flexible Control of Parallelism in a Multiprocessor PC Router. *In : Proceedings of the 2001 USENIX Annual Technical Conference*. – Boston, MA, USA, June 2001.
11. Dabek (Frank), Zeldovich (Nickolai), Kaashoek (Frans), Mazières (David) et Morris (Robert). – Event-Driven Programming for Robust Software. *In : Proceedings of the 10th ACM SIGOPS European Workshop*. – Saint-Emilion, France, September 2002.
12. Krohn (Maxwell). – Building Secure High-Performance Web Services with OKWS. *In : Proceedings of the 2004 USENIX Annual Conference*. – Boston, MA, USA, June 2004.
13. Pai (Vivek S.), Druschel (Peter) et Zwaenepoel (Willy). – Flash : An efficient and portable Web server.

In : Proceedings of the 1999 USENIX Annual Technical Conference. – Monterey, California, USA, June 1999.

14. Saha (Bratin), Adl-Tabatabai (Ali-Reza), Ghuloum (Anwar), Rajagopalan (Mohan), Hudson (Richard L.), Petersen (Leaf), Menon (Vijay), Murphy (Brian), Shpeisman (Tatiana), Sprangle (Eric), Rohillah (Anwar), Carmean (Doug) et Fang (Jesse). – Enabling Scalability and Performance in a Large Scale CMP Environment. *In : Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys'07)*. – Lisbon, Portugal, June 2007.
15. Veal (Bryan) et Foong (Annie). – Performance Scalability of a Multi-Core Web Server. *In : Proceedings of ANCS '07*. – Orlando, FL, USA, December 2007.
16. von Behren (Robert), Condit (Jeremy) et Brewer (Eric A.). – Why events are a bad idea (for high-concurrency servers). *In : Proceedings of HOTOS'03*. – Lihue, Hawaii, May 2003.
17. Welsh (Matt), Culler (David) et Brewer (Eric). – SEDA : An architecture for well-conditioned scalable internet services. *In : Proceedings of SOSP 2001*. – Banff Alberta Canada, October 2001.
18. Zeldovich (Nickolai), Yip (Alexander), Dabek (Frank), Morris (Robert), Mazières (David) et Kaashoek (M. Frans). – Multiprocessor Support for Event-Driven Programs. *In : Proceedings of the 2003 USENIX Annual Technical Conference*. – San Antonio, TX, USA, June 2003.