Université Joseph Fourier — Master 2 Recherche - Systèmes et Applications Réparties

---

Répartition adaptative d'évènements en contexte multiprocesseur

Projet réalisé par :
Sylvain GENEVÈS

Soutenu le :
23 juin 2008

Équipe Sardes
Inria Rhône-Alpes - Laboratoire d'Informatique de Grenoble

Encadrants :
Renaud Lachaize
Vivien Quéma

JURY :

Marie-Christine Fauvet	, Membre du jury permanent
Jean-Marc Vincent	, Membre du jury permanent
Marlon Dumas	, Membre du jury permanent
Sara Bouchenak	, Responsable de parcours
Jean-François Méhaut	, Examinateur externe
Renaud Lachaize	, Encadrant
Vivien Quéma	, Encadrant

Remerciements

Je tiens à remercier l'équipe Sardes pour leur accueil chaleureux ainsi que le financement de ces travaux. Je tiens à apporter toute ma gratitude à Renaud, Vivien et Fabien (Gaud et Mottet) pour leur encadrement, leur disponibilité ainsi que leur aide précieuse. Je remercie également ma famille et mes amis pour leur patience et leur participation à la relecture de ce rapport.

Table des matières

1 État de l'art **7**
1.1 Modèles de programmation . . . 7
1.1.1 Modèle de programmation par threads . . . 7
1.1.2 Modèle de programmation par événements . . . 8
1.2 Architecture matérielle et impact des caches . . . 10
1.2.1 Architectures modernes . . . 10
1.2.2 Optimisations liées aux caches . . . 13
1.3 Supports d'exécution adaptés aux architectures multiprocesseur . . . 15
1.3.1 SMP Click . . . 16
1.3.2 libasync-mp . . . 17
1.4 Synthèse . . . 19

2 Contribution **21**
2.1 Vers des supports d'exécution plus flexibles . . . 21
2.2 Architecture proposée . . . 22
2.2.1 Stratégies de placements . . . 23
2.2.2 Détail de l'architecture . . . 24
2.2.3 Mécanismes d'observation . . . 26
2.2.4 Prise de décision . . . 27
2.3 Synthèse . . . 30

3 Mise en œuvre **31**
3.1 Placements statiques . . . 31
3.1.1 Gestion de la synchronisation . . . 31
3.2 Observation des compteurs matériels . . . 32
3.2.1 Présentation de PAPI . . . 32
3.3 Adaptation . . . 33
3.4 Synthèse . . . 33

4 Résultats expérimentaux **35**
4.1 Conditions d'expérimentation . . . 35
4.2 Tests de placements statiques . . . 36
4.3 Tests avec adaptation dynamique . . . 39
4.4 Coût de l'observation . . . 42
4.5 Synthèse . . . 43

5 Conclusion **45**
5.1 Bilan 45
5.2 Perspectives 45

Table des figures

1.1 Exemple d'architecture multiprocesseur 12
1.2 Exemple d'architecture multicœur 13
1.3 Exemple d'architecture multicœur et multiprocesseur 13
1.4 Etat du programme pendant son exécution. Cette figure est tirée de l'article de Bhatia *et al.*[5] 15
1.5 un exemple de connexion avec Click 16
1.6 Architecture de libasync-mp 18

2.1 Vue d'ensemble de notre contribution 21
2.2 Le placement round robin 23
2.3 Architecture proposée 25
2.4 Redistribution de traitant d'événement 29

3.1 Exemple d'hystérésis. Les seuils sont différents selon le sens dans lequel on change d'état. 33

4.1 Résultats du benchmark CPU-intensif. 36
4.2 Résultats du benchmark filtres d'images. 37
4.3 Taux de fautes de cache L2 du benchmark filtres d'images. 38
4.4 Taux de fautes de cache L2 du benchmark CPU-intensif. 38
4.5 Résultats du benchmark CPU-intensif avec adaptation dynamique. 39
4.6 Synthèse des résultats du benchmark CPU-intensif. 40
4.7 Résultats du benchmark filtres d'images avec adaptation dynamique. 40
4.8 Synthèse des résultats du benchmark filtres d'images. 41
4.9 Résultats du benchmark CPU-intensif avec observation des compteurs de performances. 42
4.10 Résultats du benchmark filtres d'images avec observation des compteurs de performances. 43

Introduction

Problématique

Ce projet s'est déroulé dans l'équipe Sardes qui fait partie de l'INRIA[1] et du LIG[2]. Le projet Sardes s'intéresse à la définition de technologies logicielles pour la construction de systèmes adaptables.

Ces dernières années, nous avons assisté à la généralisation de certaines technologies matérielles, notamment les architectures multiprocesseur, mais surtout les processeurs multicœur . Ces technologies étaient jusqu'à maintenant réservées à des utilisations spécifiques, comme le calcul à haute performance (HPC[3]) par exemple. Dernièrement, de telles architectures parallèles sont sorties de ce cadre et ont notamment rejoint celui des serveurs de données et des ordinateurs personnels. Nous nous concentrerons principalement sur le cas de machines dédiées à une application, et bien que nos travaux soient généralisables à différents domaines applicatifs, notre étude est centrée sur le contexte des serveurs de données. Nous verrons dans quelle mesure les caches matériels de ces architectures jouent un rôle non négligeable sur les performances des applications.

Les deux principales techniques de programmation concurrente seront détaillées dans la suite de ce rapport, à savoir celle à base de processus légers (threads), et celle à base d'événements. Cette étude montrera que la programmation événementielle expose d'intéressantes propriétés de modularité et donc de reconfigurabilité.

Néanmoins, cette technique n'utilisant qu'un seul flot d'exécution, elle tire *a priori* difficilement parti des architectures parallèles. Il existe peu de solutions événementielles pour de telles architectures, et il se trouve que celles-ci ne présentent aucune optimisation au niveau des caches matériels.

Contribution

Notre contribution dans ce contexte concerne un support d'exécution flexible offrant un placement des traitants d'événements efficace sur les différentes unités d'exécution. Par placement efficace on entend un placement des traitants tirant parti des unités d'exécution offertes par l'architecture, mais aussi des caches de celle-ci.

Notre travail se découpe en trois étapes. Tout d'abord montrer les différences de comportement observées selon le type d'application au travers de certains placements statiques de traitants. Ensuite mettre en place une solution d'observation des métriques pertinentes à

[1]Institut National de Recherche en Informatique et en Automatique
[2]Laboratoire d'Informatique de Grenoble
[3]High Performance Computing

notre contribution, métriques que nous prendrons soin de définir. Enfin, étudier des techniques d'adaptation possibles pour décider dynamiquement et indépendamment de l'application du meilleur placement à adopter.

Organisation du document

Ce rapport est organisé de la manière suivante. Le premier chapitre est consacré à l'étude des techniques de programmation d'applications concurrentes, des caractéristiques matérielles présentes dans les architectures modernes, et des solutions existantes pour en tirer parti.

Le deuxième chapitre présente les manques de ces solutions existantes, ainsi que les objectifs de notre contribution, et l'évolution de l'architecture du modèle événementiel en multiprocesseur. Dans le troisième chapitre, nous décrivons la réalisation de son implantation, ainsi que les choix effectués. Enfin, nous exposons les résultats de nos expérimentations menées sur différentes applications.

Chapitre 1

État de l'art

Ce chapitre présente différentes techniques existantes pour concevoir des applications concurrentes efficaces, ainsi que les défis présentés par leur optimisation dans le contexte des architectures modernes. Deux techniques se distinguent notamment : la programmation à base de processus légers (threads) et la programmation événementielle. Nous verrons les avantages et les inconvénients de chacune de ces techniques.

Puis nous présenterons les principales caractéristiques à prendre en compte dans les architectures modernes, notamment la hiérarchie mémoire et la multiplicité des unités d'exécution. Enfin, nous verrons comment les deux styles de programmation abordés peuvent tirer parti de ces caractéristiques.

Terminologie Dans ce chapitre ainsi que la suite de ce rapport nous différencierons les termes *cœur* et *processeur*. Pour éviter toute confusion, nous emploierons le terme générique d'*unité d'exécution* pour désigner une unité matérielle minimale permettant d'exécuter un flot d'exécution. En revanche nous emploierons souvent par abus de langage le terme multiprocesseur pour désigner des architectures matérielles parallèles aussi bien multicœur que multiprocesseur, le terme architecture parallèle pouvant désigner un ensemble plus large de matériel.

1.1 Modèles de programmation

1.1.1 Modèle de programmation par threads

Le modèle de programmation concurrente le plus utilisé à l'heure actuelle est celui des processus légers, ou *threads* [17]. Pour éviter toute confusion avec les processus lourds (qui représentent les fils d'exécution ayant leur propre espace d'adressage), nous utiliserons par la suite le terme thread pour désigner les processus légers.

Principes

Pour programmer des applications concurrentes, le modèle à base de threads offre la possibilité de gérer plusieurs flots d'exécution au sein d'une seule application (un seul processus). Chaque thread correspond donc à un fil d'exécution. Un thread a sa propre pile d'exécution, même s'il partage la plupart de ses autres données avec le processus père. Pour donner l'impression de concurrence, le système d'exploitation, ou la bibliothèque utilisée (selon que le

thread soit de niveau noyau ou utilisateur), gère et ordonnance les threads en leur associant à chacun un quantum de temps. A chaque expiration de ce quantum, un nouveau thread prêt est élu pour être exécuté. Cela a pour effet d'entrelacer l'exécution des threads. Ce principe d'entrelacement est appelé *préemption*. Chaque unité d'exécution étant prévue pour gérer un fil d'exécution, cet entrelacement est présent sur chaque unité d'exécution disponible. C'est-à-dire que sur une machine ayant n unités d'exécution, n threads s'exécuteront simultanément sur la machine.

Avantages

Facilité de programmation Un thread offre au programmeur l'abstraction d'un flot d'exécution. Cela facilite grandement la programmation d'applications concurrentes, étant donné qu'à l'exception des accès aux données partagées, on n'a pas à se soucier des autres flots d'exécution présents sur la machine, ou même dans l'application.

Exploitation de l'architecture Un autre avantage remarquable dans ce modèle de programmation est qu'il permet de tirer parti des architectures parallèles. En effet, chaque thread correspondant à un fil d'exécution, il est facile pour le système d'exploitation (selon la bibliothèque de threads utilisée) de placer chaque thread sur une unité d'exécution différente, lorsque l'architecture le permet.

Inconvénients

Malgré ses avantages, le modèle à base de threads a toutefois des inconvénients non négligeables [13, 15].

Synchronisation Le programmeur n'a aucun contrôle sur la préemption de chaque thread. Même si cela facilite grandement le code des programmes multithread, cela a pour effet que le programmeur n'a aucun contrôle sur l'entrelacement des flots d'exécution. L'accès aux variables partagées d'un programme doit donc être protégé. C'est-à-dire que le programmeur doit s'assurer que les accès à ces variables sont effectués de façon atomique. Autrement dit, pour que le programme garde une certaine cohérence, un seul thread à la fois doit accéder à une variable partagée du programme. Cette synchronisation est laissée à la charge du programmeur, et est souvent source de problèmes. En effet, les mécanismes de synchronisation existants mènent souvent à des situations d'inter-blocage.

Coûts Outre les coûts importants que peuvent induire les primitives de synchronisation, un autre inconvénient des threads est leur difficulté à passer à l'échelle. En effet, lors d'un changement de contexte, le système d'exploitation doit sauvegarder l'état du thread courant (l'ensemble des registres, la pile, ...), et restaurer celui du nouveau thread à exécuter. Lorsque le nombre de threads est très grand, le temps passé à changer de contexte devient tellement important par rapport au temps d'exécution de chaque thread qu'on assiste à l'effondrement des performances du système. Ce phénomène est appelé écroulement ou *trashing*.

1.1.2 Modèle de programmation par événements

Les inconvénients de la programmation à base de threads ont conduit une partie des programmeurs à revoir leur choix de modèle de programmation concurrente [15]. Parmi eux,

certains ont opté pour le modèle événementiel.

Principes

Le modèle événementiel est axé autour d'une boucle de contrôle qui traite des événements. Dans ce modèle, l'application est découpée en traitants d'événements. Nous verrons plus loin les conséquences de cette structuration en différents traitants. L'impression de parallélisme est ici donnée par la succession d'événements traités. La boucle de contrôle est responsable de l'ordonnancement des événements. Ce modèle n'utilise qu'un seul fil d'exécution : celui qui est attribué à la boucle de contrôle. Les traitants d'événements sont exécutés sur ce même fil d'exécution.

La programmation par événements est notamment utilisée dans les mécanismes d'interfaces graphiques. Cette abstraction correspond assez bien au fait qu'une interface graphique réagit à des événements du monde extérieur, qui correspondent alors à un déplacement de souris, une frappe au clavier, etc. Cela permet en plus de garder un système réactif quelque soit le nombre d'événements ou de fenêtres à traiter, étant donné qu'on évite les changements de contexte (qui seraient présents si on utilisait une programmation à base de threads).

Pour que les applications utilisant ce modèle soient efficaces, il est préférable d'avoir des traitants courts et non bloquants. En effet, cela diminue la latence entre deux événements, et augmente ainsi la réactivité du système. Le fait d'utiliser le même fil d'exécution pour les traitants et la boucle de contrôle induit que si un traitant se bloque, sur une entrée/sortie (E/S) synchrone par exemple, alors la boucle de contrôle est aussi bloquée [18]. On assiste alors à une baisse considérable de réactivité dans le système. Par exemple, si l'unique fil d'exécution d'un serveur de données se retrouve bloqué par une E/S, le serveur se retrouve alors en état de refus de service le temps d'effectuer cette E/S.

On peut éviter cette difficulté en utilisant des E/S asynchrones, qui ne bloquent pas le traitant appelant, mais génèrent à la place un nouvel événement lorsque cette E/S est terminée. Ce nouvel événement est envoyé à la boucle de contrôle, qui appellera le traitant associé à la fin de l'E/S et à la suite du traitement.

Avantages

Comme nous l'avons vu, le modèle événementiel évite les changements de contexte en n'utilisant qu'un seul fil d'exécution (un seul thread global, en somme). Cela permet de résister à une haute charge, en comparaison du modèle de threads, où le nombre de threads (et donc de changements de contexte) augmente avec la charge. On n'observe donc pas avec un modèle événementiel le phénomène d'écroulement induit par ce grand nombre de changements de contexte.

Un autre avantage à n'utiliser qu'un fil d'exécution est de ne pas avoir à gérer de synchronisation dans le code des traitants, même s'ils accèdent à des variables partagées. En effet, la boucle de contrôle exécutant les traitants jusqu'à leur complétion, le modèle se garde de toute préemption. Cela permet d'éviter ainsi les problèmes d'accès concurrents à la mémoire.

Inconvénients

Cependant, ce modèle a également des limitations, à commencer par sa difficulté de programmation. En effet, la structuration de l'application en traitants d'événements cache toute forme de flot d'exécution. Or la plupart des programmeurs sont habitués à programmer de

façon séquentielle et cela complique significativement la compréhension des applications événementielles, et donc leur développement ainsi que leur maintenance. De plus, le passage d'informations entre traitants est rendu difficile du fait que chaque traitant retourne à la boucle de contrôle. En effet, le plus souvent, un traitant d'événements correspond à une fonction, et le fait que deux traitants successifs s'appellent de façon asynchrone ne permet pas à ces fonctions de s'appeler directement. Il est donc difficile pour des traitants de se passer des paramètres. Le contexte passé entre traitants est appelé *continuation.* Ayda *et al.* parlent de gestion de pile manuelle (ou *stack ripping*) [4] dans le cas de programmes événementiels.

Une autre limitation est que le modèle événementiel requiert l'utilisation d'E/S asynchrones, pour que les traitants ne soient pas bloquants. En effet, même si elles ne sont en théorie pas nécessaires, l'absence d'E/S non bloquantes induit des traitants bloquants, ce qui est très dommageable au niveau des performances.

On peut rajouter à cela que le fait de n'utiliser qu'un seul flot d'exécution ne permet pas à ce modèle de tirer parti efficacement des systèmes multiprocesseur. En effet, la distribution des tâches sur les différentes unités d'exécution est effectuée par le système d'exploitation. Il est donc nécessaire d'avoir plusieurs threads de niveau noyau pour garantir de répartir une application sur différentes unités d'exécution d'une machine. Ce qu'on n'a pas ici, étant donné que le modèle événementiel n'utilise qu'un flot d'exécution.

1.2 Architecture matérielle et impact des caches

Dans cette section, nous détaillons certaines caractéristiques intéressantes des architectures matérielles modernes [10], puis quelques solutions existantes qui tirent parti d'une des caractéristiques présentées : les caches.

1.2.1 Architectures modernes

Impact des caches

Les architectures modernes ont tendance à présenter des fonctionnalités toujours plus évoluées. Les processeurs étant de plus en plus rapides, souffrent de grandes latences d'accès à la mémoire centrale. En effet, l'accès à une donnée présente en mémoire centrale peut prendre de l'ordre de 200 à 300 cycles processeur, sachant qu'un cycle suffit pour effectuer une addition de deux valeurs contenues dans des registres. Même les techniques d'exécution dans le désordre ne suffisent pas à cacher ces latences d'accès à la mémoire. C'est dans cette optique qu'interviennent les mécanismes de caches.

Hiérarchie mémoire Un cache est une mémoire très rapide, que l'on place directement sur le processeur. Cependant, les coûts de fabrication des caches, ainsi que l'espace disponible sur les processeurs sont les facteurs limitants de leur présence. En effet, plus un cache est rapide, plus son coût de production augmente. C'est pourquoi la plupart des processeurs modernes ont plusieurs niveaux de caches.

Les caches de niveau L1 sont les plus proches des bancs de registres, les plus rapides (l'accès à un cache L1 se fait en 1 à 3 cycles environ) et aussi les plus petits (autour de 64Ko). Ils sont souvent séparés, c'est-à-dire qu'on trouve un cache L1 réservé aux instructions et un cache L1 réservé aux données.

Les caches de niveau L2 sont plus éloignés, un peu moins rapides (l'accès à un cache L2 se fait en moins d'une dizaine de cycles) mais plus gros (entre 2Mo et 4Mo). Ils sont souvent unifiés, c'est-à-dire que les instructions et les données sont réunies dans la même mémoire.

Accès à la mémoire Étant donné leur petite taille en comparaison de la mémoire centrale, les caches contiennent rarement toutes les données et instructions nécessaires à l'exécution d'un programme. On appelle *faute de cache* le fait de ne pas trouver la donnée (ou l'instruction) dans le cache d'un certain niveau. L'accès à une donnée (ou une instruction) se fait donc comme suit :

1. Si la donnée est dans un registre, alors on y accède directement (en un cycle).
2. Sinon, on la cherche dans le cache L1.
3. Si elle n'est pas présente dans ce cache, il y a une faute de cache L1, et on sollicite le cache L2.
4. De même, si elle n'est pas dans le cache L2, il y a une faute de cache L2 et on va chercher la donnée dans la mémoire centrale.

Pour pouvoir accéder de cette façon à une donnée, le contenu des caches doit toujours être cohérent avec celui de la mémoire centrale. Comme le système d'exploitation ordonnance les processus en utilisant un mécanisme de préemption, certaines architectures vident leurs caches lors d'un changement de contexte. Ceci est fait pour deux raisons : premièrement par souci de cohérence. En effet si les caches sont indexés avec des adresses physiques, il se peut qu'une même adresse physique soit en fait la cible de deux adresses virtuelles différentes selon le contexte. Deuxièmement les caches sont également vidés par mesure de sécurité, pour éviter qu'un programme accède aux valeurs d'un autre par ce biais. Il faut donc des politiques de remplacement des valeurs contenues dans les caches. L'algorithme le plus communément adopté pour cette tâche est celui dit LRU[1]. Le principe du LRU est de remplacer la valeur la moins récemment utilisée par celle que l'on veut faire entrer en cache.

Apport des caches Cela a pour conséquence que les programmes présentant une certaine *localité* ont plus de chances de trouver leurs données en cache. Il existe deux types de localités : la localité spatiale et la localité temporelle. La fait qu'un accès à une adresse en mémoire entraîne un accès sur une adresse voisine est appelé localité spatiale. La localité temporelle est, quant à elle, le fait qu'une adresse récemment accédée le sera à nouveau sous peu.

Multiples unités d'exécution

Dans le but d'afficher des performances toujours meilleures, les fréquences d'horloges sont toujours plus hautes et les circuits des architectures modernes sont gravés de plus en plus finement. Il en résulte des circuits de plus en plus denses et rapides. Cependant, de gros problèmes rencontrés par les constructeurs de processeurs sont liés aux fuites électriques ainsi qu'à la dissipation thermique de ces circuits, qui deviennent ingérables avec des gravures trop fines. Réduire la taille des circuits va donc bientôt devenir impossible avec l'accélération des horloges et les technologies de gravure et de conception de circuits actuelles.

Une approche pour continuer de gagner en performance est alors de dupliquer les unités d'exécution. Par unité d'exécution on entend unité matérielle capable de gérer entièrement un

[1] Least recently used, moins récemment utilisé

flot d'exécution (un thread par exemple). Dans cet approche deux solutions sont possibles : soit dupliquer les processeurs sur la carte mère, soit dupliquer les cœurs au sein d'un processeur. Dans la suite nous détaillons ces deux solutions.

Systèmes multiprocesseur Par système multiprocesseur on entend architecture comprenant deux processeurs (CPU[2]) ou plus. Il existe des systèmes multiprocesseur dans lesquels les processeurs ne sont pas tous égaux. Parmi ces systèmes où les ressources ne sont pas partagées équitablement, on peut noter les architectures ASMP[3] et NUMA[4]. Cependant, la répartition équitable des ressources étant plus répandue, nous nous concentrerons ici sur les architectures SMP[5]. Dans de tels systèmes, les processeurs sont tous égaux, autant du point de vue des accès à la mémoire que de leur puissance de traitement.

Comme chaque processeur présent dans le système est identique aux autres, il a la même quantité de cache, répartie de la même façon sur les différents niveaux (cf. Figure 1.1). Cela permet de raisonner facilement sur le placement des tâches, car une tâche placée sur un processeur va avoir le même comportement que si elle était placée sur un autre processeur (dans l'hypothèse où les processeurs concernés aient une activité identique). Dans la majorité des cas, chaque processeur dispose de deux caches de niveau L1 : un dédié aux instructions et un réservé aux données ; ainsi que d'un cache unifié (instructions et données) de niveau L2. On n'a donc aucun partage de cache entre les processeurs dans ce type d'architecture.

Il est cependant possible d'obtenir un partage de cache entre processeurs dans certains cas. En effet, certaines carte mères sont équipées d'un cache de niveau L3, souvent commun à tous les processeurs. Cette situation n'étant pas courante, nous nous concentrerons ici sur des problématiques d'optimisation des caches L1 et L2.

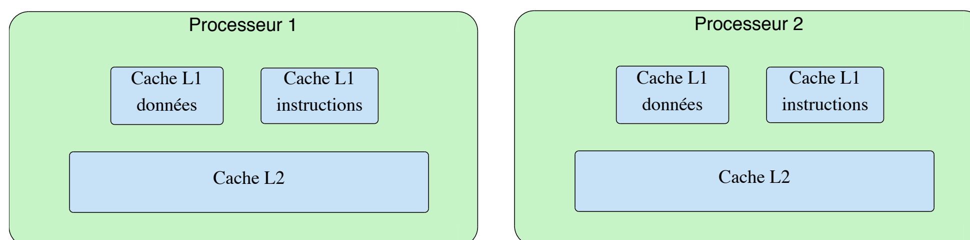

Fig. 1.1 – Exemple d'architecture multiprocesseur

Systèmes multicœur Dans les systèmes multicœur , un processeur est vu comme un ensemble de cœurs . Chaque cœur pouvant gérer indépendamment des autres un flot d'exécution. Une particularité intéressante de ces architectures est que dans la plupart des cas, chaque cœur a ses propres caches de niveau L1, mais le cache de niveau L2 reste commun à tout le processeur (cf. Figure 1.2).

On peut noter qu'il est possible de coupler des architectures multicœur et multiprocesseur. On obtient alors un système comme décrit sur la figure 1.3. Il faut alors bien faire attention

[2]Central Processing Unit, unité de traitement principale
[3]ASymmetric MultiProcessing, multitraitement asymétrique
[4]Non-Uniform Memory Accesses, accès mémoire non uniformes
[5]Symmetric MultiProcessing, multitraitement symétrique

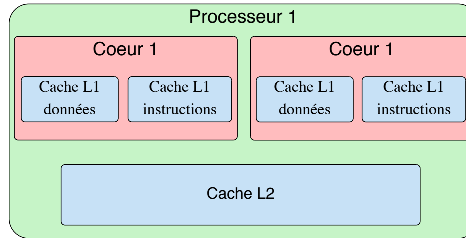

Fig. 1.2 – Exemple d'architecture multicœur

à ce que tous les cœurs ne sont plus totalement égaux les uns aux autres, étant donné que certains (qui sont sur le même processeur) partagent leur cache L2, et d'autres non. On peut donc imaginer qu'un échange de données sera plus efficace entre des cœurs appartenant au même processeur, dans l'hypothèse que ces données ne dépassent pas la capacité du cache L2.

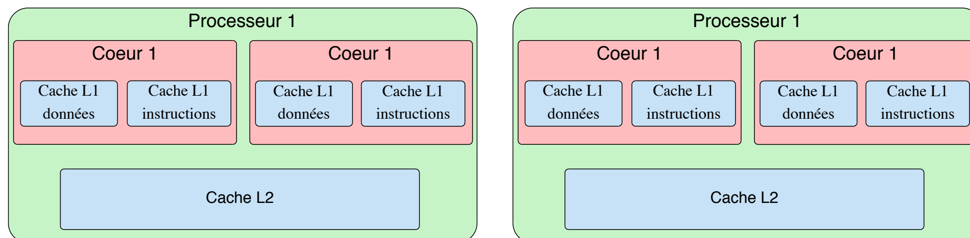

Fig. 1.3 – Exemple d'architecture multicœur et multiprocesseur

1.2.2 Optimisations liées aux caches

Certains aspects de ces nouvelles architectures, notamment la hiérarchie mémoire, ont conduit à chercher des optimisations dans les différents types d'applications existantes.

Opportunités d'ordonnancement sensible aux caches

Koka *et al.*[11] font une caractérisation détaillée de différents types de fautes de caches. Leur but est à terme de réduire les fautes de cache dues aux changements de contexte. Pour cela, des heuristiques d'ordonnancement internes au noyau du système d'exploitation sont proposées. Leur stratégie d'ordonnancement est découpée en deux parties.

Le premier travail réalisé est de déterminer l'espace de travail des processus à ordonnancer. Pour ce faire, leur idée est d'inférer ceci à partir des informations contenues dans la pile du processus. Lors d'un changement de contexte, l'ordonnanceur système peut produire une signature à partir de l'adresse de retour, des arguments et des variables locales. L'adresse de retour fournit des informations sur la phase dans laquelle se trouve une application. Les arguments et les variables locales peuvent exhiber des partages de données. Koka *et al.*proposent de générer une telle signature avec une technique de hachage simple.

A partir de cette signature, qui peut être stockée dans l'état du processus, l'ordonnanceur peut alors décider d'élire un processus ayant une signature proche de celui laissant la main. Les similarités entre signatures peuvent être calculées simplement avec une distance de hamming.

Cela devrait permettre d'améliorer les accès aux caches des applications multithread, dont chaque thread utilisateur s'exécute sur un thread noyau. Cependant pour le moment aucune implémentation de cette technique n'est connue, donc aucune évaluation n'en est disponible actuellement.

Contrôle dynamique de ressources

Welsh *et al.*[19] proposent une plateforme pour des services internet efficaces, nommée SEDA. Dans cette optique, ils adoptent une architecture à étages. Chaque étage est un traitant d'événement qui est exécuté par un groupe (ou *pool*) de threads. On peut noter l'utilisation de contrôleurs dynamiques de ressources. Il y a deux types de contrôleurs : l'un pour modifier dynamiquement le nombre de threads par étage, et l'autre pour déterminer l'ordonnancement des événements au sein d'un thread.

Ce dernier a pour but d'augmenter la localité de cache de l'application. Pour cela, la méthode est de grouper l'exécution de certaines tâches, cette technique est appelée *batching*. Ce contrôleur adapte dynamiquement le nombre de tâches dont l'exécution va être regroupée. Le batching permet surtout de profiter des caches d'instructions (sachant que la même tâche va être lancée plusieurs fois de suite). Cette méthode permet également de tirer parti des caches de données, même si les paramètres d'appels diffèrent, dans la mesure où une tâche utilise des variables locales et globales.

Conception liée de gestionnaire mémoire et d'ordonnanceur

Dans le but de tirer parti des caches pour optimiser les serveurs de données événementiels, Bhatia *et al.* [5] proposent de concevoir un allocateur mémoire allant de pair avec les ordonnanceurs de ces serveurs. Leur gestionnaire de mémoire est capable d'allouer des zones mémoire couplées directement dans les caches matériels (généralement le cache de niveau L2). Ayant remarqué que la plupart des serveurs de données se comportent de façon déterministe, leur idée est qu'il est possible de calculer l'empreinte mémoire d'un serveur. À partir de là, l'objectif est de calculer combien d'événements peuvent être placés directement en cache en fonction de leur taille et de leur durée de vie, pour les traiter en priorité.

Analyse statique Après annotation des accès mémoire dans le code du serveur, une phase d'analyse statique a lieu. Cette analyse permet de déterminer approximativement :
- la place occupée par le serveur dans la pile
- la quantité de mémoire allouée et désallouée pour traiter une requête
- la quantité de mémoire utilisée globalement dans le serveur

Allocateur mémoire avare L'étape suivante a pour but de générer une configuration de l'allocateur mémoire correspondant au code analysé. Cet allocateur est destiné à gérer un petit nombre d'instances d'objets, et de garantir que tant qu'un programme utilise seulement ces objets, ils ne vont pas interférer entre eux et seront tous présents en cache.

Ordonnanceur associé L'ordonnanceur associé à l'allocateur mémoire va assurer que lors de son exécution le programme appellera les traitants d'événements de façon à ne pas dépasser la capacité du cache. Pour mieux comprendre le principe, on prend l'exemple d'un serveur événementiel à 5 traitants, en prenant en considération que certains traitants allouent des objets

pour une certaine durée de vie (cf Figure 1.2.2.a). Le comportement en cache du programme est représenté dans le cas où l'ordonnanceur tente d'exécuter 4 requêtes simultanément. On voit que l'allocation des objets nécessaires à ce programme dans cet ordonnancement dépasse la taille du cache (cf Figure 1.2.2.b).

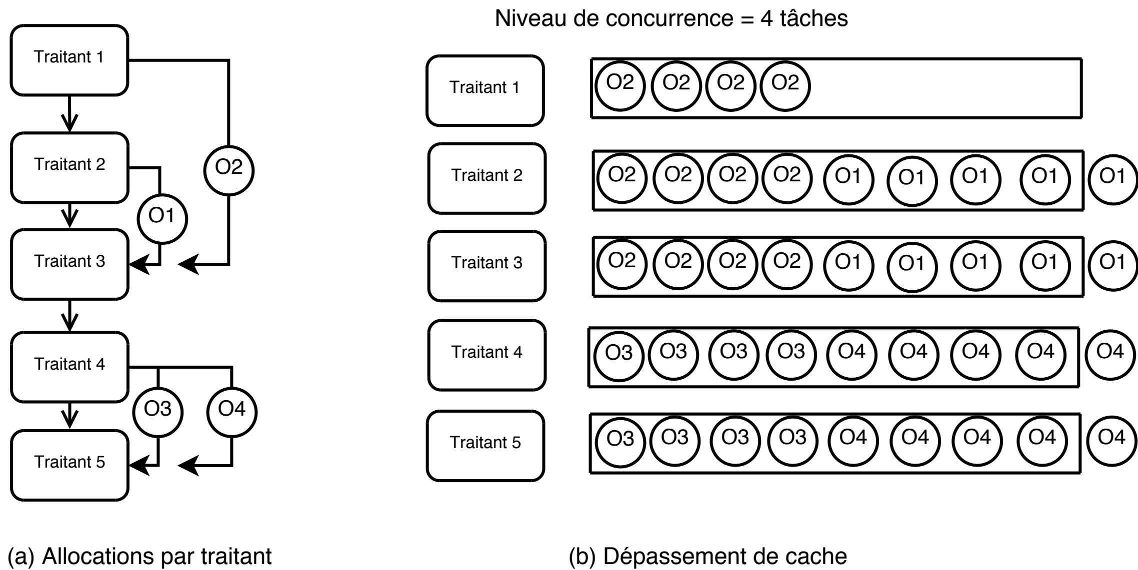

FIG. 1.4 – Etat du programme pendant son exécution. Cette figure est tirée de l'article de Bhatia *et al.*[5]

C'est ce problème que Bhatia *et al.* résolvent ici en modifiant l'ordonnanceur pour le faire fonctionner de pair avec l'allocateur avare. Ainsi, pour décider du prochain traitant à exécuter, l'ordonnanceur va faire appel à l'allocateur avare pour savoir s'il peut lancer une nouvelle requête ou bien s'il doit d'abord avancer le traitement d'une requête en cours pour libérer de l'espace dans le cache. De cette façon, cette optimisation garantit, lorsque cela est possible, que les données accédées par l'application seront disponibles en cache.

Cependant cette optimisation requiert une lourde intervention du programmeur, d'une part pour annoter le code concernant les allocations mémoire, et d'autre part pour apporter des modifications à l'ordonnanceur. On peut aussi noter que le déterminisme de l'application est fortement affecté dans un environnement multiprocesseur. Pour l'instant ce type d'optimisation n'est donc pas applicable à ces environnements.

1.3 Supports d'exécution adaptés aux architectures multiprocesseur

Nous allons maintenant détailler des solutions existantes d'environnements d'exécution ayant été portées dans des contextes multiprocesseur. Ces exemples nous intéressent par le fait qu'ils n'utilisent qu'un seul thread par unité d'exécution, ce qui limite autant que possible les changements de contexte et leurs effets néfastes sur les caches matériels.

1.3.1 SMP Click

SMP Click[7] est une extension de Click[14], une boîte à outils implémentée en C++ pour construire des routeurs de paquets réseau. Le fait est que les routeurs modernes présentent des fonctionnalités évoluées. Faire le choix de créer des routeurs purement matériels finit par se révéler coûteux, les circuits devenant plus complexes. C'est pourquoi une partie de la communauté se tourne vers des routeurs logiciels, présentant plus de flexibilité. Click a été créé dans l'optique de faciliter ce développement. Cependant, le principal problème des routeurs logiciels étant la performance, les créateurs de Click se sont vite tournés vers une solution tirant parti des architectures multiprocesseur.

Click

Les routeurs Click sont construits de façon modulaire à partir d'*éléments* reliés entre eux par des *connexions*. Les connexions relient les éléments par leurs ports. Chaque élément a donc des ports de sortie et des ports d'entrée. Pour chacun de ces types de ports, il existe deux mécanismes de communication : *push* et *pull*. Dans le cas des ports de type push, un message est généré par l'envoyeur et passé directement au receveur. A l'inverse, sur les ports de type pull, c'est le receveur qui demande un message par son port d'entrée. Une connexion ne peut exister qu'entre un port de sortie et un port d'entrée de même type.

Chaque file de message dans un assemblage Click correspond à un point de découplage dans l'architecture et est placée explicitement par le programmeur. Cela permet d'avoir des configurations dans lesquelles un élément *file* peut dispatcher ses messages sur plusieurs éléments. La figure 1.5 montre un exemple d'assemblage d'éléments. Click ne suit pas un modèle événementiel. En effet, les appels entre éléments se font de manière synchrone, donc tous les éléments ne retournent pas forcément à la boucle de contrôle.

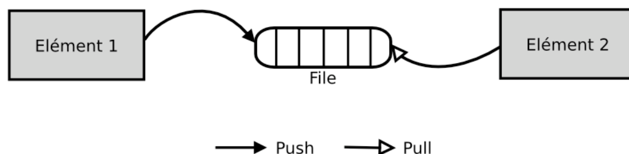

Fig. 1.5 – un exemple de connexion avec Click

SMP Click

Pour tirer parti efficacement des architectures multiprocesseur, l'idée de SMP Click est de lancer un thread par unité d'exécution disponible. Chaque thread a une liste d'éléments à exécuter. Cette liste contient uniquement des éléments d'un certain type, qui correspondent en fait à des débuts de cheminement de messages. Ces éléments vont par exemple scruter un périphérique (*PollDevice*), scruter un port pull pour passer les messages sur un port push (éléments *PullToPush*) ou bien scruter un port pull pour envoyer les messages vers un périphérique (*ToDevice*).

Lorsqu'un thread commence à exécuter un de ces éléments pour un paquet, ce thread exécute alors tout le cheminement relatif à ce paquet, jusqu'à arriver sur un élément file, *ToDevice*, ou tout autre élément qui abandonne ce paquet. Des unités d'exécution exécutant

des chemins disjoints ne vont pas interférer entre elles. À l'inverse, des chemins se rejoignant vont entrer en conflit au point de jointure. Chaque instance d'élément dans SMP Click peut être exécutée simultanément sur plusieurs unités d'exécution. Pour gérer cela, SMP Click demande au programmeur de protéger l'état global de chaque élément avec des structures de synchronisation.

SMP Click propose deux façons de répartir les éléments sur les unités d'exécution. Cela peut être laissé à la charge du programmeur et alors fait de manière statique (en affectant à chaque élément une unité d'exécution). L'autre solution étant d'adopter une répartition dynamique. SMP Click fournit en effet un algorithme de répartition dynamique de charge, calculée à partir du coût de chaque élément. Pour ce faire, le coût de chaque élément est périodiquement mesuré. Cet échantillonnage engendre un surcoût négligeable.

1.3.2 libasync-mp

Libasync-smp[20] est une extension de Libasync[8], une bibliothèque UNIX en C++ qui permet de simplifier la réalisation d'applications événementielles. Comme nous l'avons vu, un des problèmes majeurs dans la conception d'applications événementielles est la difficulté de programmation qu'elles représentent. En effet, ces programmes ont la forme d'une boucle de contrôle appelant des traitants d'événements. Outre le fait que l'exécution d'une succession de traitants peut être extrêmement difficile à appréhender, il est également à la charge du programmeur de s'assurer que le contexte d'événement est propagé entre chaque traitant. Pour pallier ces difficultés, libasync fournit certaines facilités d'écriture.

Libasync

Tout d'abord, une boucle de contrôle générique est implémentée. Elle est accompagnée de fonctions permettant l'enregistrement et la suppression de couples (événement,traitant). Cela décharge le programmeur d'une tâche fastidieuse et répétitive. La conception modulaire et la réutilisation de code sont encouragées par le fait que plusieurs modules peuvent utiliser libasync sans partager d'informations. L'utilisation de modules prédéfinis (tels que les modules DNS, par exemple) allège d'autant plus le nombre de traitants utilisés et facilite ainsi la lisibilité du code.

Toujours dans le but d'alléger l'écriture du code, un mécanisme de gestion automatique de la mémoire est mis en place. Un ramasse-miettes à base de comptage de références est utilisé dans le code de Libasync, notamment pour gérer l'allocation mémoire des événements. L'utilisation de ce ramasse-miettes est permise à travers l'interface de la bibliothèque.

Le passage de contexte entre les traitants est également simplifié par l'utilisation d'une fonction d'encapsulation appelée *wrap*. Cette fonction a pour effet de spécifier que l'on veut toujours passer un paramètre à une autre fonction. Par exemple, si on pose *w=wrap(fn,x)*, alors un appel de la forme *w(arg)* aura pour effet d'exécuter *fn(x,arg)*. Cela permet aux traitants de se passer facilement des paramètres, étant donné qu'un traitant est souvent implémenté sous la forme d'une fonction, l'envoi d'un événement se fait donc par l'appel asynchrone d'une fonction.

Libasync-smp

Pour tirer parti des environnements multiprocesseur, l'idée des auteurs est de demander au programmeur d'associer à chaque événement une couleur (et une seule). Par la suite, le but est

d'assurer que deux événements de la même couleur ne peuvent s'exécuter simultanément. Cela correspond à une exclusion mutuelle à gros grain (au niveau de l'événement), et ne permet donc pas de synchronisation plus fine. Cependant les auteurs estiment que cette solution suffit dans la majorité des cas. Les événements dont la couleur n'est pas spécifiée se voient attribués une couleur par défaut. De ce fait, si on ne spécifie pas de couleur, tous les événements auront la couleur par défaut et donc s'exécuteront séquentiellement. Le but de ce comportement est de conserver la compatibilité avec libasync, et de permettre au programmeur de paralléliser son application de manière incrémentale.

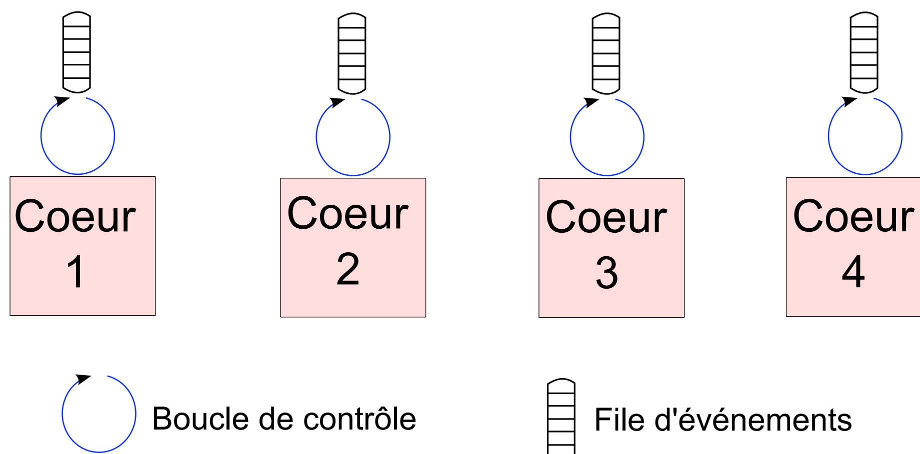

FIG. 1.6 – Architecture de libasync-mp

Libasync-smp décompose l'application en N threads, où N est le nombre d'unités d'exécution vues par le système d'exploitation (cf Figure 1.6). Chaque thread exécute une boucle de contrôle et possède une file d'attente d'événements. L'événement de couleur c est placé dans la file du thread $c \ mod \ N$. Cela permet une répartition de charge à peu près égale entre les unités d'exécution, tout en garantissant que tous les événements de même couleur seront exécutés au sein du même thread, et donc séquentiellement. Une optimisation est faite au niveau de l'utilisation des caches. Elle consiste à faire en sorte que l'ordonnanceur privilégie, pour chaque file, les événements de même couleur que celle que le thread vient d'exécuter. Ainsi, les variables partagées par les traitants d'événements de même couleur sont réutilisées dès que possible.

Pour mieux équilibrer la charge au sein des N unités d'exécution, un thread ayant une file vide peut s'approprier des événements dans les files des autres threads, on parle alors de vol de tâches, ou *workstealing*. Pour cela, il doit également voler tous les autres événements de même couleur dans la file du thread à qui il les récupère, de manière à ne jamais permettre l'exécution simultanée de deux événements de même couleur. De ce fait, un thread ne peut pas voler un traitant dont la couleur est en cours d'exécution. Quand un vol de couleur a lieu, tous les événements concernés seront redirigés vers le thread voleur.

La principale limitation de Libasync-mp est que le vol de tâches ne tient aucun compte des localités de caches des événements volés. Le fait est qu'un événement en cours d'exécution a de fortes chances d'avoir les données et/ou les instructions avec lesquelles il travaille sur l'unité qui l'exécute. Le vol de tâches déplace des événements d'une unité d'exécution à une autre, du coup le contenu des caches de la première unité d'exécution relatif à ces événements

est rendu inutile et donc se retrouve à ”polluer” ces caches, qui pourraient être plus largement utilisés par les événements restants. De plus, l'unité d'exécution voleuse n'a aucune chance d'avoir ces données dans ses caches, ce qui va engendrer autant d'accès à la mémoire centrale.

1.4 Synthèse

L'objectif de ce chapitre était de présenter les deux techniques majeures de programmation concurrente, les nouvelles tendances abordées par les architectures actuelles, les défis d'optimisation que cela présente et enfin comment ces défis ont été relevés jusqu'à maintenant. Les deux techniques majeures de programmation concurrente sont la programmation à base de threads et la programmation événementielle. On peut noter qu'il existe plusieurs modèles hybrides, rejoignant threads et événements, tels que SEDA [19], StagedServer [12], Flash [16], l'ordonnancement coopératif avec gestion de pile automatique [4], etc. Mais la généralisation de nos travaux à ces modèles fera l'objet d'une étude ultérieure.

La programmation à base de threads est la plus utilisée aujourd'hui, du fait de sa simplicité d'écriture. Elle permet de tirer parti des architectures parallèles. Toutefois, la gestion de la synchronisation est une lourde tâche laissée au programmeur, et les faibles performances observées à haute charge sont des inconvénients non négligeables.

La programmation événementielle offre une solution à certains de ces problèmes. L'utilisation d'un seul flot d'exécution éliminant toute forme de synchronisation et offrant une dégradation gracieuse des performances (pas de phénomène d'écroulement). Cependant, elle ne tire pas efficacement parti des architectures parallèles.

Nous avons également introduit les enjeux liés aux architectures modernes, notamment l'importance d'exhiber une certaine localité de cache. Les optimisations existantes au niveau des caches sont très souvent axées autour d'un ordonnancement particulier. Selon le niveau auquel on se place (système d'exploitation, bibliothèque utilisateur ou bien application) et le type de localité que l'on cherche à exhiber, les informations disponibles et les possibilités d'optimisation diffèrent.

Nous avons vu que les plate-formes événementielles fonctionnant en multiprocesseur étaient rares. De plus, malgré les travaux existant pour tirer parti des effets de cache et optimiser les plate-formes événementielles, on remarque que ces travaux ne sont pas ou peu appliqués aux architectures parallèles.

Chapitre 2

Contribution

Dans ce chapitre, nous présenterons notre contribution qui consiste à améliorer les supports d'exécution événementiels en multiprocesseur afin de les rendre dynamiquement adaptables.

Par dynamiquement adaptables, on entend des supports d'exécution permettant de replacer les traitants d'une application en cours d'exécution, afin d'améliorer ses performances. La figure 2.1 montre l'architecture que nous proposons.

Dans ce chapitre, nous verrons sur quelles informations se baser et comment récupérer ces informations pour replacer des traitants (détail des sondes). Puis, nous étudierons différentes décisions de replacement, ainsi que les techniques d'application de ces décisions, autrement dit comment replacer des traitants dynamiquement (détail des actionneurs).

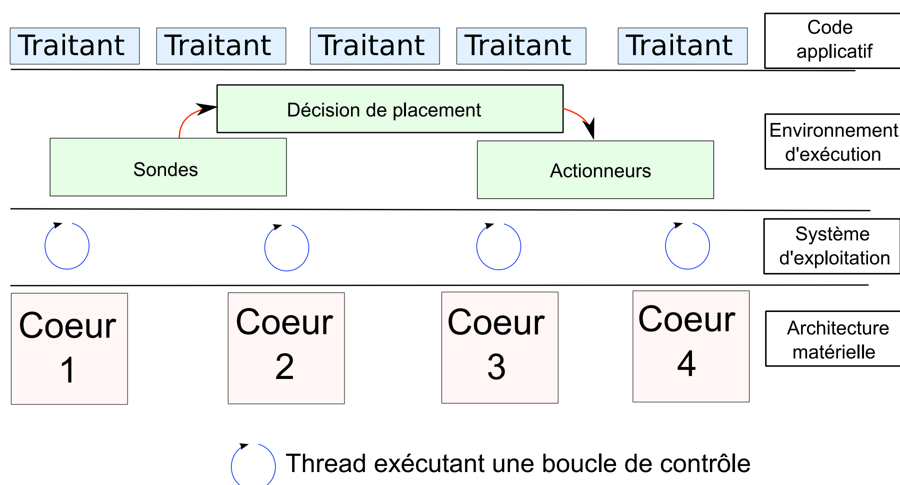

FIG. 2.1 – Vue d'ensemble de notre contribution

2.1 Vers des supports d'exécution plus flexibles

La programmation événementielle est un moyen efficace de concevoir des applications résistantes à une charge importante. Nous nous intéresserons ici à ce modèle de programmation dans des environnements multiprocesseur.

Nous pensons que les supports d'exécution basés sur ce modèle gagneraient à être plus flexibles. En effet, la répartition des traitants d'événements sur les unités d'exécution disponibles se fait indépendamment du type de charge induite par chaque traitant. On pense ici au mécanisme de vol de tâches de Libasync-mp, qui comme nous l'avons vu, répartit les événements de couleurs différentes sur les unités d'exécution. Lorsqu'un vol est effectué, cela entraîne une migration des événements concernés, et donc une perte des données en cache.

Il se trouve que des applications dont les traitants sont intensifs au niveau de la mémoire auraient de meilleures performances s'il pouvaient bénéficier d'une certaine localité de cache. Il n'est donc à priori pas judicieux de les faire migrer lors de leur exécution. Cependant, le mécanisme de vol de tâches présent dans Libasync-mp n'est pas sans avantages. En effet, dans le cas d'applications intensives au niveau processeur, les gains de la parallélisation sont indéniables.

C'est pourquoi nous pensons que les environnements d'exécution doivent s'adapter aux applications. De la même manière, ce sont aux environnements d'exécution de s'adapter aux architectures matérielles sous-jacentes, et ce de façon transparente du point de vue de l'application. En particulier le nombre d'unités d'exécution ne devrait pas être connu de l'application, ce qui devrait lui permettre d'être portable et efficace sur différentes machines. La répartition des traitants de l'application sur les unités d'exécution devrait être faite au niveau de l'environnement d'exécution.

Notre objectif est donc de proposer une architecture permettant de placer les traitants d'événements sur les unités d'exécution disponibles de manière efficace. Ce placement doit être calculé de façon dynamique, étant donné que le type de traitement effectué par une application peut changer au cours du temps. Cela est vérifié notamment dans le cas des serveurs de données, où par exemple on ne peut pas prévoir statiquement à quel moment une connexion sécurisée va être préférée à une connexion non sécurisée. Sachant que la sécurisation d'une connexion entraîne des traitements supplémentaires non négligeables, on peut penser que les placements optimaux des traitants ne seront pas les mêmes selon le type de charge demandé. Un autre exemple est que la taille des données échangées entre le serveur et les clients peut changer de façon non prédictible. Le fait que ces données dépassent ou non la capacité des caches affecte le placement optimal.

Le travail présenté dans ce rapport est basé sur l'implémentation de Libasync-mp. Nous présenterons, dans la suite du rapport, les modifications apportées à cette implémentation. Ces premières modifications ont pour but de déterminer quelles métriques et quelles heuristiques priment pour l'adaptation dynamique du support d'exécution. L'objectif étant, à terme, d'intégrer ces travaux au modèle à composants Fractal[2], notamment à son implémentation en C, *Cécilia*[1]. En effet, ces travaux seront réutilisés dans le cadre de projets de recherche en cours qui mettent en scène des MPSoC[1] dans laquelle la programmation événementielle a été choisie car elle reflète assez bien le comportement du matériel sous-jacent.

2.2 Architecture proposée

Dans cette section, nous présenterons les différentes stratégies de placements sur les ressources d'exécution. Nous proposerons ensuite une évolution de l'architecture événementielle en multiprocesseur permettant la reconfiguration automatique du placement des traitants lorsque cela est nécessaire.

[1]MultiProcessor System-on-Chip, système multiprocesseur sur puce.

Cette architecture, basée sur Libasync-mp, exécute un thread par unité d'exécution. Il faut cependant s'assurer que l'ordonnanceur du système d'exploitation ne replace pas ces threads[2]. Nous fixons donc chaque thread de notre architecture sur une unité d'exécution, de manière à assurer qu'une fois créé, un thread s'exécutera toujours sur la même unité d'exécution. Ceci est réalisé avec des fonctions spécifiques de la bibliothèque de threads NPTL[3][3].

Une de nos hypothèses de travail actuelles est que chaque thread ordonnance les événements d'un traitant de façon FIFO[4].

2.2.1 Stratégies de placements

On considère une application en forme de pipeline, c'est une hypothèse assez forte, mais acceptable dans notre contexte, car la grande majorité des serveurs de données événementiels ont cette forme. C'est-à-dire que chaque traitant a un et un seul traitant suivant. La question est alors de savoir comment placer la suite des traitants sur les différentes unités d'exécution.

Dans un pipeline, on peut noter globalement trois stratégies différentes :

1. La première vise à tirer parti des ressources d'exécution des systèmes multiprocesseur. L'idée étant que deux traitants qui se suivent dans le pipeline ne soient pas placés sur la même unité d'exécution. Si on prend l'exemple d'un pipeline comprenant 16 traitants, avec une architecture à 4 unités d'exécution[5], on obtient le placement suivant, également décrit par la figure 2.2 (la notation utilisée est la suivante : le i-ème numéro est celui de la ressource d'exécution attribuée au i-ème traitant du pipeline, parcouru de gauche à droite) :

placement round robin : 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4

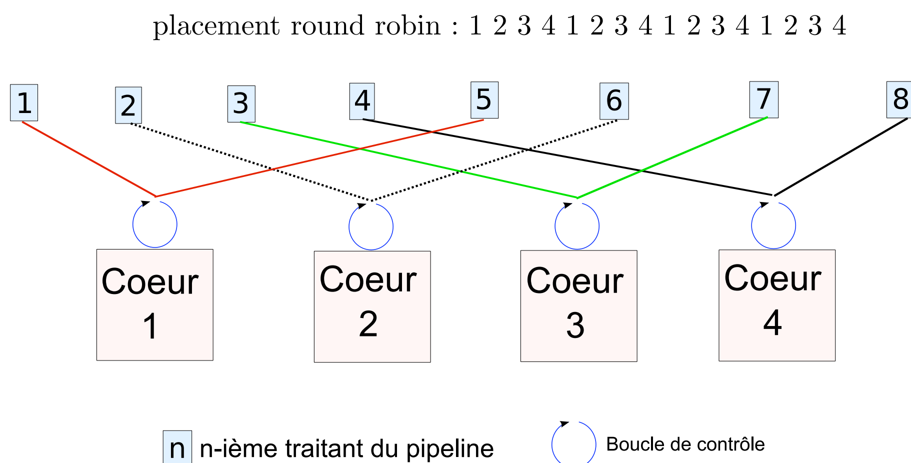

FIG. 2.2 – Le placement round robin

2. Une deuxième solution consiste à vouloir tirer parti des caches. Pour cela, on va placer deux traitants consécutifs sur la même unité d'exécution. En reprenant notre exemple

[2]La version récupérée de Libasync-mp ne faisait pas cette vérification
[3]Native POSIX Thread Library
[4]First In First Out, premier arrivé premier servi.
[5]Il s'agit ici d'une architecture bi-processeur, où chaque processeur est double-cœur . les cœurs 0 et 1 sont sur le même processeur et partagent ainsi leur cache L2, mais ont un cache L1 privé. Il en va de même pour les cœurs 2 et 3.

et notre notation, on obtient la suite :

placement sensible aux caches : 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4

On remarque que ce placement n'est pas forcément efficace (a priori) du point de vue de la parallélisation des tâches, étant donné que la totalité de la charge entrante se trouve sur une seule unité d'exécution, et que le deuxième traitant est placé sur cette même unité. Plus précisément, la politique d'ordonnancement des messages FIFO nous oblige dans ce cas là à traiter tous les messages avant de passer au traitant d'après. Ce comportement est une conséquence du placement de traitants consécutifs sur la même unité d'exécution, et est donc valable sur tout le pipeline, pas seulement au premier niveau.

3. Enfin, nous avons pensé à une solution mixte, où l'idée est de tirer parti à la fois des caches et des ressources d'exécution. Pour cela, nous faisons en sorte que deux traitants consécutifs soient sur deux unités d'exécution différentes, mais qui ont un cache partagé. L'idée est de découper le pipeline en séquences de même longueur. Chaque séquence se voit attribuer les deux cœurs d'un même processeur, et deux séquences successives sont couplées sur deux processeurs différents. Ainsi, on met en avant une localité de cache L2 au sein d'une séquence, tout en gardant un certain degré de parallélisme dans le pipeline. De plus, la parallélisation est améliorée par le fait qu'au sein d'une séquence, les deux cœurs disponibles alternent successivement. Les placements correspondant sont les suivants :

placement intermédiaire 1 : 1 2 1 2 1 2 1 2 3 4 3 4 3 4 3 4

placement intermédiaire 2 : 1 2 1 2 3 4 3 4 1 2 1 2 3 4 3 4

Remarque Bien entendu, tous les placements proposés tirent parti des ressources d'exécution lorsque le système est en régime permanent, étant donné que toutes les unités d'exécution sont utilisées. Cependant, il est intéressant de noter que les placements proposés ici ne sont pas tous égaux en ces termes face à une charge variable. En effet, le fait qu'un événement entrant dans le pipeline reste sur la même unité d'exécution le temps de plusieurs traitants n'a pas le même effet que si chacun de ses traitants associés était sur une unité d'exécution différente. Cela vient du fait que chaque unité d'exécution traite ses événements de manière FIFO.

2.2.2 Détail de l'architecture

Comme nous l'avons vu dans Libasync-mp, seule la notion d'événement existe. Dans le but de pouvoir raisonner sur des traitants d'événements, nous avons modifié l'architecture initiale. Ainsi, comme le montre la figure Fig 2.3, nous avons introduit la notion de traitant d'événement. Chaque traitant a alors sa propre file d'événements. On conserve le fait d'avoir un thread par unité d'exécution, vu que cela limite le nombre de changements de contexte. Chaque unité d'exécution va alors chercher la prochaine tâche à exécuter en parcourant les traitants qui lui sont associés selon un algorithme round robin. C'est une méthode simple pour garantir l'équité entre les traitants. D'autre part, nous avons enlevé toute notion de couleur

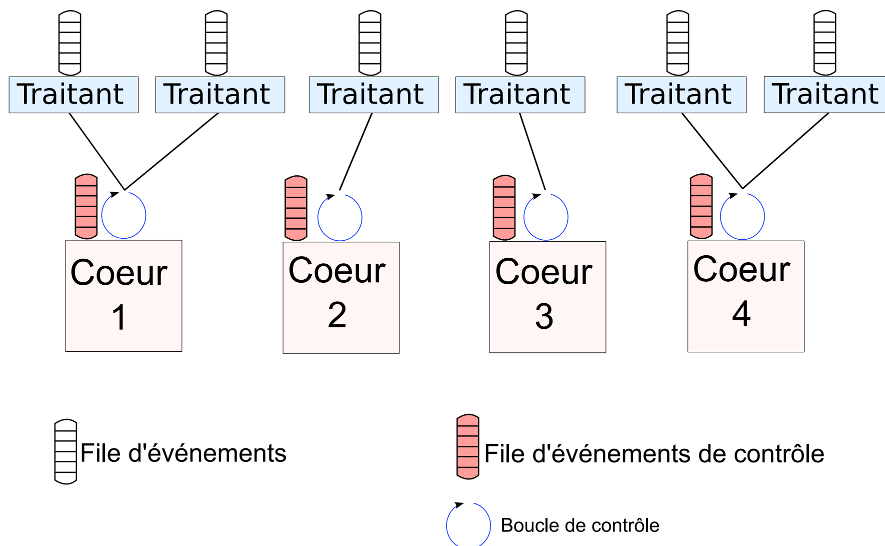

FIG. 2.3 – Architecture proposée

d'événement, ainsi que le mécanisme de vol de tâches, dans le but de contrôler finement le placement des traitants.

Elle met également en avant des files de contrôle sur chaque unité d'exécution. Ces files sont en réalité des files d'événements réservées au support d'exécution, et facilitent les communications entre les unités d'exécution. Elles ne contiennent jamais d'événements applicatifs, et sont prioritaires sur les files associées aux traitants de l'application. Nous verrons par la suite comment les utiliser pour collecter des informations et replacer les traitants au sein du système.

Cette nouvelle architecture nous facilite grandement la distribution des traitants entre les unités d'exécution, autrement dit, le placement au sens où il est présenté dans la section 2.2.1. En effet, en réifiant les traitants, le placement se trouve réifié lui aussi, donc le contrôle que l'on peut avoir dessus est renforcé.

Un autre avantage fourni par cette architecture est qu'elle permet de mettre en place des techniques fines d'ordonnancement interne à chaque unité d'exécution. On peut par exemple modifier l'algorithme de round robin pour introduire des priorités et ainsi privilégier des traitants par rapport aux autres, au sein d'une même unité d'exécution. De même, il est également facile de mettre en place des techniques dites de *batching*, qui consistent à exécuter plusieurs fois le même traitant pour réutiliser des données en cache. De telles techniques d'ordonnancement feront l'objet de travaux ultérieurs.

Certes, cette architecture induit une indirection supplémentaire. Cependant, nos premiers tests montrent que son coût est négligeable, comme on le verra par la suite.

D'autres solutions ont été envisagées, notamment la possibilité de déplacer les traitants d'événements d'une unité d'exécution à l'autre directement, à la demande, sans réifier la notion de traitant. Cela s'apparente à la solution de vol de tâches déjà en place dans Libasync-mp. Cette solution doit être moins facile à manipuler car les liens entre traitants et unités d'exécution sont alors implicites. Au vu du faible coût de la solution choisie et des multiples avantages apportés par la réification des traitants, nous n'avons pas retenu ces autres solutions.

2.2.3 Mécanismes d'observation

Pour décider du meilleur placement à appliquer, il nous faut être capable d'observer l'application que nous exécutons. Nous allons détailler ici les différentes informations qui nous sont accessibles, les mécanismes qui nous permettent de les observer et leur utilité.

Informations accessibles

Différentes informations sont accessibles selon le niveau d'abstraction auquel on les demande. La plupart des architectures sont maintenant capables de fournir le nombre d'accès aux différents caches ainsi que le nombre de fautes d'accès engendrées sur ces caches. Elles permettent également d'obtenir les accès et les fautes d'accès au TLB[6]. Toutes ces informations sont tenues à jour sous la forme de compteurs matériels, dont les valeurs peuvent être remontées à l'utilisateur par le biais de bibliothèques. Une autre information pouvant s'avérer utile est le compteur associé au nombre de cycles processeur exécutés, qui permet par exemple de savoir combien de cycles sont nécessaires à l'exécution d'une fonction particulière.

Le système d'exploitation remonte aussi des informations au processus courant sur les ressources qui lui sont allouées. Parmi ces informations, on peut noter le nombre de fautes de pages engendrées par le processus, aisément récupérable par le biais d'un appel système. L'utilisation mémoire et processeur peuvent également être disponibles, mais sous une forme assez coûteuse à récupérer. En effet, ces informations ne sont accessibles que par le biais du système de fichiers contenu dans */proc*, dont les coûts ne sont pas maîtrisés. Il est ensuite nécessaire de faire une analyse syntaxique du fichier résultat. Sachant que ces informations concernent des métriques difficiles à interpréter, et sont en fait des estimations, nous avons décidé de nous concentrer sur les autres mesures ; celles-ci pouvant faire l'objet d'un travail ultérieur.

Le support d'exécution lui-même peut également fournir des informations. En effet, il donne accès au nombre de traitants présents dans l'application, ainsi qu'au nombre d'événements à traiter dans chaque file à un instant donné.

Collecte des informations

Chaque thread étant associé à une unité d'exécution, il n'a pas accès aux compteurs matériels de ses voisins. Dans l'optique d'une observation globale de l'application, il faut donc mettre en place un mécanisme de collecte des informations.

Une solution est d'utiliser un des threads existant, qui recevra sous la forme d'événements de contrôle les informations provenant de tous les threads. Il existe plusieurs façons de mettre un tel mécanisme en place. Le thread moniteur peut être fixe, c'est-à-dire toujours le même, ou bien élu, par un algorithme round robin par exemple.

Chaque thread peut choisir d'envoyer ses informations périodiquement, ou bien attendre un événement de contrôle pour les envoyer. La première solution d'envoi périodique pose le problème de la synchronisation des threads. En effet, il faut garantir que chaque thread ait fini d'envoyer ses informations au thread moniteur avant que celui-ci ne les collecte. La deuxième solution présente l'avantage que le thread moniteur envoie un événement de contrôle à chaque thread pour que chacun relève ses propres informations. La synchronisation peut alors être effectuée en comptant les réponses spécifiques à ce message, et en collectant les données une

[6]Translation Look-aside Buffer, tampon de traduction d'adresses virtuelles en adresses physiques.

fois que le thread moniteur a vu toutes les réponses. Cette solution a pour inconvénient qu'elle envoie plus d'événements de contrôle que la précédente. Cependant, la gestion de la synchronisation des threads sur les événements de contrôle n'étant pas simple, nous avons adopté la deuxième solution.

Au lieu d'utiliser un thread existant comme thread receveur, une solution possible est de créer un thread dédié à la collecte des informations. Cette solution a été testée mais n'a pas été retenue. En effet, étant donné que toutes les unités d'exécution ont déjà un thread propre, l'ajout d'un thread supplémentaire entraîne des changements de contexte. Ce comportement induit une certaine entropie dans le système, et l'impact sur les caches est difficilement prévisible.

Observation locale à un traitant

Chaque traitant étant réifié, notre architecture permet d'associer facilement un coût à chacun. Par coût on entend une mesure précise de chacune des informations citées en 2.2.3. Un traitant est le plus souvent destiné à être appelé plusieurs fois. Par conséquent, nous procédons à un échantillonnage sur chaque exécution de traitant.

2.2.4 Prise de décision

Comme nous l'avons vu, le support d'exécution doit être capable de décider dynamiquement du placement des traitants sur les différentes unités d'exécution.

Heuristiques de replacement

Nous allons maintenant expliquer en quoi les paramètres d'observation que nous avons vus peuvent influer sur la décision de replacement des traitants.

Nos principales métriques ici sont les compteurs matériels de performances et plus précisément ceux concernant les caches. En effet, si le nombre de fautes de caches augmente significativement (non proportionnellement aux accès à ces mêmes caches), ou dépasse un certain seuil, c'est qu'il y a une mauvaise utilisation des ressources matérielles. Il faut donc identifier quel(s) traitant(s) est(sont) à l'origine de ce comportement, et tenter de le replacer sur la même unité d'exécution qu'un traitant dont on a détecté qu'il utilisait les mêmes données. Cette détection d'affinités entre traitants peut faire l'objet d'une analyse statique, à la façon de Bathia *et al.*. L'observation locale à un traitant peut également être utile pour une telle détection, si on s'autorise à avoir une phase d'apprentissage pendant laquelle le système teste et observe différents placements. Il faut cependant garantir que le système est soumis à tous les types de charge possibles lors de cette phase d'apprentissage, pour être sûr d'observer tous les comportements possibles.

On s'intéresse moins ici aux fautes de pages et de TLB car elles reflètent à plus gros grain le comportement des caches. En effet, les caches sont gérés par lignes, ainsi une ligne de cache va contenir une ou plusieurs variables simples (selon l'associativité du cache et la taille des variables considérées), tandis que les pages contiennent généralement 4Kb de données. Une faute de page intervient lorsqu'un processus accède à une donnée qui n'est pas en mémoire centrale. Le système d'exploitation va dans ce cas chercher la page contenant la donnée voulue sur le disque dur. Il y a donc possibilité d'avoir un programme causant des fautes de caches mais n'entraînant aucune faute de page. Nous préférons donc nous concentrer sur l'observation des caches, sachant que la minimisation des fautes de caches devrait minimiser également les

fautes de pages. Cependant, nous ne négligeons pas cet aspect, étant donné le coût d'un accès disque à notre niveau.

Le nombre d'événements dans chaque file de traitant peut aussi être intéressante à observer. Il paraît nécessaire de les observer pour effectuer un rééquilibrage de charge intelligent. Cela peut servir notamment à déterminer si une unité d'exécution (un thread) est surchargé par rapport aux autres. Le découpage en traitants d'événements permet ensuite de savoir si un déplacement de traitant est envisageable sans affecter les affinités de cache du système, et donc les performances.

Le temps moyen d'exécution d'un traitant est également intéressant. Il permet d'attribuer à chaque traitant une pondération en fonction de son coût. Cela facilite la répartition des traitants sur les unités d'exécution, une fois étudiés les affinités de cache ainsi que les aspects de synchronisation entre traitants. La synchronisation entre traitants peut en effet constituer une contrainte de placement, étant donné que l'exclusion mutuelle est réalisée à la manière de Libasync-mp, mais au niveau du traitant plutôt que de l'événement : en plaçant deux traitants sur la même unité d'exécution, on assure qu'ils ne pourront s'exécuter simultanément. Cette garantie est due au fait qu'il n'y a pas de préemption dans le modèle de programmation événementiel.

Nous avons donc la possibilité de vérifier qu'un placement est efficace ou non. La question restante est la suivante : comment déterminer un placement ? On peut imaginer une solution qui énumère tous les placements possibles. Cependant, le nombre de placements envisageables explose de façon exponentielle avec le nombre d'unités d'exécution et de traitants dans l'application. Il existe alors deux approches :

- Demander au programmeur quels placements lui semblent adaptés à son application, et réduire l'analyse à cet ensemble. Cela peut très vite devenir contraignant quand le nombre de traitants et/ou de programmeurs augmente. De plus, Les placements proposés seront dépendant d'une architecture spécifique (en terme de nombre d'unités d'exécution), et devront être revus à chaque changement de ce facteur.
- Créer des heuristiques pour couper l'arbre des possibilités dans l'énumération des placements possibles. Cela rend ainsi le système de replacement complètement autonome, et de fait portable quelque soit le nombre d'unités d'exécution disponibles. Un exemple d'heuristique de coupe est de tenir compte de notre hypothèse selon laquelle toutes les unités d'exécution sont égales (en terme de puissance de calcul et d'accès à la mémoire centrale). L'idée est donc d'éliminer tous les placements symétriques, en tenant compte tout de même des hiérarchies : deux cœurs n'appartenant pas au même processeur ne sont pas égaux étant donné qu'ils ne partagent pas le même cache L2. Par exemple, si on considère un placement mettant en jeu consécutivement deux cœurs d'un même processeur, les possibilités d'équivalence pour ce placement sont alors de choisir deux cœurs appartenant au même processeur, et inversement si les cœurs considérés n'appartiennent pas au même processeur.

Redistribution des traitants

Dans le cadre de l'architecture proposée, nous avons étudié plusieurs heuristiques de redistribution des traitants (cf. Figure 2.4). Après avoir vu comment choisir un nouveau placement, nous allons maintenant considérer les différentes stratégies possibles pour appliquer la redistribution des traitants imposée par un nouveau placement. Pour effectuer une telle redistribution, il est judicieux de considérer l'architecture comme un système réparti. Par

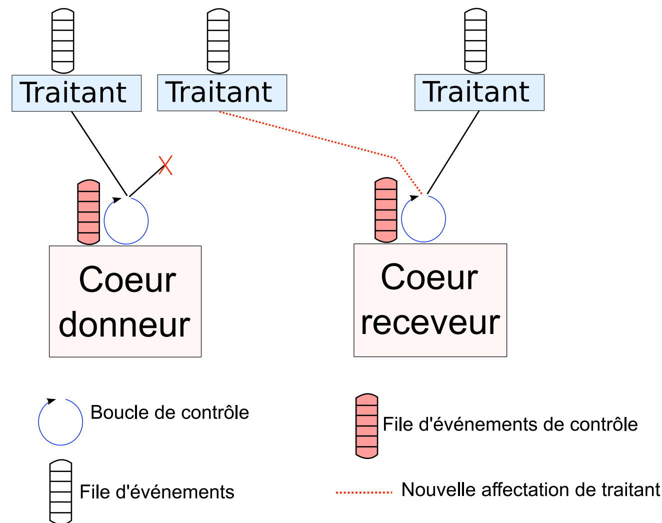

Fig. 2.4 – Redistribution de traitant d'événement

analogie, on appellera envoi de message la génération d'un événement (ici, de contrôle). Une fois la redistribution calculée par le thread moniteur, il existe plusieurs façons de l'appliquer à tout le système :

Stratégie de diffusion La nouvelle redistribution est envoyée à tous les threads. Le but étant que chaque thread mette à jour ses données propres, pour éviter les coûts de synchronisation à payer lorsque ce n'est pas le cas. Cela a pour inconvénient qu'un thread non concerné par la redistribution (qui ne donne ni ne reçoit aucun traitant) va tout de même recevoir un message de contrôle. Cela entraîne donc des traitements inutiles.

Un autre inconvénient notable est qu'aucune garantie n'est faite sur l'ordre des opérations. Ainsi, pendant une redistribution en diffusion, un traitant peut être placé sur aucune ou deux unités d'exécution. En effet, du point de vue d'un traitant replacé, aucune synchronisation n'est garantie entre le moment où le thread donneur se sépare du traitant et celui où le thread receveur l'enregistre. Le traitant peut donc être enregistré par le thread receveur avant d'avoir été donné par le thread donneur, ou inversement. Cela est problématique lorsqu'on veut garantir des propriétés d'exclusion mutuelle sur des traitants.

Stratégie réception de file Le thread moniteur envoie un message à chaque thread receveur. Sur réception de ce message, un thread va voler le traitant spécifié au thread auquel il appartient. Cela a pour avantage que seuls les threads concernés par la réception de traitants, les threads receveurs, vont faire l'objet d'un envoi de message. Cependant, chaque thread receveur doit, en plus d'enregistrer le traitant replacé, modifier l'état du thread donneur pour retirer ce traitant.

Outre les coûts de synchronisation associés à cette modification, on remarque qu'il est difficile de garantir que le traitant n'est pas en train de s'exécuter sur le thread donneur au moment où on le vole. Cela entraîne le même type de problème que précédemment lorsqu'il s'agit de garantir des propriétés d'exclusion mutuelle sur des traitants.

Stratégie donation de file Il s'agit de la stratégie duale de la réception de file. Le thread moniteur envoie un message à chaque thread donneur. Cette stratégie reprend le principal avantage de la réception de file, à savoir qu'un message n'est échangé qu'au besoin. Le fait que ce soit le thread donneur qui ait le contrôle lors de l'échange permet d'assurer que le traitant n'est pas en cours d'exécution à ce moment là. Cette stratégie peut donc garantir une exclusion mutuelle sur certains traitants.

En revanche, cela nécessite toujours qu'un thread modifie l'état d'un autre. En l'occurrence, le thread donneur doit enregistrer le traitant déplacé dans l'état du thread receveur.

Stratégie mixte Cette solution consiste à utiliser les deux principes de donation et réception. Le thread moniteur envoie un message au thread donneur, qui va se séparer du traitant concerné, et à son tour envoyer un message au thread receveur pour qu'il enregistre ce traitant. L'avantage de cette solution est que chaque thread met à jour son propre état. De plus, étant basé sur une stratégie donation, elle permet de garantir une exclusion mutuelle lors du replacement.

Cependant, cette stratégie nécessite deux envois de messages par échange de traitant, un pour la donation et un pour la réception. On ne peut donc pas garantir qu'un traitant sera en permanence associé à une unité d'exécution. En effet, le déplacement de chaque traitant se faisant de manière asynchrone, il peut y avoir une latence entre les deux événements, pendant laquelle un traitant sera détaché de son thread donneur, mais pas encore rattaché à son thread receveur. C'est cette stratégie que nous avons retenu pour la suite.

Optimisation Pour les stratégies autres que le diffusion, il est possible de réduire le nombre total de messages envoyés. Pour ce faire, l'idée consiste à grouper les traitants à échanger dans un unique message par thread. Ainsi, chaque thread reçoit une liste de traitants dont il va devoir se séparer ou bien récupérer selon la nature du message.

2.3 Synthèse

Dans ce chapitre, nous avons présenté notre contribution, qui consiste à améliorer la programmation événementielle en environnement multiprocesseur, en fournissant un mécanisme d'adaptation dynamique du support d'exécution. Nous avons décrit une architecture basée sur une modification de Libasync-mp. Aussi, nous avons vu quels paramètres étaient disponibles, comment s'en servir pour prendre une décision d'adaptation appropriée, et enfin comment appliquer cette décision au sein du support d'exécution.

Chapitre 3

Mise en œuvre

Ce chapitre présente les différentes phases du travail d'implémentation réalisé à partir de Libasync-mp. Ce travail se découpe notamment en trois phases distinctes. La première correspond à la mise en place d'une solution de placement statique des événements. La seconde consiste à apporter les mécanismes d'observation nécessaires à l'adaptation dynamique. Enfin, c'est cette adaptation qui constitue la troisième phase de réalisation. Pour des raisons de temps nous n'avons pas pu implémenter toutes les solutions proposées dans le chapitre précédent. Nous présentons ici l'état d'avancement de notre travail de mise en oeuvre.

3.1 Placements statiques

Le but des placements statiques est, dans un premier temps, de mesurer les effets des différentes stratégies de placement (vues en 2.2.1) sur des tests spécifiques (micro-benchmarks). Nous avons tout d'abord enlevé le mécanisme de vol de tâches, pour pouvoir assurer qu'une fois placé sur une unité d'exécution, un traitant y reste.

Nous avons ensuite retiré toute notion de couleur d'événement, au sens Libasync-mp, car elles sont absentes dans notre architecture. En effet, chaque événement doit être enregistré dans la file appartenant à son traitant (cf Figure 2.3).

Cela nous a donc permis d'implémenter notre architecture telle qu'elle est décrite au chapitre précédent, avec une file d'événements par traitant. Cela a notamment nécessité de réécrire entièrement les fonctions d'enregistrement d'événements, ainsi que la boucle de contrôle exécutée par chaque thread. A partir de ce stade, nous avons pu contrôler finement le placement de chaque traitant au lancement de l'application. C'est ce que nous appelons les placements statiques, car ils ne changent pas au cours de l'exécution.

3.1.1 Gestion de la synchronisation

Il est à rappeler que pour des raisons de performances un traitant ne devrait pas se bloquer dans son exécution, donc *a fortiori* aucun traitant ne devrait avoir à utiliser des primitives de synchronisation bloquantes. C'est pourquoi ces primitives doivent être fournies par le support d'exécution sous-jacent.

Dans un premier temps et par souci de simplicité, notre architecture garantit pour toute synchronisation une exclusion mutuelle dite *intra-traitant*. C'est-à-dire que le support d'exécution garantit à tout instant que deux événements associés à un même traitant ne s'exécu-

teront pas simultanément. Autrement dit, un traitant ne peut être placé que sur une seule unité d'exécution à chaque instant.

Cela a deux conséquences majeures, la première étant que deux traitants partageant des données doivent être placés explicitement par le programmeur sur la même unité d'exécution, car le support d'exécution n'a actuellement aucun moyen de protéger autrement ces accès concurrents. Cependant, notre hypothèse sur les applications de type pipeline réduit fortement les chances d'avoir ce genre de situation. En effet, les variables partagées entre traitants étant souvent contenues dans les messages qu'ils s'envoient, la forme même du pipeline suggère qu'un traitant ait fini son travail avant d'envoyer un événement sur le traitant suivant.

Une autre conséquence est que chaque traitant s'exécutera uniquement sur une et une seule unité d'exécution, qu'il soit en section critique ou non. Cela peut avoir une influence sur les performances observées, comme nous le verrons plus tard.

Nous comptons régler ces problèmes en intégrant la possibilité pour le programmeur de spécifier qu'un traitant peut être placé sur plusieurs unités d'exécution, ou doit être en exclusion mutuelle avec un autre traitant.

3.2 Observation des compteurs matériels

3.2.1 Présentation de PAPI

Comme nous l'avons dit plus tôt, les architectures modernes offrent des compteurs d'événements matériels, aussi appelés compteurs de performances. Cependant, ces compteurs sont propres à chaque architecture, et diffèrent selon les versions et les déclinaisons. Pour obtenir les valeurs de ces compteurs, nous avons utilisé la bibliothèque PAPI[1][9, 6], qui fournit entre autres une interface de nommage standard des compteurs matériels. De ce fait, elle s'impose comme une interface portable sur un certain nombre d'architectures existantes.

PAPI peut émuler des compteurs non présents sur l'architecture à partir d'autres compteurs disponibles, lorsque cela est possible. Prenons l'exemple d'une architecture qui ne permet pas d'observer les succès d'accès au cache L2 (L2 cache hits), mais qui offre des compteurs pour observer les accès totaux au cache L2 ainsi que les fautes de cache L2. PAPI propose dans ce cas un compteur de succès de cache émulé à partir des deux autres.

L'utilisation de cette bibliothèque nécessite un noyau modifié du système d'exploitation. En effet, étant donné qu'elle offre la possibilité de s'attacher à un fil d'exécution particulier, il lui faut un moyen de savoir quand ont lieu les changements de contexte effectués par le système d'exploitation.

Cette modification permet également de multiplexer les compteurs de performances, c'est-à-dire d'effectuer un échantillonnage des compteurs, dans le but de faire apparaître plus de compteurs qu'il n'y en a vraiment sur l'architecture. En effet, une limitation souvent rencontrée sur les architectures est qu'elles ont un nombre limité de compteurs génériques. Le multiplexage résout ce problème, dans le sens où l'échantillonnage des compteurs génériques matériels va permettre d'émuler un plus grand nombre de compteurs.

Cet échantillonnage fait cependant perdre en précision lors des mesures. Si les observations sont suffisamment espacées, alors cette perte est négligeable, l'échantillonnage étant plus précis, mais à l'inverse pour des observations très courtes, il vaut mieux s'en passer.

[1]the Performance Application Programming Interface

3.3 Adaptation

La solution d'adaptation que nous avons mis en oeuvre est la suivante. Le programmeur spécifie tout d'abord un ensemble de placements qu'il juge efficaces connaissant les affinités entre les traitants de son application. Lorsque l'application est lancée, le support d'exécution itère parmi ces placements, en observant les accès et les fautes de cache L2 pour chacun d'eux. Cela met en avant une hypothèse forte selon laquelle la charge est constante durant la phase d'itération.

A la fin de cette itération, les traitants de l'application sont placés selon une heuristique simple. Si l'application n'a pas eu une sur-activité en cache, alors le placement choisi est celui qui maximise la parallélisation des traitements. De cette manière, l'adaptation favorise la parallélisation des tâches lorsque ce n'est pas coûteux.

Sinon, si une sur-activité a été observée sur les caches, alors le placement est choisi de manière à minimiser les fautes de cache L2. Ainsi, la perte de performances est minimisée.

La sur-activité est mesurée en fonction des accès aux cache, et comparée à un seuil, fixé à partir de nos observations d'expérimentations (cf Chapitre 4). Pour se protéger des erreurs de mesure et ainsi éviter des redistribution de traitants trop fréquentes, nous avons mis en place un mécanisme d'hystérésis (cf Figure 3.1). Cela revient à dire qu'une marge d'erreur est prise en compte lors de la mesure de la sur-activité des caches, ce qui augmente la stabilité du système.

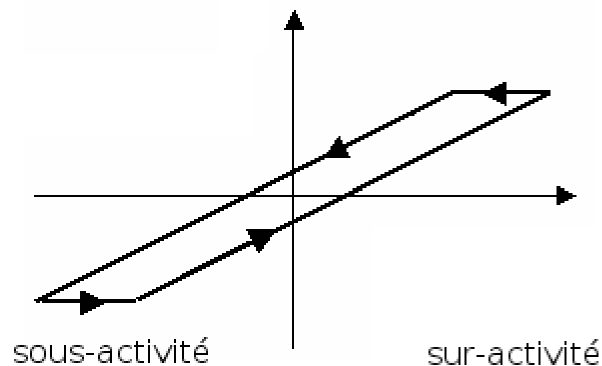

Fig. 3.1 – Exemple d'hystérésis. Les seuils sont différents selon le sens dans lequel on change d'état.

Nous avons prévu d'élargir pour notre comparaison la notion de seuil vers une notion de tendance. Cette tendance pouvant être implémentée comme un couple (valeur,gradient), ou bien, plus simplement correspondre à une limite sur le nombre de fois que le seuil est atteint.

3.4 Synthèse

Dans ce chapitre, nous avons présenté l'état d'avancement du travail d'implémentation qui a été effectué. L'architecture proposée au chapitre précédent a été implémentée. Nous avons utilisé la bibliothèque PAPI pour observer les compteurs de performances et mettre en place notre solution d'observation. Une première solution d'adaptation dynamique a également été

mise en oeuvre. Celle-ci est basée sur un ensemble de placements choisis par le programmeur, et utilise un mécanisme de seuil avec hystérésis pour la redistribution des traitants.

Chapitre 4

Résultats expérimentaux

4.1 Conditions d'expérimentation

Nos tests ont été effectués sur des machines ayant la configuration suivante :

- Intel Xeon 5100
- Bi-processeur double cœur cadencé a 1.8GHz
- 32 -Ko de cache L1 d'instructions et 32 Ko de cache L1 de données sur chacun des quatre cœurs
- 4 Mo de cache L2 sur chacun des processeurs
- 2Go de mémoire vive
- Noyau Linux 2.6.22, modifié avec le patch *perfmon2*

Les micro-benchmarks considérés sont des applications en pipeline, où chaque traitant effectue un traitement similaire (les traitants restant tout de même différenciés). Chaque pipeline étudié contient 16 traitants. Chaque test est initialisé avec un certain nombre de messages dans la file du premier traitant. Le remplissage de sa file n'est pas pris en compte dans le temps d'exécution du pipeline, car le traitement des messages ne commence que lorsque celle-ci est pleine. On répète chaque test cent fois afin de faire une analyse en moyenne. L'écart à cette moyenne est la plupart du temps tellement fin (il atteint au maximum 2%) qu'il est difficile de distinguer les marges d'erreurs sur nos courbes[1]. Notre architecture permettant un *batch*, nous l'avons réglé à 1. C'est-à-dire qu'après avoir traité un événement, la boucle de contrôle va ordonnancer un événement d'un autre traitant s'il y en a. Cela va favoriser les caches de données, au détriment des caches d'instructions.

Nous présentons ici deux types de d'applications de tests. Le premier est un micro-benchmark, qui nous permet de tester le comportement du système avec une application excessivement intensive au niveau CPU. Il consiste à effectuer des racines carrées en boucle. Chaque traitant du pipeline fait donc 10000 racines carrées par événement reçu. Le test est initialisé avec 4000 événements dans la première file.

La deuxième application de test est plus réaliste. Il s'agit d'appliquer successivement des filtres sur des images. Par filtre on entend une opération point à point. Plus précisément, chaque traitant effectue un parcours entier de l'image, et exécute un opérateur binaire (multiplication par un scalaire, addition ou modulo) sur chaque pixel de celle-ci. L'objectif de ce benchmark est de mettre en avant un comportement intensif au niveau mémoire, et d'exposer

[1]Ces marges d'erreurs sont toutefois absentes des courbes qui ont plus d'un histogramme par placement (4.3 4.4 4.6 et 4.8), car cela surchargerait inutilement ces graphiques

ainsi des effets de cache. Le test est initialisé avec 4000 événements dans la première file, chaque événement correspondant à une image de 700Ko.

4.2 Tests de placements statiques

Les figures 4.1 et 4.2 montrent les performances obtenues lors des premiers tests avec placements statiques. Les courbes bleues représentent les performances de Libasync-mp avec vol de tâches, et les courbes rouges les performances avec les placements statiques décrits en 2.2.1.

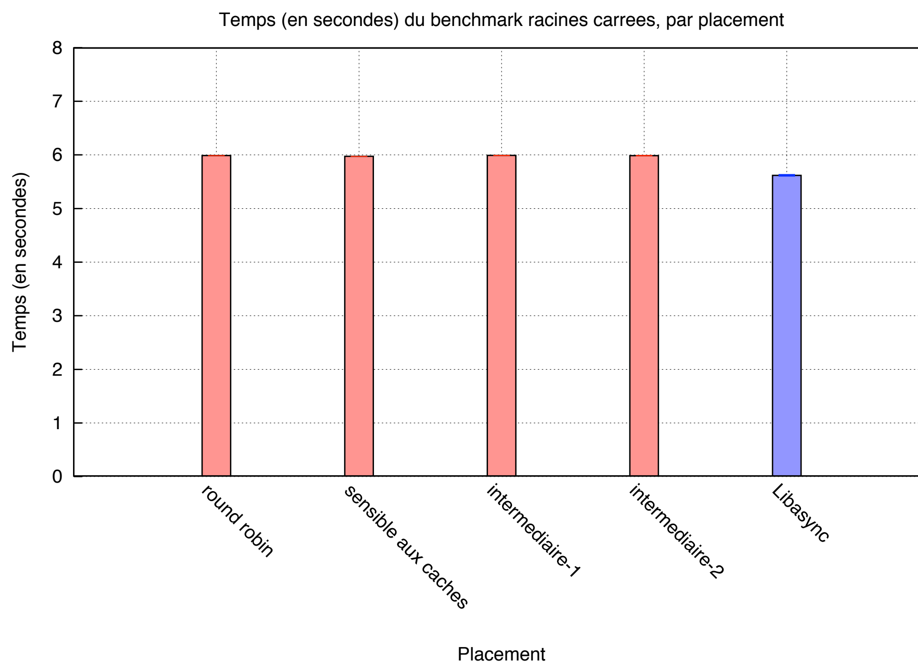

FIG. 4.1 – Résultats du benchmark CPU-intensif.

On remarque que Libasync-mp est meilleur de 6% par rapport à nos placements statiques sur les benchmarks cpu-intensifs (cf Figure 4.1). Nous pensons que cela est dû au fait que notre solution actuelle impose une exclusion mutuelle sur chaque traitant, donc deux événements ayant le même traitant s'exécutent sur la même unité d'exécution. A contrario, Libasync-mp n'a pas cette contrainte, et son vol de tâches lui permet d'assurer une bonne répartition des événements sur les différentes unités d'exécution. Comme le degré de parallélisme est le paramètre le plus important sur des tests cpu-intensifs, cela explique pourquoi nos placements statiques ne sont pas meilleurs que Libasync-mp. Ces différences devraient disparaître lorsque nous aurons implémenté une gestion fine de la synchronisation.

En revanche, sur les tests de filtres d'images (cf Figure 4.2), les gains des placements statiques par rapport à Libasync-mp varient entre 15% et 42% selon le placement. Nous attribuons ces différences de performances à des phénomènes de localité de cache. En effet, le placement sensible aux caches engendre les meilleures performances (42% de gain par rapport à Libasync-mp). Le fait que tous les placements testés aient de meilleures performances que

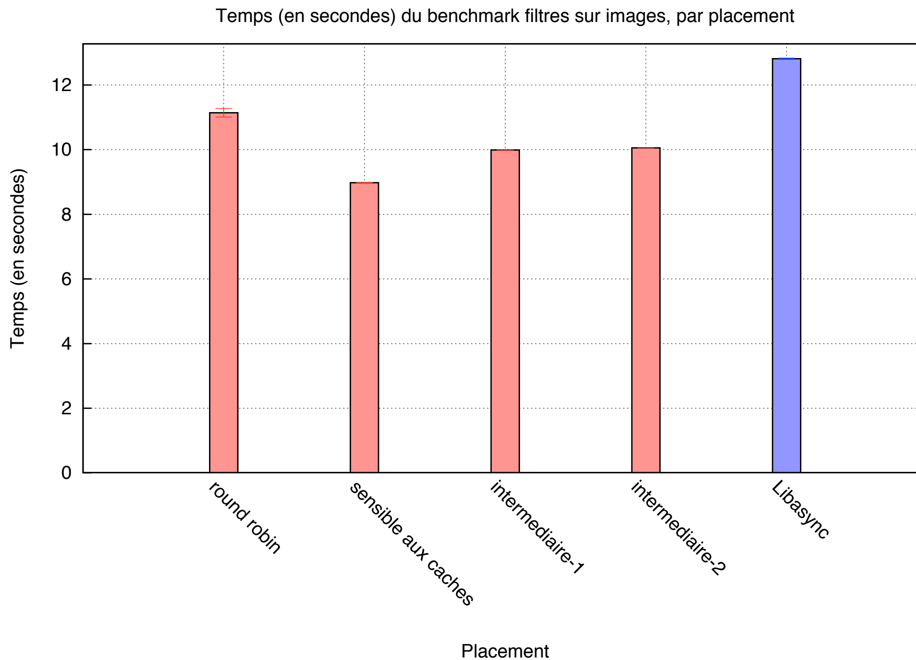

FIG. 4.2 – Résultats du benchmark filtres d'images.

Libasync-mp est facilement explicable. Les effets de cache ont un rôle très important dans l'application, or le vol de tâches réduit la localité de cache de par sa nature. Le fait d'avoir un placement statique a tendance à augmenter la localité des traitements sur chaque unité d'exécution.

Observation des compteurs de performances

L'observation des caches nous permet de corréler les différences de performances observées avec le nombre de fautes de caches. Cela confirme donc notre hypothèse selon laquelle les caches jouent un rôle important dans les performances, et donc qu'un placement sensible aux caches permet de gagner de façon non négligeable.

La figure 4.3 montre le taux de fautes de cache L2 engendré par le benchmark filtres d'images (nombre de fautes de cache divisé par nombre total d'accès à ce cache). Ainsi, on peut voir que le vol de tâches induit un grand nombre de fautes de cache par rapport aux placements statiques, notamment celui sensible aux caches. Cela est aussi vrai même sur les autres placements, mais c'est moins apparent étant donné que chaque cœur n'induit pas le même nombre de fautes de caches. On rappelle que les cœurs 0 et 1 partagent leur cache L2 et qu'il en va de même pour les cœurs 2 et 3, car ces paires de cœurs sont situés sur les mêmes processeurs.

Le taux de fautes de cache lors des tests CPU-intensifs montre que les différences entre les placements et la Libasync-mp sont beaucoup moins marquées, sinon inexistantes (cf Figure 4.4). Cela s'explique par le fait que ce benchmark n'utilise que très peu de mémoire, donc les fautes de caches rencontrées sont dues uniquement aux premiers chargements des variables dans les caches, appelées *cold misses* par Koka *et al.*[11]. Ces fautes-ci sont inévitables, celles

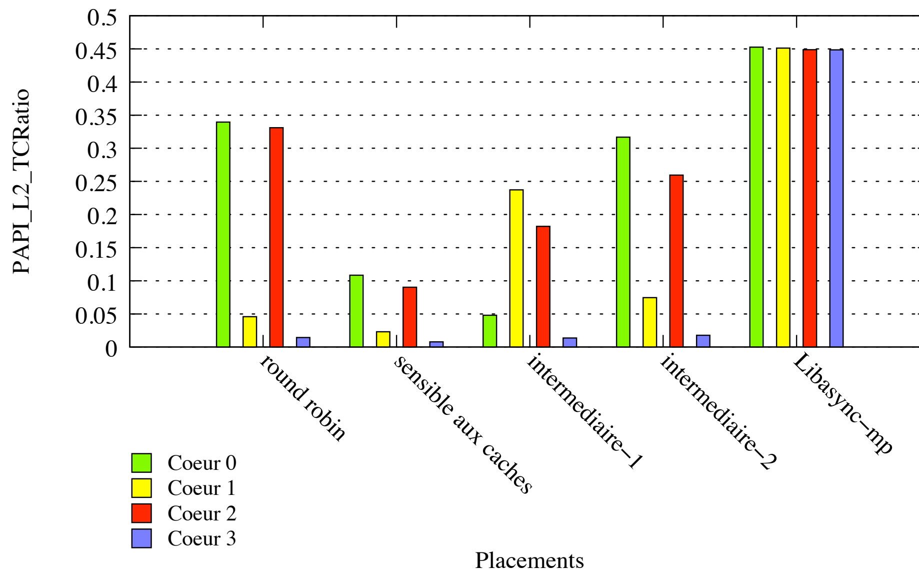

FIG. 4.3 – Taux de fautes de cache L2 du benchmark filtres d'images.

que l'on peut la plupart du temps éviter sont généralement dues à des problèmes d'ordonnancement de code réutilisant des variables déjà chargées. Or il n'y a aucune réutilisation de données dans ce benchmark, ce qui explique les faibles différences observées ici.

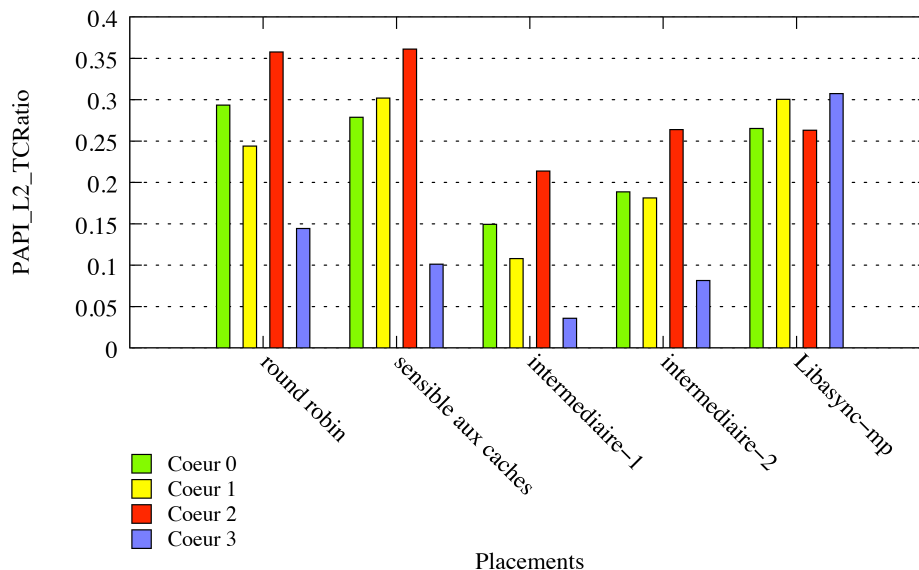

FIG. 4.4 – Taux de fautes de cache L2 du benchmark CPU-intensif.

4.3 Tests avec adaptation dynamique

Les figures 4.5 et 4.7 montrent les performances obtenues lors des tests avec adaptation dynamique. Notre système d'adaptation utilise le même système d'observation que décrit précédemment, avec un thread interne élu par round robin, et une relève des compteurs périodique et globale à l'application toutes les 2 secondes. L'adaptation se fait telle que décrite au chapitre 3.3, c'est-à-dire en itérant sur les quatre placements choisis, puis en choisissant le meilleur.

Nous avons vu que sur nos tests CPU-intensifs que les placements statiques choisis étaient tous équivalents, donc l'adaptation dynamique de placement n'apporte aucun gain ici, étant donné que la charge ne change pas de nature. Le coût du replacement des traitants doit donc apparaître lors de ces benchmarks CPU-intensifs avec adaptation dynamique. La figure 4.6 montre que coût est négligeable (inférieur à 1%). Cependant, ces tests nous avaient également montré un coût d'observation négligeable, donc cela est certainement à relativiser, et cette conclusion ne doit s'appliquer qu'aux tests CPU-intensifs.

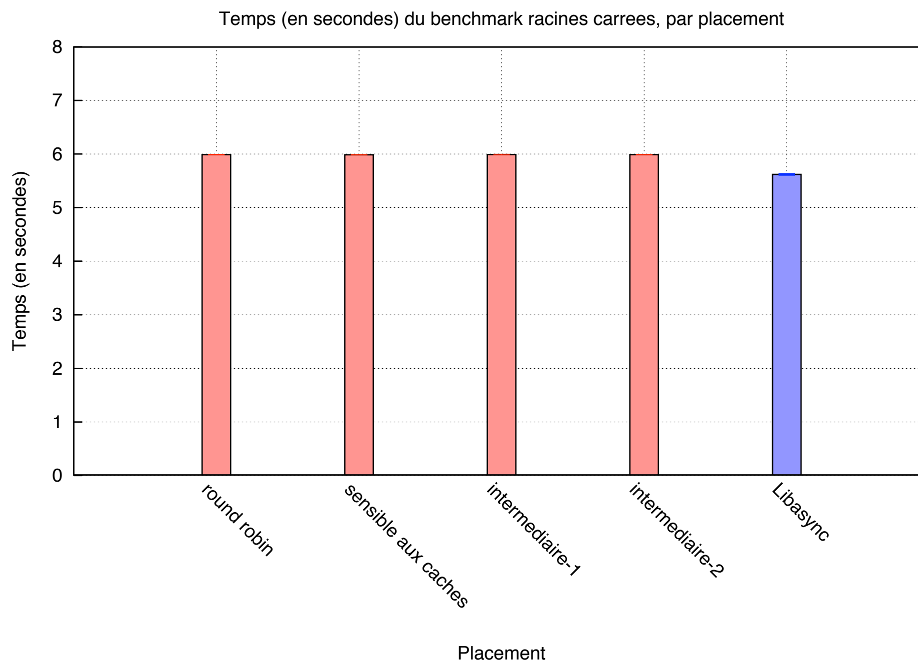

Fig. 4.5 – Résultats du benchmark CPU-intensif avec adaptation dynamique.

Dans les autres cas, les placements statiques n'étant pas tous équivalents, l'adaptation dynamique apporte un gain de performances, et son coût est donc difficilement appréhendable. En revanche, il est intéressant de voir si l'on continue à gagner en performances par rapport au vol de tâches avec une telle adaptation. Dans le cas du benchmark de filtres d'images (cf Figure 4.7), on voit que quelque soit le placement de départ, les performances des différents placements ont été lissées.

Cela est dû au temps que prend l'itération sur les différents placements choisis, chacun étant testé pendant une période d'observation, soit 8 secondes en tout.

Ce qui a pour effet qu'il faut attendre environ 8 secondes d'exécution après le début de

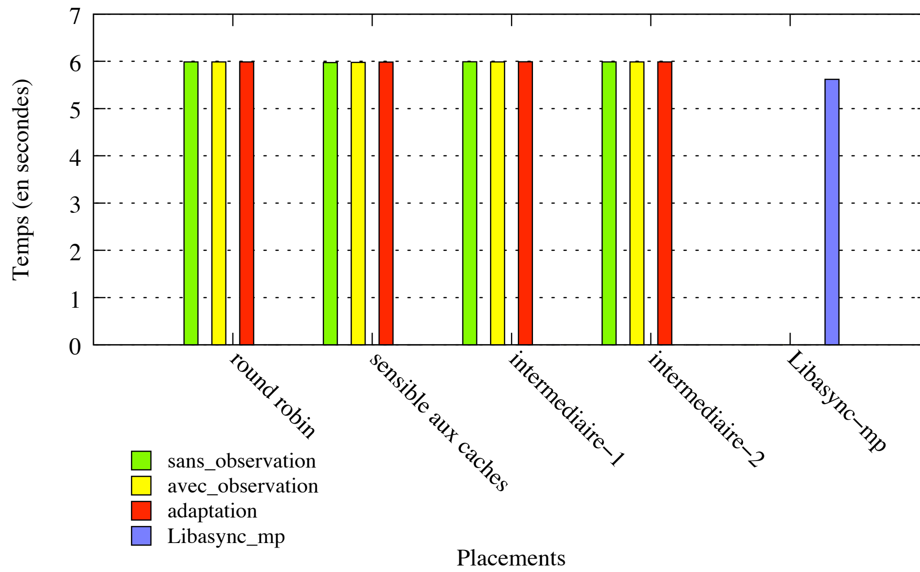

Fig. 4.6 – Synthèse des résultats du benchmark CPU-intensif.

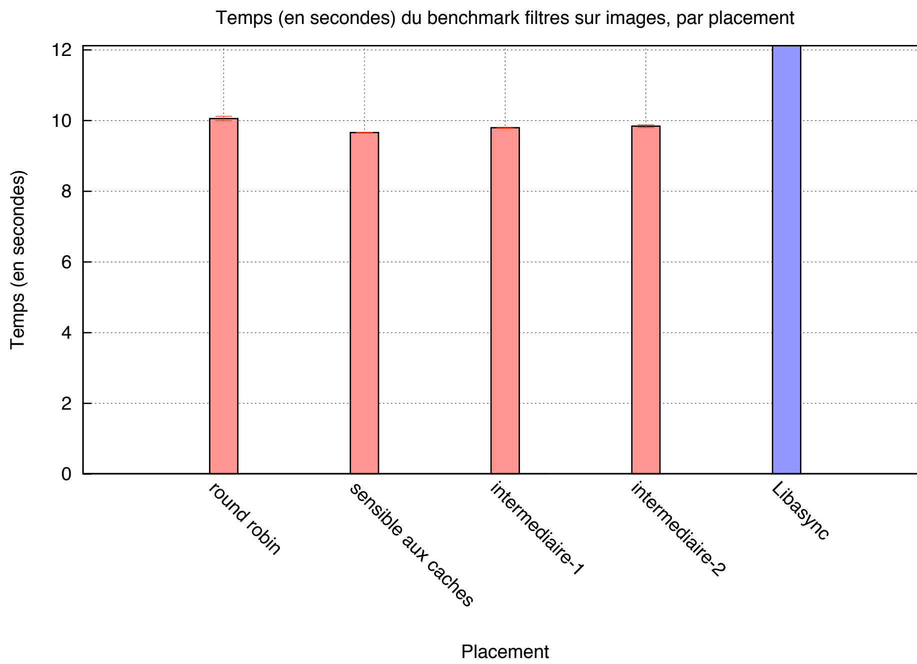

Fig. 4.7 – Résultats du benchmark filtres d'images avec adaptation dynamique.

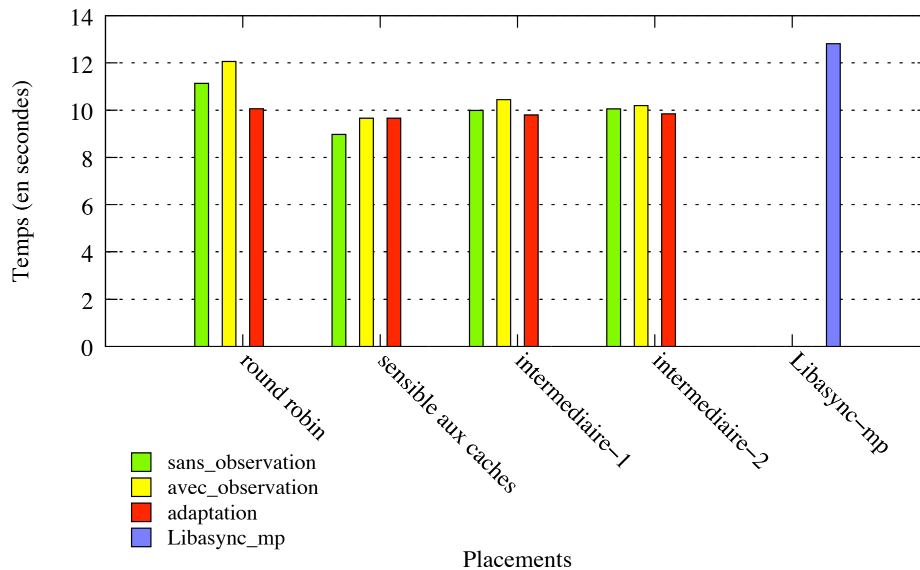

FIG. 4.8 – Synthèse des résultats du benchmark filtres d'images.

l'itération pour que le meilleur placement soit appliqué, or notre test dure environ 10 secondes.

Les différences observées entre les différents placements statiques sont dues au fait que les deux premières secondes d'exécution sont spécifiques au placement de départ du test, l'itération ayant lieu après. Cela explique également que les placements choisis apparaissent encore sur les tests d'adaptation.

Notre test est donc trop court pour mesurer le véritable impact de notre solution d'adaptation. On remarque tout de même un gain situé entre 27% et 32% (selon le placement de départ) par rapport à Libasync-mp et son vol de tâches (cf Figure 4.7). Nous pensons que le fait d'allonger la longueur de ces tests devrait creuser l'écart des performances entre notre solution et celle du vol de tâches.

Cependant, nous sommes ici limités par la capacité mémoire de notre machine de test. En effet, pour rallonger l'exécution de ce test, il faudrait initialiser le pipeline avec plus de messages en entrée. Or si l'on tente d'en rajouter plus, le nombre de messages initiaux dépasse la capacité mémoire de la machine, et le test se retrouve soumis à des entrées-sorties sur disque dur non contrôlées. Ces E/S modifient le comportement de l'application et ajoutent beaucoup d'entropie dans les performances, qui deviennent difficiles à analyser.

Pour régler ce problème, nous sommes en train de réaliser des benchmarks tirant leur charge de travail depuis le réseau. Nos premiers benchmarks dans cette optique consistent en un serveur web, et un serveur de filtrage d'images. Il est trop tôt pour avoir des résultats concrets. Cependant, il semblerait que le placement des traitants manipulant le réseau (ouverture, lecture, écriture et fermeture de sockets) ait également une importance cruciale sur les performances des applications réparties.

4.4 Coût de l'observation

Pour évaluer le coût du mécanisme d'observation mis en place, nous comparons les tests avec et sans observation des compteurs de performances, sur des placements statiques. Dans le but de mesurer uniquement le coût de l'observation, nous avons désactivé l'adaptation dynamique du système.

Les figures 4.9 et 4.10 montrent les performances obtenues lors des tests avec placements statiques et observation des compteurs de performances. Notre système d'observation utilise un thread interne élu par un algorithme de round robin. Le thread moniteur demande la relève des compteurs de performances toutes les 2 secondes. Cela correspond au mécanisme d'observation globale présenté au chapitre 2.2.3.

On peut voir sur la figure 4.6 que les performances avec observation sont identiques à celles sans observation sur les tests CPU-intensifs. Nous en déduisons que l'observation n'est pas intensive du point de vue CPU, et a un coût négligeable (moins d'1%) dans le cas d'applications qui le sont.

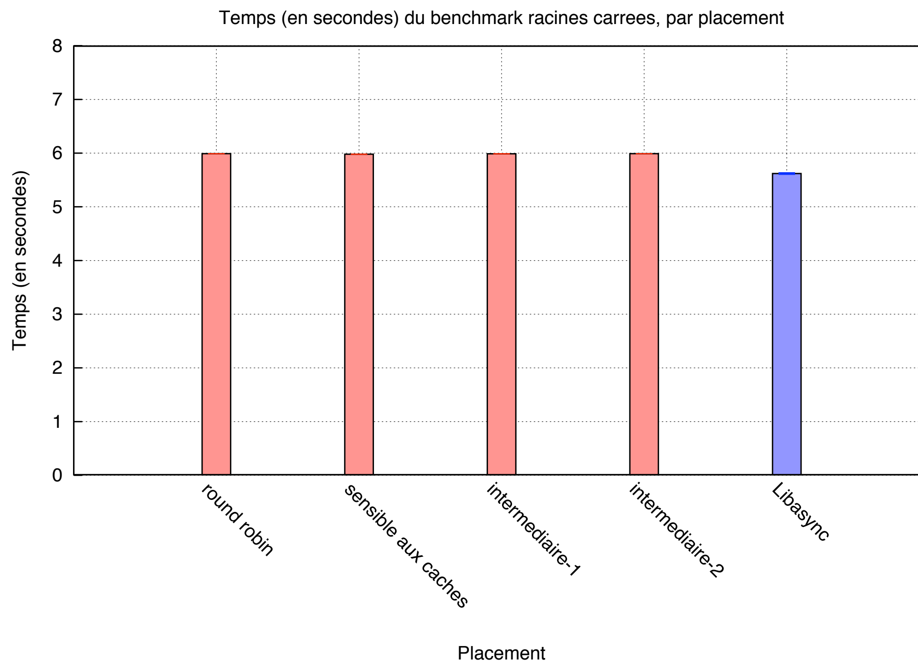

Fig. 4.9 – Résultats du benchmark CPU-intensif avec observation des compteurs de performances.

Le coût de l'observation apparaît plus important sur la figure 4.10, avec le benchmark de filtres d'images. La différence de performances avec et sans observation atteint 7%, voire 8% dans le pire des cas (cf Figure 4.8). Nous n'avons pas constaté de coût plus élevé parmi les autres benchmarks que nous avons étudiés. Ce coût peut être imputé à la perturbation des caches dûe à l'observation. Cela expliquerait pourquoi il n'est pas visible sur des tests purement CPU-intensifs, sur lesquels les caches n'ont pas ou peu d'effet.

Cependant, l'observation ne change pas le comportement global de l'application, et les placements statiques restent avantageux par rapport au vol de tâches sur ce type de test. Le

placement le plus avantageux ici reste celui qui est sensible aux caches, qui fait tout de même gagner 32% par rapport au vol de tâches.

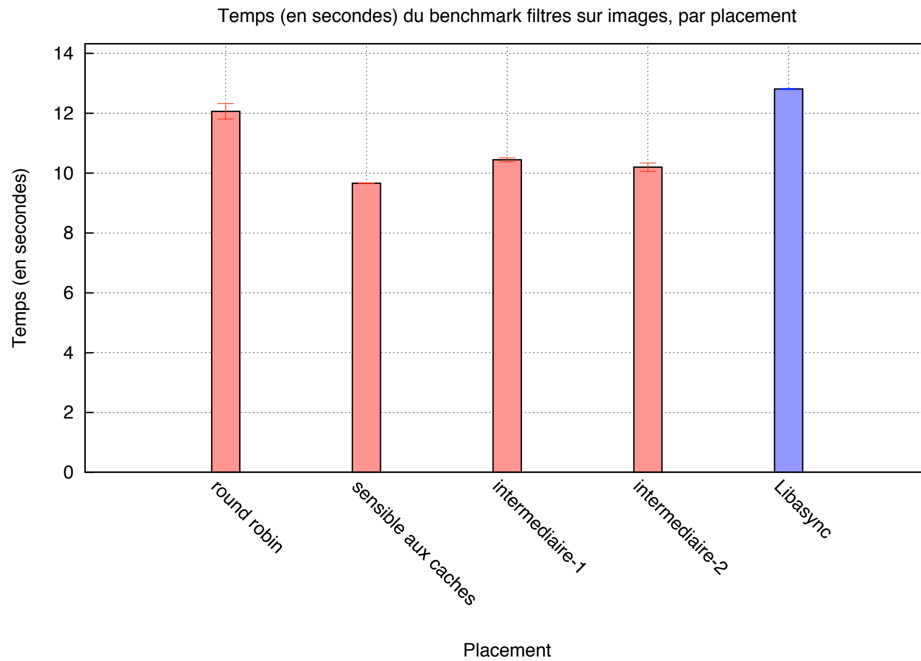

Fig. 4.10 – Résultats du benchmark filtres d'images avec observation des compteurs de performances.

4.5 Synthèse

Dans ce chapitre, nous avons présenté des résultats d'expériences réalisées à l'aide de l'architecture que nous proposons. Ces expériences ont tout d'abord permis de confirmer que les placements statiques étaient avantageux par rapport au vol de tâches, dans le cas d'applications utilisant la mémoire (et pouvant exhiber une certaine localité de cache). Ces expériences ont également permis de valider le mécanise de modification dynamique du support d'exécution. Elles ont notamment mis en avant le fait que le surcoût lié à la prise de décision parait très faible, et que notre adaptation dynamique permet d'obtenir un gain de performances d'environ 30% par rapport à Libasync-mp dans cetains cas.

Chapitre 5

Conclusion

5.1 Bilan

Nous avons détaillé les enjeux apportés par les architectures modernes, ainsi que différentes solutions proposant de tirer parti de ceux-ci. On retiendra notamment les travaux de Bhatia *et al.*[5] portés sur les optimisations sensibles aux caches, et de Zeldovich *et al.*[20, 8], sur le portage du modèle événementiel en contexte multiprocesseur.

Nous avons ensuite montré qu'il est intéressant de considérer plusieurs placements de traitants selon les cas. Pour cela, nous avons testé différentes applications événementielles, en observant leurs comportements en cache selon le placement choisi. En comparant les résultats obtenus par ces tests, nous avons montré qu'il existe un gain réel à placer les traitants des applications utilisant intensivement la mémoire de manière à tirer parti des caches, étant donné que l'on peut gagner de l'ordre de 30% par rapport au vol de tâches de Libasync-mp (notre référence ici). Nous avons également vu que les applications purement intensives au niveau CPU ont un comportement assez différent. Ces expériences montrent qu'il est utile de pouvoir modifier dynamiquement le placement des traitants, notamment si l'on s'intéresse à des serveurs de données, dont la charge, venant du réseau, peut varier en nature et en intensité de manière non prévisible.

Nous avons enfin exposé une solution d'adaptation dynamique basée sur les observations des caches matériels. Les résultats ont montré que cette solution est viable et que ses gains sont réels.

5.2 Perspectives

Types d'applications supportés

Dans ces travaux nous nous sommes attachés à montrer qu'il existe un gain réel à un changement dynamique de placement des traitants. Les premiers résultats sont encourageants, cependant ils restent basés sur des hypothèses fortes, qu'il convient de lever. Ces résultats doivent donc être confirmés sur des applications plus réalistes, notamment sans forcément avoir un pipeline de traitants d'événements, mais des structures d'applications plus complexes, contenant branchements et boucles.

Dans le but d'avoir des résultats pertinents sur notre solution d'adaptation, nous devons rallonger la longueur des tests effectués. Comme nous l'avons vus, cela est impossible sans

prendre en compte des entrées-sorties, soit contrôlées (injecter de la charge par le réseau sur un serveur), soit non contrôlées (E/S disque pour utiliser le swap du système d'exploitation). Pour cela, il nous faut comprendre finement les mécanismes utilisés et les comportements des E/S, notamment les différences de comportements relatives au placement des traitants effectuant ces E/S.

Toujours dans le but de traiter des situations plus réalistes, nous devrons également assurer que notre adaptation dynamique se comporte de façon convenable face à des variations de charge. En effet, si une telle variation intervient après une décision de replacement, il faut recalculer une nouvelle décision. Cependant, si cette variation prend effet lors de l'itération sur les placements possibles, nous devrons tout de même garantir que la décision prise restera efficace. Dans ce cas, relancer une nouvelle itération face à une variation de charge n'est pas une solution, étant donné que ces variations peuvent avoir une fréquence supérieure au temps d'itération (on se retrouverait alors à relancer perpétuellement des itérations les unes après les autres).

Synchronisation

Nous avons également fait une hypothèse forte sur la synchronisation des traitants, et ce à deux niveaux. Le premier étant que le programmeur doit pouvoir spécifier que deux traitants doivent être en exclusion mutuelle. Cela ne change pas en soi l'architecture proposée, mais va limiter les possibilités de replacement dynamique, en ajoutant des contraintes du type : tel ensemble de traitant doit être placé au même endroit pour respecter une exclusion mutuelle.

Le deuxième niveau est du point de vue de la synchronisation intra-traitant que notre solution adopte actuellement en permanence. Il est fort probable que nous puissions gagner en performance en n'utilisant une telle synchronisation que lorsqu'elle est nécessaire, et en autorisant ainsi la plupart des traitants à s'exécuter sur plusieurs processeurs. Cela devrait notamment nous permettre d'atteindre les performances du vol de tâches sur des applications intensives au niveau CPU. Cependant cela risque d'entraîner des changements dans nos heuristiques de replacement, qu'il faudra augmenter pour prendre en compte de telles situations.

Stratégies de placements

Il est également intéressant d'étudier la solution énumérant les placements pertinents, plutôt que de laisser le programmeur fournir un ensemble de placements qu'il juge pertinents. En effet, plus l'application grandit, plus le nombre de traitants est important, et donc les possibilités de placement sont nombreuses et fastidieuses à étudier. Pour mettre en place une telle solution, il faut déterminer des heuristiques réduisant les possibilités sans éliminer les placements efficaces pour l'application concernée. On pourra certainement se servir de contraintes telles que l'exclusion mutuelle entre certains ensembles de traitants pour cela, cependant il faudra d'autres types de contraintes pour réduire suffisamment l'arbre des possibilités, réduire le temps de la phase d'itération sur les placements retenus, et ainsi rendre efficace notre adaptation de façon générique.

Ordonnancement à grain fin

Un autre objectif est, une fois étudiées les différentes solutions menant à un placement des traitants efficace, de réaliser une étude des politiques d'ordonnancement internes à chaque unité d'exécution. En effet, chaque unité d'exécution peut avoir intérêt à ordonnancer en

priorité un sous-ensemble des traitants qui lui sont affectés. Par exemple pour tenir compte plus finement des effets de cache, ou bien mettre en place des politiques de gestion de qualité de service, etc... Il est à noter que l'adaptation dynamique du placement pourrait au besoin communiquer des informations sur la structure de l'application à chaque unité d'exécution pour les aider dans cet ordonnancement.

Validation

Enfin, un dernier objectif est, de porter nos travaux sur plusieurs plateformes. Nous nous comparons pour l'instant à Libasync-mp, mais ces travaux sont portables sur d'autres environnements d'exécution, et nous avons en vue de les tester sur SMP Click ainsi que Cecilia.

Bibliographie

[1] Cecilia, http ://fractal.objectweb.org/c.html.

[2] Fractal, http ://fractal.objectweb.org/.

[3] Native posix thread library, http ://people.redhat.com/drepper/nptl-design.pdf.

[4] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 289–302, Monterey, CA, June 2002. USENIX Association.

[5] Sapan Bhatia, Charles Consel, and Julia Lawall. Memory-manager/scheduler co-design : Optimizing event-driven programs to improve cache behavior. In *Proceedings of the 2006 ACM International Symposium on Memory Management (ISMM'2006)*, pages 104 – 114, Ottawa, Canada, June 2006.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3) :189–204, Fall 2000.

[7] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 333–346, Boston, Massachusetts, June 2001. USENIX Association.

[8] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *EW10 : Proceedings of the 10th ACM SIGOPS European workshop : Beyond the PC*, pages 186–189, Saint-Emilion France, September 2002. ACM Press.

[9] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters, 2003.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture. A quantitative approach. Fourth edition.* Morgan Kaufmann, 2006.

[11] P Koka and M H Lipasti. Opportunities for cache friendly process scheduling. In *In Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[12] James R. Larus and Michael Parkes. Using cohort scheduling to enhance server performance (extended abstract). In *LCTES '01 : Proceedings of the ACM SIGPLAN workshop on Languages compilers and tools for embedded systems*, pages 182–187, New York NY USA, 2001. ACM Press.

[13] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5) :33–42, May 2006.

[14] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *SOSP '99 : Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 217–231, Charleston South Carolina, United States, 1999. ACM Press.

[15] John K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, http ://-home.pacbell.net/ouster/threads.pdf, January 1996.

[16] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash : An efficient and portable web server. In *USENIX Annual Technical Conference General Track*, pages 199–212, Monterey, California, USA, June 1999.

[17] Andrew Tanenbaum. *Modern Operating Systems, third edition*. Pearson International Edition, 2008.

[18] Xi Wang, Zhenyu Guo, Xuezheng Liu, Zhilei Xu, Haoxiang Lin, Xiaoge Wang, and Zheng Zhang. Hang analysis : fighting responsiveness bugs. In *Eurosys '08 : Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 177–190, New York, NY, USA, 2008. ACM.

[19] Matt Welsh, David Culler, and Eric Brewer. SEDA : An architecture for well-conditioned scalable internet services. In *SOSP '01 : Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, Banff Alberta Canada, 2001. ACM Press.

[20] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and M. Frans Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference General Track*, pages 239–252, San Antonio, Texas, June 2003. USENIX Association.

Résumé

La programmation événementielle est une technique connue pour mettre en œuvre des applications résistantes à la charge. Dans sa version classique, ce modèle de programmation ne permet pas de tirer parti efficacement des architectures multicœur et/ou multiprocesseur. Or ce type d'architectures, autrefois réservé au calcul à haute performance, a tendance à se répandre de plus en plus et à percer dans d'autres usages, notamment celui des serveurs de données. Dans ce contexte, les travaux présentés dans ce rapport proposent un support d'exécution événementiel et flexible en environnement multiprocesseur. Plus précisément, l'environnement d'exécution que nous développons permet d'associer chaque traitant d'évènement d'une application événementielle à une unité d'exécution de l'architecture (cœur ou processeur). L'objectif de ces travaux de recherche est de proposer un ensemble d'heuristiques permettant le choix d'un placement efficace des traitants d'évènements sur les unités d'exécution disponibles.

Mots Clés : programmation événementielle, adaptation dynamique, serveurs de données, parallélisme, caches, multiprocesseur, multicœur , processus légers.