HAL
open science

Étude et amélioration de la performance des serveurs de données pour les architectures multi-cœurs

Fabien Gaud

To cite this version:

Fabien Gaud. Étude et amélioration de la performance des serveurs de données pour les architectures multi-cœurs. Réseaux et télécommunications [cs.NI]. Université de Grenoble, 2010. Français. NNT : . tel-00563868

HAL Id: tel-00563868

https://theses.hal.science/tel-00563868

Submitted on 7 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : Informatique

Arrêté ministériel : 7 août 2006

Présentée et soutenue publiquement par

Fabien GAUD

le 02 Décembre 2010

ÉTUDE ET AMÉLIORATION DE LA PERFORMANCE DES SERVEURS DE DONNÉES POUR LES ARCHITECTURES MULTI-CŒURS

Thèse dirigée par Jean-Bernard STEFANI et codirigée par Renaud LACHAIZE et Vivien QUÉMA

JURY

M. Pierre SENS	Professeur, Université Paris 6	Président
M. Pascal FELBER	Professeur, Université de Neuchâtel	Rapporteur
M. Gilles GRIMAUD	Professeur, Université de Lille	Rapporteur
M. Gilles MULLER	Directeur de recherche, INRIA	Examinateur
M. Jean-Bernard STEFANI	Directeur de recherche, INRIA	Examinateur
M. Renaud LACHAIZE	Maître de conférence, Université Joseph Fourier	Examinateur
M. Vivien QUÉMA	Chargé de recherche, CNRS	Examinateur

Thèse préparée au sein du Laboratoire d'Informatique de Grenoble, dans l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

À ma famille

À Natacha

Remerciements

Je tiens tout d'abord à remercier Pierre Sens, Professeur à l'Université Paris 6, de me faire l'honneur de présider ce jury.

Je remercie également Pascal Felber, Professeur à l'Université de Neuchâtel, et Gilles Grimaud, Professeur à l'Université des Sciences et Technologies de Lille, d'avoir accepté d'être rapporteurs de cette thèse et d'avoir évalué mon travail de manière approfondie et constructive. Merci également à Gilles Muller, Directeur de recherche à l'INRIA d'avoir accepté d'être examinateur de ce travail et d'avoir participé à ces travaux.

Je souhaite également remercier mes encadrants sans qui cette thèse n'aurait pas été possible. Tout d'abord, je remercie Jean-Bernard Stefani, Directeur de recherche à l'INRIA, de m'avoir permis de faire une thèse au sein de l'équipe Sardes et pour toute la liberté qu'il m'a laissée dans le choix de mes travaux. Je remercie également vigoureusement Renaud Lachaize, Maître de conférence à l'Université Joseph Fourier, et Vivien Quéma, Chargé de recherche au CNRS, pour m'avoir accepté en tant que stagiaire lors de mon Master puis en tant que doctorant. Je ne les remercierai jamais assez pour leur encadrement et pour leur disponibilité.

Je tiens également à remercier les personnes suivantes sans qui ces trois années de thèse n'auraient pas été aussi intéressantes :

- mes amis et collègues du bureau B218 : Fabien, Baptiste et Sylvain qui ont grandement contribué à mes travaux et à ce que ces trois années se passent dans la bonne humeur.
- toute l'équipe Sardes pour m'avoir accueilli et pour la bonne ambiance qui y règne.
- Willy pour m'avoir accompagné pendant mes travaux de Master, de thèse et pendant les séminaires de l'équipe.
- tous les enseignants avec qui j'ai pu collaborer et notamment Pierre-Yves Gibello pour m'avoir permis de participer au projet Ecom, Pascal Sicard pour ces deux années très intéressantes et toute l'équipe d'Intergiciels et Bases de Données pour m'avoir confié les clés de la partie « Intergiciels », ce qui fut une expérience très enrichissante.
- tous mes amis, et notamment Julien, Walid, Dudu et Grunch, qui ont supporté mes humeurs et mes disponibilités changeantes.

Enfin, je garde une pensée toute particulière pour toutes les personnes de ma famille, qui m'ont accompagné tout au long de mes études, et pour Natacha, qui a réussi à me supporter tout au long de mon parcours universitaire.

Table des matières

Introduction **1**

1 Émergence des processeurs multi-cœurs **5**
1.1 Processeurs mono-cœur 6
1.1.1 Principaux composants d'un processeur 6
1.1.2 Problèmes d'évolution 8
1.1.3 Vers des processeurs parallèles 8
1.2 Processeurs SMT 9
1.2.1 Fonctionnement d'une architecture SMT 9
1.2.2 Limitations 9
1.3 Processeurs multi-cœurs à accès mémoire uniforme 10
1.3.1 Fonctionnement d'une architecture multi-cœurs à accès mémoire uniforme . 10
1.3.2 Mesures de performances 12
1.3.3 Limitations 13
1.4 Processeurs multi-cœurs à accès mémoire non uniforme 14
1.4.1 Fonctionnement d'une architecture multi-cœurs à accès mémoire non uniforme 15
1.4.2 Mesures de performances 16
1.4.3 Limitations 18
1.5 Processeurs du futur 18
1.5.1 Le projet Tera-scale 19
1.5.2 Processeurs hétérogènes 20
1.6 Bilan 22

2 Modèles de programmation pour les architectures multi-cœurs **23**
2.1 Modèles de programmation classiques 24
2.1.1 Programmation à base de threads 24
2.1.2 Améliorations du modèle de threads 27
2.1.3 Programmation événementielle 29
2.1.4 Amélioration du modèle d'événements 31
2.1.5 Approches visant à unifier les deux modèles 34
2.2 Nouveaux modèles de programmation multi-cœurs 35
2.2.1 Modèles de programmation *fork-join* 35
2.2.2 Programmation par étages 36
2.2.3 Files d'éléments 37
2.2.4 MapReduce 37
2.2.5 Grand central 38
2.3 Bilan 38

3 Environnements d'exécution pour architectures multi-cœurs **41**
3.1 Repenser les systèmes d'exploitation pour les architectures multi-cœurs 41
3.1.1 Rôles principaux du système d'exploitation 42
3.1.2 Architecture des systèmes d'exploitation 44
3.1.3 Systèmes d'exploitations pour les architectures multi-cœurs 46
3.2 Repenser l'exécution des tâches sur les architectures multi-cœurs 51
3.2.1 Équilibrage de charge 52
3.2.2 Affinités et compétition 54
3.2.3 Allocation mémoire efficace 56
3.3 Bilan 58

4 Positionnement de la contribution **59**
4.1 Contexte de la contribution 59
4.2 Vol de tâches efficace pour les systèmes événementiels multi-cœurs 60
4.3 Étude et amélioration des performances du serveur Web Apache sur les architectures multi-cœurs 61

5 Vol de tâches efficace pour les systèmes événementiels multi-cœurs **63**
5.1 Description du support d'exécution Libasync-smp 64
5.1.1 Programmation événementielle colorée 64
5.1.2 Architecture du support d'exécution Libasync-smp 67
5.1.3 Algorithme de vol de tâches 69
5.1.4 Évaluation des performances du vol de tâches 71
5.2 Un nouvel algorithme de vol de tâches 73
5.2.1 Vol de tâches sensible à la localité 74
5.2.2 Vol de tâches avec analyse de la rentabilité 74
5.2.3 Vol de tâches avec pénalité 75
5.3 L'environnement d'exécution Mely 76
5.3.1 Architecture de Mely 76
5.3.2 Mise en œuvre du vol de tâches 78
5.3.3 Détails supplémentaires de mise en œuvre 79
5.4 Évaluation du mécanisme de vol de tâches de Mely 82
5.4.1 Conditions expérimentales 82
5.4.2 Micro-tests 82
5.4.3 Serveurs de données 85
5.5 Conclusion 91

6 Étude et amélioration des performances du serveur Web Apache sur une architecture multi-cœurs NUMA **93**
6.1 État de l'art de l'évaluation des serveurs Web 94
6.2 Mécanismes d'injection de charge 95
6.2.1 Caractéristiques des différents mécanismes d'injection de charge 96
6.2.2 Charge et environnement d'injection employés 98
6.3 Présentation du serveur Web Apache 100
6.3.1 Modèle d'exécution 100
6.3.2 Interaction/Intégration avec PHP 102
6.3.3 Détails de configuration 102
6.4 Configuration des expériences et métriques utilisées 104

6.4.1 Configuration matérielle 104
6.4.2 Configuration logicielle 105
6.4.3 Méthodologie et métriques utilisées 107
6.5 Étude de la performance du serveur Web Apache sur les architectures multi-cœurs . . 109
6.5.1 Mesure de performances initiale 109
6.5.2 Analyse de l'utilisation des cartes réseau 110
6.5.3 Analyse de l'utilisation des différents cœurs 111
6.5.4 Analyse du temps passé dans les fonctions 111
6.5.5 Analyse de l'efficacité des accès mémoire 112
6.5.6 Bilan 114
6.6 Première étape : Réduction des accès distants en colocalisant les processus Apache et PHP 115
6.6.1 Mesure de la performance de la solution 116
6.6.2 Analyse de l'utilisation des cartes réseau 117
6.6.3 Analyse de l'utilisation des différents cœurs 117
6.6.4 Analyse de l'efficacité des accès mémoire 119
6.6.5 Bilan 119
6.7 Deuxième étape : Équilibrage de la charge entre les nœuds en N-Copy 120
6.7.1 Mesure de la performance de la solution 121
6.7.2 Analyse de l'utilisation des cartes réseau 121
6.7.3 Analyse de l'utilisation des différents cœurs 122
6.7.4 Analyse du temps passé dans les fonctions 122
6.7.5 Analyse de l'efficacité des accès mémoire 124
6.7.6 Bilan 126
6.8 Discussion 126
6.9 Conclusion 127

Conclusion & Perspectives **129**

Bibliographie **139**

Table des figures

1.1 Architecture avec deux processeurs Intel Xeon E5420 à quatre cœurs. 11
1.2 Architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs. . . . 15
1.3 Évolution du débit mémoire en fonction du nombre de nœuds accédant à leur mémoire locale sur une architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs. 18

2.1 (a) Pseudo-code d'un serveur de données utilisant le modèle de threads (b) diagramme de séquences du même serveur pour deux connexions reçues. 25
2.2 (a) Pseudo-code de la boucle principale d'un serveur de données utilisant une programmation à base d'événements (b) Illustration de la production d'événements pour le même serveur. Les événements sont produits soit de manière interne soit depuis des requêtes réseau. 30
2.3 Exemple de programme Cilk permettant de calculer la suite de Fibonacci. 36
2.4 Exemple de phase Map et Reduce permettant de compter les mots d'un texte. 37

3.1 Architecture classique d'un système d'exploitation. 42
3.2 Possibilités d'architecture pour les systèmes d'exploitation multi-cœurs. 47
3.3 Deux organisations typiques en files d'un ordonnanceur de tâches. 53

5.1 Un exemple de coloration d'une architecture découpée en 4 traitants d'événements et utilisant 4 couleurs. 66
5.2 Code d'un serveur écho simple dans sa version mono-cœur ainsi que les changements nécessaires pour le paralléliser en utilisant des couleurs. 68
5.3 Architecture du support d'exécution événementielle multi-cœurs Libasync-smp. . . . 69
5.4 Pseudo-code illustrant le mécanisme de vol de tâches de Libasync-smp. 70
5.5 Performances du serveur de fichiers sécurisé SFS en utilisant Libasync-smp avec et sans mécanisme de vol de tâches. 72
5.6 Performance du serveur Web SWS en utilisant Libasync-smp avec et sans vol de tâches. 72
5.7 Architecture de Mely sur un cœur. 77
5.8 Passage à l'échelle de Mely avec des flots d'événements indépendants et observation du nombre d'accès au cache L2 par événement à 8 cœurs. 81
5.9 Architecture du serveur Web SWS. 86
5.10 Performance du serveur Web SWS utilisant Mely par rapport à la version Libasync-smp ainsi que d'autres serveurs Web connus. 88
5.11 Performance du serveur de fichiers sécurisé utilisant Mely par rapport à la version Libasync-smp avec et sans vol de tâches. 90

6.1 Illustration des liens entre les 4 processeurs AMD Opteron 8380 à quatre cœurs, les nœuds mémoire et les périphériques d'E/S. 105

6.2 Nombre maximum de clients pouvant être traités pour la charge SPECWeb 2005 Support. 110
6.3 Utilisation des cœurs regroupés par nœuds sur le test SPECWeb Support à 16 cœurs. 111
6.4 Utilisation des liens HyperTransport à 16 cœurs. 115
6.5 Nombre maximum de clients pouvant être traités pour la charge SPECWeb 2005 Support avec la configuration N-Copy. 116
6.6 Utilisation des cœurs regroupés par nœuds avec la configuration N-Copy. 117
6.7 Utilisation des liens HyperTransport à 16 cœurs avec la configuration N-Copy. 120
6.8 Nombre maximum de clients pouvant être traités pour la charge SPECWeb 2005 Support avec et sans mécanisme d'équilibrage de charge. 122
6.9 Utilisation des cœurs regroupés par nœuds pour la configuration avec équilibrage de charge. 123
6.10 Utilisation des liens HyperTransport à 16 cœurs pour la configuration avec équilibrage de charge. 125

Liste des tableaux

1.1 Latence des accès mémoire en fonction de la localisation de la donnée sur une architecture avec deux processeurs Intel Xeon E5420 à quatre cœurs. Le cache L2 privé et le cache L2 voisin désignent le même cache L2. 13
1.2 Débit de la mémoire centrale en fonction du nombre de cœurs sur une architecture avec deux processeurs Intel Xeon E5420 à quatre cœurs. 13
1.3 Latence des accès mémoire en fonction de la localisation de la donnée sur une architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs. 17
1.4 Débit des différentes mémoires en fonction du nombre de cœurs sur une architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs. 17

5.1 Interface de programmation de Libasync. 65
5.2 Ajouts à l'API de Libasync pour prendre en compte la coloration d'événements. . . . 65
5.3 Comparaison du temps passé à voler des événements par rapport au temps passé à exécuter ces événements volés dans le cas du serveur Web SWS et du serveur de fichiers SFS. 73
5.4 Évolution de l'API pour fournir au programmeur la possibilité de fixer le temps estimé d'un événement ainsi que la pénalité sur un traitant. 79
5.5 Impact du mécanisme de vol de tâches de Mely sur les performances. 83
5.6 Impact de l'heuristique de vol VAR sur la performance du vol de tâches. 83
5.7 Impact de l'heuristique de vol VAP sur les performances du vol de tâches. 84
5.8 Impact de l'heuristique de vol VSAL sur les performances du vol de tâches. 85

6.1 Répartition observée de l'accès aux ressources avec l'injection de charge SPECWeb 2005 Support. 99
6.2 Répartition des parts de marché entre les 5 principaux serveurs Web. 100
6.3 Impact du nombre de processus PHP sur le débit du serveur Apache à 16 cœurs et 30000 clients. 103
6.4 Principaux paramètres de configuration du serveur Web Apache. 103
6.5 Principaux paramètres de configuration du système d'exploitation. 106
6.6 Débit total observé en fonction du nombre de cœurs sur un micro-test simulant une injection SPECWeb. 107
6.7 Utilisation moyenne et maximale des cartes réseau pour la charge SPECWeb Support. 110
6.8 Nombre de cycles par octet transmis dans les dix principales fonctions ainsi que leur augmentation par rapport à l'exécution sur quatre cœurs. 112
6.9 Évolution du nombre d'instructions par cycle sur le serveur Web Apache en fonction du nombre de cœurs. 113
6.10 Nombre de fautes de cache par requête traitée en fonction du nombre de cœurs. . . . 113
6.11 Taux de requêtes vers un banc de mémoire locale par rapport à la totalité des accès mémoire. 114

6.12 Taux d'utilisation moyen et maximal des cartes réseau pour la charge SPECWeb Support en colocalisant les processus Apache et PHP avec la configuration N-Copy. . . . 117
6.13 Débit mémoire en fonction du placement de la zone mémoire écrite. Un seul cœur par noeud accède à la mémoire. 118
6.14 Évolution de l'IPC en fonction du nombre de cœurs avec la configuration N-Copy. . . 119
6.15 Taux de requête à une mémoire locale par rapport à la totalité des accès mémoire avec la configuration N-Copy. 119
6.16 Utilisation moyenne et maximal des cartes réseau pour la configuration avec équilibrage de charge. 121
6.17 Temps passé par octet transmis dans les dix principales fonctions ainsi que leur augmentation par rapport à l'exécution à quatre cœurs pour la configuration avec équilibrage de charge. 123
6.18 Évolution de l'IPC en fonction du nombre de cœurs pour la configuration avec équilibrage de charge. 124
6.19 Taux de requêtes à une mémoire locale par rapport à la totalité des accès mémoire avec la configuration N-Copy. 124
6.20 IPC observé des 10 principales fonctions pour la configuration avec équilibrage de charge. 125
6.21 Augmentation de la latence observée pour des accès depuis le nœud 0 vers les différents nœuds par rapport à la version 4 cœurs. 127

Introduction

Contexte des travaux

L'évolution des processeurs a pris ces quatre dernières années une tournure radicalement différente des tendances en vigueur pendant longtemps. Auparavant, pour faire évoluer les processeurs, les fabricants comptaient essentiellement sur une augmentation de la fréquence de ces derniers ainsi que sur des optimisations microarchitecturales (calculs effectués dans le désordre, *pipeline*). Cependant, au fur et à mesure de l'évolution des processeurs, complexifier l'architecture interne est devenu de plus en plus difficile. Par exemple, le processeur Intel Pentium 4 disposait d'un *pipeline* extrêmement long dont il était difficile de tirer parti efficacement. L'augmentation de la fréquence de ces processeurs est devenue de plus en plus difficile, car une augmentation de la fréquence implique également une augmentation de la dissipation thermique ainsi que de la consommation du processeur. Pour toutes ces raisons, les concepteurs de processeurs ont changé de stratégie et ont choisi de proposer des puces contenant non plus un mais plusieurs cœurs d'exécution pour pouvoir exécuter plusieurs flots d'exécution en simultané. Bien que chaque cœur soit moins performant individuellement, l'union de tous les cœurs permet d'augmenter la puissance globale du processeur tout en limitant sa consommation et son dégagement de chaleur. Le futur des processeurs semble être une parallélisation toujours plus importante avec éventuellement l'introduction de cœurs hétérogènes en fréquence ou en fonctionnalités.

Auparavant réservées aux machines onéreuses effectuant généralement des calculs scientifiques, les architectures parallèles deviennent petit à petit incontournables. Ces nouvelles architectures impliquent de repenser tous les logiciels pour pouvoir les exploiter efficacement. Cette refonte doit être faite en profondeur depuis les abstractions fournies au programmeur à la manière de concevoir des applications en passant par les systèmes d'exploitation et les environnements d'exécution. Par conséquent, le spectre des travaux devant être menés dans ce domaine est très large.

Les serveurs de données (serveurs Web, serveurs de fichiers, etc.) sont un type d'applications extrêmement répandu. En terme de performances, l'objectif de ces applications est de minimiser la latence perçue par les clients tout en maximisant le nombre de clients pouvant être traités simultanément. Pour atteindre ces objectifs, il est impératif d'utiliser toute la puissance fournie par l'architecture sous-jacente. Par conséquent, les serveurs de données doivent être capables de tirer parti au maximum du parallélisme proposé par les architectures multi-cœurs. Bien évidemment, il est souvent possible de dupliquer le serveur sur plusieurs machines physiques pour résoudre les problèmes de montée en charge. Toutefois, être capable d'utiliser efficacement une machine permet de réduire le nombre total de machines nécessaires et ainsi de réduire la place occupée ainsi que les différents coûts, notamment la consommation électrique. La réduction du nombre de machines utilisées permet de baisser également le coût de maintenance global. Enfin, cette classe d'application se prête bien à la parallélisation massive, puisqu'il est généralement possible de traiter indépendamment des requêtes issues de clients différents.

Objectifs & Contributions

L'objectif principal de ces travaux est de faire un état des lieux de la performance des serveurs de données sur les architectures multi-cœurs, ainsi que de faire différentes propositions pour améliorer ces performances. La première partie consiste a étudier un modèle de programmation émergent tandis que la seconde consiste à voir comment il est possible de modifier un serveur Web réputé pour tirer parti de ces nouvelles architectures. Plus précisément, nous avons étudié le problème du passage à l'échelle des serveurs de données à deux niveau différents des couches logicielles :

- *Au niveau de l'environnement d'exécution.* L'environnement d'exécution, et notamment l'ordonnanceur de tâches, est un point crucial dans la recherche de l'amélioration de la performance des applications sur les architectures multi-cœurs. Nous avons choisi de travailler sur l'exécution de programmes événementiels. La programmation événementielle, et tout particulièrement la programmation événementielle colorée capable d'utiliser les architectures multi-cœurs, est un modèle de programmation qui nous semble être un concurrent crédible à la programmation par threads, notamment de part sa simplicité dans la gestion des accès concurrents aux ressources partagées. Un des objectifs de l'ordonnanceur de tâches est de maximiser le taux d'utilisation de chacun des cœurs disponibles. Pour arriver à cet objectif, une des méthodes est d'utiliser un mécanisme de vol de tâches. Les travaux menés sur ce sujet consistent en l'analyse des problèmes de ce mécanisme dans le cadre de la programmation événementielle colorée ainsi qu'en la proposition de solutions pour améliorer ses performances et son efficacité. Nous analysons la performance de nos solutions sur deux serveurs de données différents : un serveur Web et un serveur de fichiers. L'originalité de cette contribution se situe essentiellement dans deux aspects. Premièrement, les mécanismes de vol de tâches n'avaient jusqu'alors été évalués que dans le cadre d'applications de calculs scientifique. Nous montrons dans ces travaux que ce mécanisme peut améliorer sensiblement les performances des serveurs de données, mais peut également grandement les baisser s'il n'est pas employé avec précaution. Deuxièmement, le vol de tâches n'a réellement été évalué que dans le cadre de la programmation par threads. La programmation événementielle colorée a des contraintes qu'il est nécessaire de prendre en compte dans ce mécanisme. Ces travaux ont donné lieu à deux publications dans des conférences, l'une nationale [47] et l'autre internationale [46].

- *Au niveau applicatif.* Dans la seconde partie de nos travaux, nous avons choisi d'étudier le comportement du serveur Web Apache sur ces nouvelles architectures multi-cœurs dans des conditions de charge réelles. Le serveur Web Apache est le serveur le plus déployé à l'heure actuelle : plus de la moitié des sites Web l'utilise. Par conséquent, comprendre le comportement de ce serveur sur les architectures multi-cœurs est un sujet à fort impact. Développé en utilisant un thread par connexion, le serveur devrait être en mesure d'utiliser efficacement le parallélisme offert par les processeurs modernes. Notre première contribution est de montrer que le passage à l'échelle d'Apache n'est pas idéal et d'identifier les causes de ces mauvaises performances. Nous proposons ensuite un ensemble d'améliorations et montrons qu'il est nécessaire de prendre en compte la topologie de l'interconnexion entre les processeurs pour obtenir de bonnes performances. Enfin, après avoir méthodiquement analysé un ensemble de métriques importantes, nous proposons une discussion sur la possibilité de gains supplémentaires et sur les pistes à creuser pour les obtenir. L'originalité de ces travaux par rapport aux travaux précédents se situe dans l'architecture matérielle étudiée, une architecture NUMA à 16 cœurs, ainsi que dans les conclusions et les propositions effectuées.

Organisation du document

Ce document est structuré en six chapitres. Parmi ces six chapitres, les trois premiers présentent l'état de l'art des recherches autour du domaine des architectures multi-cœurs, le quatrième positionne les travaux par rapport à ces recherches et les deux suivants présentent nos contributions.

Le premier chapitre a pour thème l'évolution des processeurs. Il montre comment les processeurs ont évolué pour devenir petit à petit de plus en plus parallèles. Nous donnons également dans ce chapitre des indications sur les perspectives de recherche dans le domaine des multi-cœurs.

Le second chapitre présente les deux grands modèles de programmation que sont le modèle de programmation à base de threads et celui à base d'événements. Nous expliquons dans ce chapitre les limites de ces modèles et quelles sont les évolutions nécessaires pour pouvoir utiliser efficacement le parallélisme fourni par les processeurs. Nous détaillons les modèles spécialisés pour ces architectures.

Le troisième chapitre fait le point sur les environnements d'exécution. Dans ce chapitre, nous présentons premièrement le rôle et les concepts fondamentaux des systèmes d'exploitation. Deuxièmement, nous décrivons ensuite les travaux menés sur le passage à l'échelle de ces systèmes d'exploitation. Nous présentons enfin l'état de l'art sur l'ordonnancement des tâches sur les différents cœurs ainsi que sur les mécanismes d'allocation mémoire efficaces.

Le quatrième chapitre a pour objectif d'introduire nos travaux et de les situer par rapport aux travaux présentés dans les trois chapitres précédents.

Le cinquième chapitre présente les travaux que nous avons menés sur l'étude de la performance du mécanisme de vol de tâches sur les serveurs de données dans le contexte de la programmation événementielle multi-cœurs. Nous proposons dans ce chapitre un ensemble d'heuristiques ainsi qu'une modification de l'architecture du support d'exécution pour améliorer les performances de ce mécanisme.

Le sixième et dernier chapitre se concentre sur l'étude du passage à l'échelle du serveur Web Apache. Après avoir montré que ce dernier ne passe pas à l'échelle, nous introduisons un ensemble de solutions pour améliorer les performances du serveur et étudions leurs impacts.

Après ce dernier chapitre, nous donnons une conclusion générale à tous nos travaux ainsi qu'un ensemble de perspectives pour des travaux futurs.

Chapitre 1

Émergence des processeurs multi-cœurs

Sommaire

1.1 Processeurs mono-cœur 6

1.1.1 Principaux composants d'un processeur 6

1.1.2 Problèmes d'évolution 8

1.1.3 Vers des processeurs parallèles 8

1.2 Processeurs SMT 9

1.2.1 Fonctionnement d'une architecture SMT 9

1.2.2 Limitations 9

1.3 Processeurs multi-cœurs à accès mémoire uniforme 10

1.3.1 Fonctionnement d'une architecture multi-cœurs à accès mémoire uniforme 10

1.3.2 Mesures de performances 12

1.3.3 Limitations 13

1.4 Processeurs multi-cœurs à accès mémoire non uniforme 14

1.4.1 Fonctionnement d'une architecture multi-cœurs à accès mémoire non uniforme 15

1.4.2 Mesures de performances 16

1.4.3 Limitations 18

1.5 Processeurs du futur 18

1.5.1 Le projet Tera-scale 19

1.5.2 Processeurs hétérogènes 20

1.6 Bilan 22

Les processeurs ont connu une évolution rapide pendant des années. Les gains de performances provenaient essentiellement de l'augmentation de la fréquence ainsi que du raffinement de leur architecture. Ces deux paramètres sont de plus en plus difficiles à faire évoluer. Les architectures parallèles sont donc devenues la norme chez les fabricants de processeurs. Par exemple, le grand public a maintenant accès à des processeurs possédant jusqu'à 4 cœurs.

Dans cette section, nous allons dans un premier temps détailler les raisons de cette évolution. Nous étudierons ensuite les changements architecturaux effectués au sein des processeurs et verrons leurs limitations.

1.1 Processeurs mono-cœur

Un processeur est une puce permettant de faire des opérations arithmétiques et logiques plus ou moins complexes sur des données. Les processeurs ont évolué pour devenir des puces extrêmement complexes. En effet, la conjecture de Moore, valide depuis les années 1970, prédit le doublement de la densité des transistors au sein des processeurs tous les deux ans. Cependant, si durant une longue période il a été possible d'augmenter les performances des applications en améliorant l'architecture du cœur des processeurs ainsi que leur fréquence (nombre de cycles par unité de temps), les fondeurs se sont par la suite heurtés à une série de problèmes.

Dans les parties suivantes, nous allons détailler les grandes lignes du fonctionnement d'un processeur. Nous verrons ensuite quels ont été les problèmes à résoudre pour les fabricants de processeurs.

1.1.1 Principaux composants d'un processeur

Un processeur est chargé d'effectuer des calculs sur des opérandes. Ces calculs sont effectués par les unités de calcul. Ces unités de calcul sont notamment l'*ALU* (l'unité arithmétique et logique) et la *FPU* (l'unité utilisée pour les calculs en virgule flottante). Le processeur fournit au programmeur un jeu d'instructions (ISA, *Instruction Set Architecture*) pour pouvoir exprimer ses opérations. Actuellement, dans les processeurs grand public, le plus utilisé est le jeu d'instructions *x86*. Ce jeu d'instructions permet de manipuler des données sur 32 bits (ou moins). Des instructions vectorielles permettant de manipuler des données sur 128 bits ont également été introduites. Ces instructions permettent d'appliquer la même opération à plusieurs données en simultanée (ce sont, par exemple, les extensions *SSE*). Ce fonctionnement est appelé *SIMD (Single Instruction Multiple Data)*. Les adresses sont exprimées sur 32 bits, ce qui limite la capacité d'adressage à 4 Go. Cette taille maximum étant maintenant atteinte, une extension du jeu d'instructions permet de manipuler des données et des adresses sur 64 bits. Il s'agit du jeu d'instructions *x86_64*, utilisé à la fois par Intel et AMD.

Les opérations réalisées par l'unité de calcul s'effectuent généralement sur des registres. Les registres sont des zones de mémoire permettant de stocker temporairement une donnée pour effectuer des calculs dessus. Les registres ont un temps d'accès d'une instruction par cycle. En contrepartie, très peu de registres sont disponibles. Par exemple, dans l'architecture *x86_64*, 16 registres généraux sont disponibles en mode 64 bits (il y a également des registres d'état ainsi que des registres spécialisés).

Pour effectuer des opérations, le processeur a besoin de récupérer les données depuis la mémoire centrale ainsi que d'y stocker les résultats. La latence d'accès (c'est-à-dire le temps nécessaire pour accéder à ces données) est devenue un point important de ralentissement des processeurs. En effet, le débit de la mémoire a évolué à une vitesse moindre que le nombre de cycles par unité de temps. Pour résoudre ces problèmes, les fabricants de processeurs ont intégré diverses améliorations à leur architecture.

Unité de gestion de la mémoire virtuelle. Sur un processeur, les applications sont souvent multiplexées pour donner l'illusion d'une exécution parallèle. Cependant, pour des raisons de sécurité, une application ne doit pas pouvoir accéder aux données d'une autre application. Pour résoudre ce problème, les applications vont manipuler de la mémoire virtuelle. Pour gérer la mémoire virtuelle, la méthode qui est généralement utilisée est la méthode de pagination. La mémoire est découpée en pages (c'est-à-dire en zones mémoire) de taille fixe. L'application ne voit que des adresses virtuelles et le système d'exploitation s'occupe de gérer la table des pages (qui fait la correspondance entre adresses virtuelles et adresses physiques).

Au sein du processeur, l'unité chargé de la gestion de la mémoire est la *MMU (Memory Management Unit)*. La *MMU* fournit notamment un cache de la table des pages, appelé TLB (*translation*

lookaside buffer). Lorsqu'une entrée n'est pas dans le TLB, le processeur va aller chercher la donnée dans la table des pages du système d'exploitation. Le placement mémoire de cette table est généralement connu grâce à un registre d'état. Le TLB peut être annoté avec le numéro du processus ou non. Dans le cas où il n'est pas annoté, il est donc nécessaire de vider le TLB à chaque changement d'application. Le second cas est celui rencontré dans les architectures x86.

Caches. Pour réduire les latences d'accès mémoire, une approche classique consiste à intégrer des mémoires de petite taille très proches de l'unité de calcul de manière à réduire les latences d'accès aux données. Plusieurs niveaux de cache sont également embarqués sur le processeur. Deux niveaux de caches (L1 et L2) sont utilisés généralement dans les processeurs mono-cœurs [1]. Le cache L1 est scindé en un cache pour les instructions et un cache pour les données. Il a une latence de quelques cycles, mais une taille très restreinte. À l'inverse le cache L2 à une taille plus importante, mais également une latence d'accès plus grande. Il est souvent unifié (c'est-à-dire qu'il contient à la fois les données et les instructions).

Il est possible d'utiliser différentes stratégies pour gérer les caches L1 et L2. La première consiste à rendre le cache L2 inclusif, c'est-à-dire qu'il contient également les données qui sont dans le cache L1. La seconde est de rendre le cache exclusif, c'est-à-dire qu'un zone mémoire n'est jamais conjointement dans les deux caches. Lorsque le processeur cherche à accéder à une donnée, soit la donnée est en L1 (succès du cache L1) soit elle n'y est pas (échec du cache L1). Dans le deuxième cas, le processeur va ensuite interroger le cache L2. Dans le cas où la donnée n'est pas dans ce cache, le processeur va alors aller la chercher en mémoire et la copier dans le cache L1. Cette copie peut nécessiter de remplacer une ligne de cache déjà présente. La politique de remplacement généralement appliquée est de remplacer la donnée utilisée le moins récemment (aussi appelée politique LRU, *least recently used*). Dans le cas d'un cache exclusif, cela provoque la copie de cette ligne dans le cache L2 (provoquant potentiellement une éviction dans ce cache). Le remplacement d'une ligne de cache est limité par l'associativité du cache. Un cache *associatif à N entrées* (*N-way*) laisse N possibilités de remplacement. N représente le nombre de lignes de cache associées à une adresse. L'algorithme de remplacement choisira une ligne parmi ces N possibilités.

Le choix de l'associativité est un facteur déterminant de la performance des caches. Plus N est petit, moins le nombre de possibilités de remplacement est important mais plus l'algorithme de remplacement est rapide. À l'inverse, plus N est grand, plus le nombre de possibilités de remplacement est grand et permettra un remplacement pertinent. Cependant, le coût de l'algorithme de remplacement sera plus important. Par conséquent, les caches L1 ont un N généralement petit pour avoir des temps d'accès faibles au détriment de la taille de stockage. À l'inverse, les caches L2 ont un N plus grand pour trouver un bon compromis entre taille et latence.

Avec un cache, les accès mémoire fonctionnent de la manière suivante. Lorsque le processeur a besoin d'une donnée, il va d'abord regarder son cache L1 et également demander la traduction de l'adresse virtuelle en adresse physique au TLB. Le cache L1 peut être interrogé avec l'adresse virtuelle (et marqué avec l'adresse physique), ce qui permet de paralléliser ces deux opérations. Si l'adresse n'est pas dans le cache L1, le processeur va alors vérifier si la donnée est dans le cache L2 et ensuite va éventuellement accéder à la mémoire.

Notons que la taille d'une ligne de cache est généralement plus grosse que la taille d'une donnée ce qui fait que plusieurs données sont récupérées en même temps. Une application avec une bonne localité spatiale va ainsi voir ses accès à la mémoire réduits.

Enfin, pour améliorer encore la possibilité d'avoir la donnée en cache, le processeur essaye de détecter le modèle d'accès mémoire et, en fonction d'heuristiques internes, charge automatiquement

1. Certaines machines peuvent également proposer un cache de troisième niveau généralement sur la carte mère

en cache les données pouvant être accédées par la suite (mécanisme de *prefetch*).

La parallélisation des instructions. Pour améliorer l'efficacité des accès à la mémoire, une solution est de pouvoir effectuer d'autres instructions pendant qu'une instruction est en attente de données. Pour cela, le processeur dispose d'un *pipeline* permettant d'exécuter plusieurs instructions indépendantes en parallèle. L'efficacité de cette technique est d'autant plus grande que le processeur est capable d'exécuter les instructions dans un ordre différent de celui spécifié par le programmeur (tout en garantissant la cohérence des données manipulées) et de faire de la prédiction de branchement. La prédiction consiste à extrapoler le résultat d'un branchement pour charger le pipeline avec la suite des instructions à exécuter. Si la prédiction de branchement s'avère mauvaise, le pipeline doit nécessairement être vidé, ce qui entraîne un coût supplémentaire. L'efficacité de la prédiction de branchement est donc un critère important pour la performance des processeurs.

Ces optimisations permettent d'exécuter sur les processeurs modernes entre 3 et 5 instructions par cycle. Ces processeurs capables d'effectuer plusieurs instructions par cycle sont appelés *processeurs superscalaires*. Le parallélisme des instructions ainsi engendré est appelé *ILP* (*Instruction Level Parallelism*).

1.1.2 Problèmes d'évolution

L'évolution des processeurs a connu une période faste. En effet, pendant plusieurs générations, l'accélération des processeurs a essentiellement consisté à complexifier encore ceux-ci (avec par exemple la prédiction de branchement, les pipelines ou encore l'exécution désordonnée) et à augmenter leur fréquence. Ces approches permettaient d'améliorer les performances des applications sans avoir à les modifier. Changer de processeur était donc l'assurance pour les utilisateurs d'un gain de performance.

Cependant, cette évolution s'est heurtée à plusieurs problèmes. Premièrement, l'augmentation constante de la fréquence des processeurs ainsi que de la densité des transistors a entraîné une augmentation importante de la consommation et du dégagement thermique (TDP, *Thermal Design Power*). Par exemple, le TDP des processeurs Intel Pentium 4 (sans *Hyper-Threading*, voir section 1.2.1) varie entre 47 Watts et 89 W alors que la version précédente, l'Intel Pentium 3, variait entre 16 Watts et 43 Watts. Deuxièmement, le mécanisme de pipeline, bien que très efficace, ne peut que difficilement évoluer. En effet, la parallélisation automatique des instructions est un problème très complexe à résoudre à cause de la dépendance entre les opérations. Par exemple, la version *Prescott* du Pentium 4 embarque un pipeline très long de 31 étages rendant ainsi les vidages du pipeline très coûteux.

1.1.3 Vers des processeurs parallèles

Devant ces difficultés d'évolution, les fabricants de processeurs ont changé leur stratégie en parallélisant les processeurs. L'ILP au sein des processeurs a été le premier degré de parallélisme étudié pour compenser les temps d'attente vers la mémoire. Cette technique de parallélisation automatique est complexe à faire évoluer. Les processeurs ont, par la suite, exporté ce parallélisme aux applications. Le principe est de pouvoir exécuter plusieurs applications (ou flots d'exécution d'une même application) en parallèle. Dans ce cas, contrairement à l'ILP, le parallélisme devient explicite et la gestion de la cohérence est déportée sur l'application. Pour pouvoir gérer simultanément plusieurs flots d'exécution, il est nécessaire de dupliquer tout ou partie d'une unité d'exécution (cœur).

Il est intéressant de remarquer que cette approche parallèle avait déjà été étudiée dans les systèmes multiprocesseurs symétriques *(SMP, Symmetric multiprocessing)* et que de nombreuses techniques sont donc similaires.

Dans la suite de ce chapitre, nous allons voir quelles sont les orientations choisies pour exécuter plusieurs flots de contrôle en simultané et les problèmes engendrés par chacune des approches.

1.2 Processeurs SMT

Dans la précédente section, nous avons vu que l'évolution des processeurs passe par l'introduction du parallélisme explicite au sein de ceux-ci. La première approche étudiée est de pouvoir exécuter plusieurs flots en simultané au sein d'un processeur (ces flots d'exécution sont appelés *threads* et leur fonctionnement sera décrit ultérieurement). Notons que les threads peuvent appartenir à la même application ou à deux applications distinctes et ne partagent pas forcément un même espace de mémoire virtuelle. La création de threads peut être implicite (le parallélisme est par exemple détecté par le compilateur) ou explicite (dans ce cas, l'application a été écrite en utilisant plusieurs threads).

Diverses méthodes existent pour exécuter plusieurs threads en parallèle sur un processeur. La première consiste à entrelacer les instructions des threads. Dans ce cas, à chaque cycle d'horloge une instruction d'un thread différent est exécutée. La seconde méthode consiste à changer de thread lorsque ce dernier effectue une opération bloquante (par exemple un accès à la mémoire). Ces deux méthodes ne permettent d'avoir qu'un seul thread actif au même instant. La dernière méthode est d'exécuter plusieurs threads simultanément. Cette approche, appelée *SMT (Simultaneous multithreading)* a été évaluée comme celle apportant les meilleures performances. Dans cette section, nous ne nous intéresserons qu'à cette solution, car c'est celle choisie dans les processeurs modernes. Ungerer *et al.* ont publié en 2003 une vue d'ensemble de l'état des processeurs multi-threads [94].

1.2.1 Fonctionnement d'une architecture SMT

Pour pouvoir fournir du parallélisme par thread, il est nécessaire de dupliquer certaines parties du processeur. Par exemple, il est nécessaire que chacun des threads dispose de ses propres registres (à la fois les registres généraux et les registres d'état) pour que les threads puissent avancer en parallèle. Le contrôleur d'interruptions (*APIC*, Advanced Programmable Interrupt Controller) est également dupliqué. Ces duplications font que les deux contextes apparaissent au système comme deux processeurs. À l'inverse, certaines unités, telles que les unités d'exécution (ALU, FPU) ou encore les caches de différents niveaux sont mis en commun. Enfin d'autres composants, tels que le TLB, utilisent le numéro du processeur logique pour annoter les données. En effet, il est nécessaire de marquer le TLB car deux processus différents peuvent être ordonnancés simultanément. Un des avantages de cette duplication partielle est d'augmenter faiblement la taille de la puce (seulement 5 % sur les processeurs Intel).

Les processeurs SMT ont été introduits en 2002 par Intel pour le grand public sous le terme *Hyper-Threading*. Cette dénomination désigne un processeur capable d'exécuter deux threads en parallèle. Après son abandon suite à l'introduction des processeurs multi-cœurs, cette technologie a fait son grand retour avec l'architecture Nehalem, chaque cœur pouvant exécuter deux threads. D'autres processeurs utilisent une architecture SMT pour leurs cœurs tels que l'architecture *POWER* d'IBM (depuis sa version 5) ou encore l'architecture Niagara de SUN. Ces processeurs sont orientés pour des utilisations de type serveur de données et sont très coûteux.

1.2.2 Limitations

Marr *et al.* [67] ont évalué le processeur Xeon en activant puis en désactivant l'*Hyper-Threading*. Leurs résultats ont montré que sur des applications de type serveur Web ou base de données, les performances pouvaient être améliorées de 16 % à 28 % en utilisant l'*Hyper-Threading*.

Cependant, d'autres études telles que celle de Curtis-Maury *et al.* [33] ont mis en évidence que les résultats pouvaient être plus mitigés. Les tests ont été effectués en utilisant l'implantation OpenMP de la série de tests NAS [55]. Cette suite de tests simule des applications scientifiques. Dans ces tests, les performances des applications peuvent parfois être augmentées, parfois être dégradées lors de l'activation de l'*Hyper-Threading*. Le problème vient essentiellement de la contention sur les ressources partagées. En effet, étant donné que deux threads accèdent maintenant à un TLB commun, ce dernier est moins efficace et engendre plus d'échecs. Une conclusion similaire est faite sur la baisse du taux de succès dans le cache L2. Ces évictions de cache, dues au partage, produisent finalement un effet contraire à celui escompté. En effet, les mesures ont révélé que le processeur était plus souvent en attente lorsque l'*Hyper-Threading* est activé. Cependant, bien que le taux de succès dans le cache (ainsi que dans le TLB) diminue fortement, certaines applications peuvent quand même arriver à tirer parti du parallélisme proposé par l'*Hyper-Threading*.

Un second problème, soulevé par les auteurs de McRT [81], est que l'application doit être consciente d'utiliser un processeur SMT. En effet, si une application fait une boucle d'attente active, il est nécessaire d'utiliser des instructions telles que *mwait* pour favoriser la progression des autres threads et éviter de consommer des ressources inutilement.

Déterminer si une application peut tirer parti d'un processeur SMT est donc une tâche compliquée. Il faut notamment prendre en compte le fait que le processeur est souvent en attente de données depuis la mémoire. Dans le cas contraire, il n'est pas possible de mieux utiliser les unités de calcul du processeur (spécialement si l'application utilise des instructions de type *SSE* qui permettent un bon remplissage du pipeline). Il faut également évaluer le coût supplémentaire dû aux fautes de cache plus fréquentes. Enfin, il est nécessaire qu'un des threads ordonnancés en parallèle ne bloque pas inutilement la progression des autres threads et de vérifier que les threads ordonnancés en parallèle ne saturent pas les bus (mémoire et E/S).

1.3 Processeurs multi-cœurs à accès mémoire uniforme

Nous avons vu dans la précédente section les limitations des architectures SMT (et principalement celles de l'*Hyper-Threading*). Une autre direction a été prise par la suite par les fabricants de processeurs. L'idée a été d'introduire plusieurs cœurs au sein du processeur. Contrairement au SMT, la totalité logique du processeur est dupliquée (à l'exception éventuellement de certains caches) dans des *cœurs* distincts.

Ces processeurs partagent la même approche que les systèmes multiprocesseurs, auparavant réservés aux coûteuses architectures de serveurs de données et de calcul. Cependant, le dégagement thermique et la consommation d'une architecture multi-cœurs sont bien inférieurs à celle d'une architecture multiprocesseur. De plus, l'intégration de plusieurs cœurs sur une même puce peut améliorer la vitesse de communication entre ces derniers. Cependant, la problématique de gestion de plusieurs cœurs est à de nombreux égards identique à celle de la gestion de plusieurs processeurs.

Dans cette partie, nous détaillerons notamment le fonctionnement d'une machine équipée de deux processeurs Xeon E5420 disposant de quatre cœurs chacun. Cette architecture sera utilisée pour une partie des expériences décrites ultérieurement dans ce document.

1.3.1 Fonctionnement d'une architecture multi-cœurs à accès mémoire uniforme

Le principe des architectures multi-cœurs est de dupliquer les cœurs d'exécution au sein du processeur. Le processeur est dans ce cas vu comme deux processeurs indépendants par le système d'exploitation. Par exemple, le TLB est dupliqué sur chacun des cœurs, de même que les mécanismes de

préchargement automatique de la mémoire (*prefetcher*) ou les mécanismes de prédiction de branchements.

La figure 1.1 présente l'architecture d'une machine bi-processeurs, chacun de ces processeurs étant lui-même composé de 4 cœurs. Sur cette figure, les cœurs sont numérotés tel que le remonte un système d'exploitation Linux 2.6.24. Notons toutefois que cette numérotation peut être différente selon le système d'exploitation (et même selon la version du système d'exploitation).

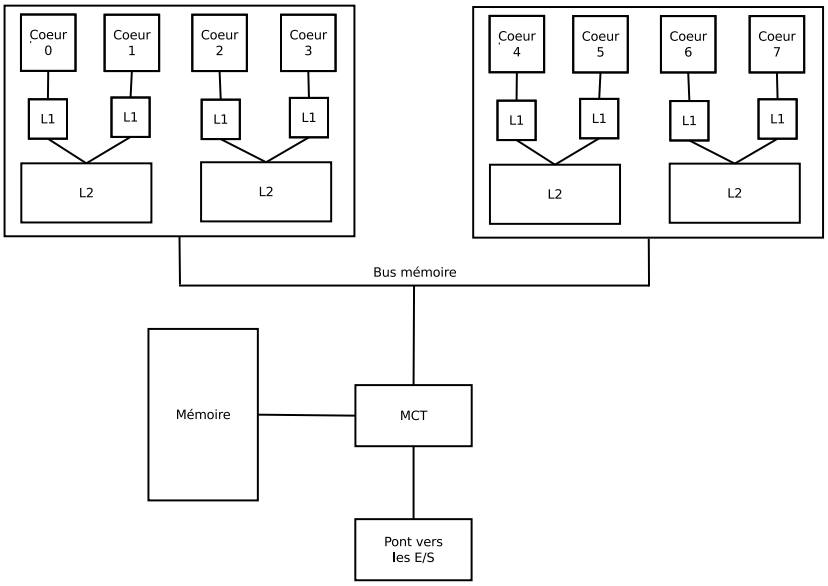


Figure 1.1 – Architecture avec deux processeurs Intel Xeon E5420 à quatre cœurs.

Sur cette figure, nous pouvons voir que chaque cœur dispose de son cache de premier niveau. En fait, sur cette architecture le cache L1 n'est pas unifié, la moitié du cache L1 est réservée aux instructions et l'autre moitié réservée aux données. Le cache L1 à une taille de 64Ko, soit 32Ko pour les instructions et 32Ko pour les données. Chaque ligne de cache peut contenir 64 octets et les deux caches L1 sont associatifs à 8 entrées.

Contrairement au cache L1, le cache L2 est partagé par groupe de deux cœurs (nous appellerons par la suite *voisin* un cœur avec lequel un cache est partagé). Ce cache L2 à une taille importante de 6 Mo (soit 12 Mo de cache L2 par processeurs). Tout comme le cache L1, la taille des lignes de cache est de 64 octets. L'associativité est plus importante que pour le cache L1 puisque ce cache est associatif à 24 entrées. Le cache L2 n'est pas forcement utilisé de façon homogène par les différents cœurs, c'est-à-dire qu'il n'y a pas de contrôle d'équité au niveau de l'utilisation du cache par les coeurs. Dans cette architecture, le cache L2 est non inclusif. Un cache non inclusif est un cache qui ne garantit ni que les données en L1 seront en L2 ni que les données en L1 ne seront pas en L2.

Tout comme dans les architectures SMP, il est nécessaire de gérer correctement la cohérence de cache. Le problème est le suivant : le cœur *0* accède à l'adresse x, il va donc faire une copie de cette valeur dans son cache. Si le cœur 2 accède à cette adresse x, il va également faire une copie de cette valeur dans son cache. Dans ce cas, les deux cœurs possèdent des copies disjointes de la même donnée et vont donc faire des modifications sans prendre en compte les modifications de l'autre cœur. Le protocole de cohérence de cache permet de résoudre ce problème. L'idée est qu'une ligne de cache garde un état (modifiée ou non, partagée ou non, valide ou invalide). Pour savoir si une ligne est partagée, chaque cœur va écouter le bus mémoire (*bus snooping*). Dans cette architecture, il est aisé d'écouter le bus mémoire, car il est partagé entre les cœurs. Lorsque le cœur modifie sa donnée, si cette

ligne est partagée il faut tout d'abord invalider les autres copies. Si cette ligne a été invalidée, il faut d'abord récupérer la donnée depuis la mémoire. Enfin, à partir du moment où une ligne est modifiée dans le cache, il faut écouter le bus mémoire pour intercepter toute demande de lecture de cette ligne depuis la mémoire (et, dans ce cas, resynchroniser les données) pour que le cœur demandeur reçoive une valeur à jour.

Sur les architectures de ce type, les accès mémoire sont uniformes (comme sur les multiprocesseurs), c'est-à-dire que tous les cœurs ont la même latence pour accéder à n'importe quelle zone de la mémoire. Ces architectures sont appelées UMA (*Uniform Memory Access*). On peut voir sur la figure 1.1 que pour accéder à la mémoire, il faut passer par le contrôleur mémoire (*MCT*, memory controller).

Les accès mémoire fonctionnent de manière identique à un processeur mono-cœur. En plus de vérifier si la donnée est dans son cache L1 ou dans le cache L2 partagé, le cœur va essayer d'interroger successivement le cache L1 du cœur voisin puis les caches des cœurs de la même puce et enfin les caches des cœurs des autres puces. Enfin, si la donnée n'est dans aucun cache, un accès à la mémoire centrale sera effectué.

Un autre point important concerne la gestion du TLB. Le TLB agit comme un cache de la table des pages pour accélérer les traductions des adresses virtuelles vers des adresses physiques. De manière similaire aux caches du processeur, lorsqu'une page est libérée, il est nécessaire d'invalider cette entrée dans les TLB de tous les cœurs.

Enfin, il est également important de noter que, dans l'architecture étudiée, le bus d'E/S passe par le contrôleur mémoire.

1.3.2 Mesures de performances

Dans cette partie, nous allons nous intéresser à la latence des différents caches ainsi qu'au débit de la mémoire. Nous avons mesuré la latence en utilisant les outils fournis par Corey [100] (et également utilisés par le projet Barrelfish [18]). Le tableau 1.1 présente les résultats obtenus. Ce tableau montre notamment que les latences d'accès au L1 et au L2 sont très faibles. Ces tests ont été effectués 100 fois et l'écart type est faible (excepté pour le test accès au cache L1 d'un voisin).

L'accès à un cache local permet de diviser le temps d'accès mémoire par 10 si la donnée est en L2 et par près de 90 si la donnée est en L1. Ce tableau est également en phase avec l'enchaînement des demandes décrit dans la partie précédente. De manière étonnante, l'accès à un L1 sur un cœur voisin est relativement coûteux. Nous n'avons pas d'explication pour ce résultat. De même, il est étonnant que l'accès au L2 d'un voisin soit légèrement plus coûteux que l'accès à son propre L2 (puisque le L2 est partagé). Cependant, nous attribuons ce résultat au fait que dans ce cas, il s'agit d'un autre cœur qui a alloué les données. Par conséquent, il est nécessaire de remplir le TLB. Enfin, il est intéressant de constater que la communication inter-cœurs est bien plus efficace entre deux cœurs d'une même puce (même s'ils ne partagent pas de cache) qu'entre deux cœurs de deux processeurs différents.

Nous avons également mesuré le débit de la mémoire lorsque plusieurs cœurs accèdent à cette dernière. Les tests ont été faits avec un programme simple utilisant des instructions *SSE* qui permettent de ne pas utiliser les caches. Les résultats sont présentés dans le tableau 1.2. Nous faisons varier le nombre de cœurs accédant à la mémoire et observons le débit total ainsi que le débit moyen par cœur.

À partir de ces résultats, nous pouvons faire plusieurs observations. Premièrement, un cœur est capable de saturer la bande passante mémoire disponible. Deuxièmement, lorsque plusieurs cœurs accèdent à la mémoire, la bande passante est partagée de manière équitable entre les cœurs. Troisièmement, plus le nombre de cœurs accédant à la mémoire augmente, plus le débit mémoire total maximal atteignable diminue. Ce résultat est dû à la contention sur le contrôleur mémoire.

Mémoire accédée	Localisation	Coût (cycles) / Écart type
Cache L1	Privé	3 / 0.1
Cache L2		15 / 0.5
Cache L1	Voisin	62 / 8.2
Cache L2		19 / 0.5
Cache L1	Sur la même puce	116 / 3.8
Cache L2		121 / 1.3
Cache L1	Sur une puce différente	213 / 1.8
Cache L2		212 / 1.5
Mémoire principale	-	263 / 3.1

TABLE 1.1 – Latence des accès mémoire en fonction de la localisation de la donnée sur une architecture avec deux processeurs Intel Xeon E5420 à quatre cœurs. Le cache L2 privé et le cache L2 voisin désignent le même cache L2.

Nombre de cœurs	Débit mémoire total (Mo/s)	Débit mémoire moyen par cœur (Mo/s) / Écart type
1	6759	6759 / -
2	6589	3295 / 0.4
4	6217	1554 / 0.3
8	6047	756 / 0.1

TABLE 1.2 – Débit de la mémoire centrale en fonction du nombre de cœurs sur une architecture avec deux processeurs Intel Xeon E5420 à quatre cœurs.

1.3.3 Limitations

La difficulté d'augmenter la fréquence des processeurs a entraîné une refonte de l'architecture des processeurs. Au lieu d'augmenter la fréquence, les fabricants se sont tournés vers plus de parallélisme. Par conséquent, chaque cœur est cadencé à une fréquence généralement moins importante que les anciens processeurs mono-cœur. Cette caractéristique permet notamment de réduire le dégagement thermique et la consommation des nouveaux processeurs. Cependant, la fréquence par cœur étant moindre, une application utilisant un seul et unique thread fonctionnera moins vite sur cette nouvelle architecture. Le premier problème est donc d'avoir des applications capables d'utiliser les processeurs multi-cœurs.

Les expériences présentées dans la section précédente nous ont permis de souligner un certain nombre de problèmes apportés par ces architectures multi-cœurs. Si le fait de dupliquer complètement les cœurs permet d'exécuter des threads en parallèle, les cœurs rentrent toujours en contention pour l'accès aux ressources partagées.

Le premier exemple est la contention sur le bus et le contrôleur mémoire. Nous avons pu voir dans les mesures précédentes que le bus est partagé entre les différents cœurs. Cela implique donc que si tous les cœurs accèdent à la mémoire simultanément, chaque cœur sera en attente de la donnée plus longtemps puisque les cœurs se partagent le bus. Nous avons pu également constater que le débit mémoire maximum atteignable diminue légèrement lorsque le nombre de cœurs accédant à la mémoire augmente. Il devient donc dans ce cas encore plus important d'utiliser au maximum les caches processeurs.

Cependant, l'utilisation des caches peut également entraîner d'autres problèmes. En effet, certains niveaux de caches sont partagés entre les cœurs. Outre la contention d'accès au cache, cela peut

provoquer, en fonction de l'empreinte mémoire des applications, une compétition pour obtenir le plus de lignes de cache possible. Cette compétition va entraîner une mauvaise utilisation des caches puisque chaque application va évincer les données de l'autre application. Des techniques logicielles de partitionnement des caches en fonction de la caractéristique des applications ont été étudiées et nous y reviendrons dans le chapitre 3.

Un autre problème vient du fait que le bus d'E/S est connecté sur le contrôleur mémoire. Ce choix architectural augmente finalement la contention sur le contrôleur mémoire ainsi que le trafic sur le bus connectant les cœurs au MCT. Ces problèmes ont notamment été expérimentés par les auteurs du projet RouteBricks [38] qui se sont rendu compte que leurs problèmes pour passer à l'échelle venaient de cette contention.

La cohérence de cache est un mécanisme très pratique pour développer des applications. Cependant, il a été montré que le mécanisme de cohérence de cache par écoute de bus est très compliqué à faire passer à l'échelle lorsque le nombre de cœurs augmente.

Enfin, le dernier problème est le problème de faux partage entre les cœurs. Le faux partage vient de la taille des lignes de caches. Une ligne de cache peut contenir plusieurs données disjointes (puisque sa taille est de 64 octets). Cette ligne est répliquée dans les caches de plusieurs cœurs. Prenons par exemple une ligne de cache contenant deux variables entières A et B stockées sur 64 bits. Le cœur 0 manipule la donnée A et le cœur 1 la donnée B. Lorsque le cœur 0 va changer la valeur de la donnée A, la ligne de cache va être invalidée sur le cœur 1 qui devra donc recharger cette donnée en mémoire. Ce fonctionnement est particulièrement inefficace et peut être très coûteux. Le problème est que, bien que les cœurs travaillent sur des données disjointes, la modification d'une valeur va entraîner l'invalidation d'une autre, déclenchant inutilement le mécanisme de cohérence de cache. Ce problème de faux partage, également présent sur les architectures multiprocesseurs, a été décrit par Torrellas *et al.* [93].

1.4 Processeurs multi-cœurs à accès mémoire non uniforme

Les limitations décrites dans la partie précédente ont entraîné les fabricants de processeurs à revoir le mode de fonctionnement des architectures multi-cœurs. Comme nous l'avons vu précédemment, un des défauts majeurs des architectures multi-cœurs vient du partage de la bande passante vers la mémoire ce qui entraîne une surcharge du contrôleur mémoire. De plus, le fait de connecter le bus d'E/S sur le bus mémoire est également un facteur aggravant de la surcharge du contrôleur mémoire.

Les nouvelles architectures multi-cœurs ont apporté une réponse à ce problème en exhumant une technique précédemment utilisée dans les machines massivement multiprocesseurs. Cette technique, appelée *NUMA* (*Non-Uniform Memory Access*), consiste à diviser la mémoire pour permettre des accès parallèles à des zones mémoire disjointes. L'accès mémoire est non uniforme, car le cœur accédera soit à sa mémoire locale, soit à de la mémoire distante (plus ou moins proche) et donc les coûts d'accès ne seront pas identiques en fonction de la zone mémoire accédée.

AMD a lancé en 2007 ses architectures NUMA avec l'architecture Opteron *Barcelona* suivie par la suite les architectures *Shangai* (2008) et *Istanbul* (2009). Intel produit également depuis 2008 des architectures NUMA avec son architecture *Nehalem* utilisée notamment dans les processeurs *core i7* et *core i5*.

Dans la suite de cette partie, nous détaillerons le fonctionnement d'une architecture comportant quatre processeurs AMD Opteron 8380 (architecture *Shangai*). Ces processeurs sont composés de quatre cœurs. Cette architecture sera utilisée dans la suite de ce document pour une partie des tests.

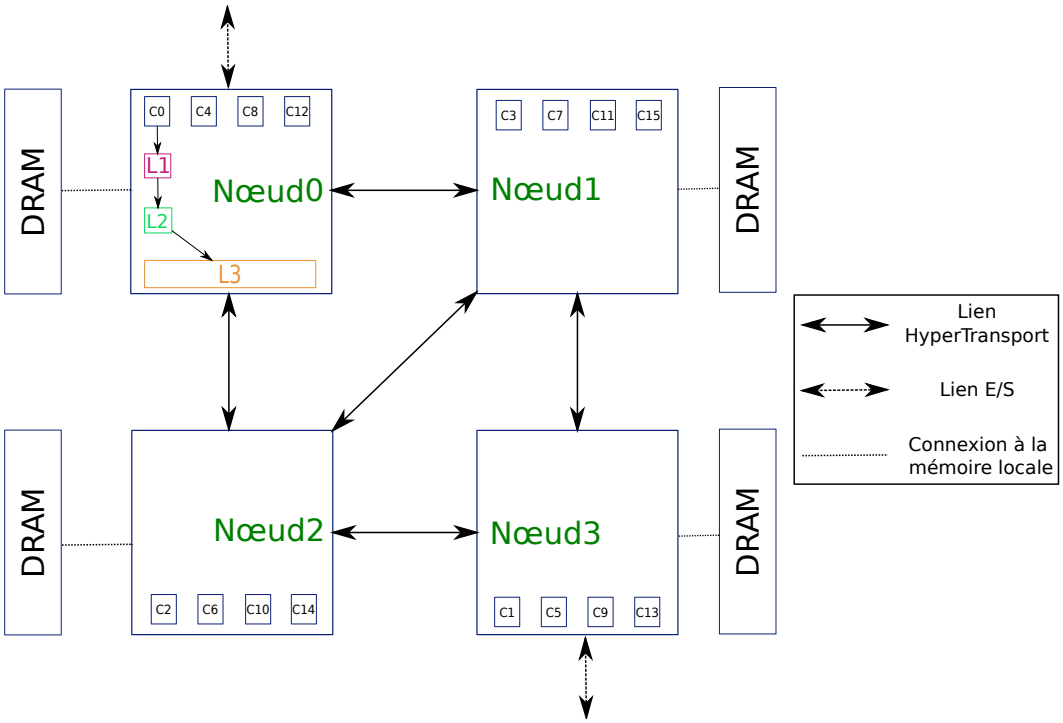


Figure 1.2 – Architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs.

1.4.1 Fonctionnement d'une architecture multi-cœurs à accès mémoire non uniforme

La figure 1.2 illustre l'architecture d'une machine AMD composée de quatre processeurs de quatre cœurs chacun. Chaque processeur est appelé un *nœud*. Les cœurs et les nœuds sont numérotés selon la convention employée par le système d'exploitation Linux 2.6.32. Pour des raisons de lisibilité, les caches L1 et L2 ne sont représentés que pour le cœur 0. Cependant, chaque cœur dispose de ses caches L1 et L2. Pour la même raison, le cache L3 n'est représenté que pour le nœud 0.

Dans le cas d'une architecture NUMA, le processeur se rapproche d'un système distribué. L'intérêt d'une telle architecture réside dans le fait que chaque nœud possède sa propre mémoire locale permettant ainsi aux applications de passer plus aisément à l'échelle.

Au sein d'un nœud, le principe de fonctionnement est identique aux processeurs multi-cœurs décrits précédemment. Chaque cœur possède ses caches L1 et L2. Dans cette architecture AMD, les caches L1 et L2 sont exclusifs (donc les données ne sont pas dupliquées). Il y a un cache L1 pour les données et un cache L1 pour les instructions. Ces caches ont une taille de 64 Ko divisés en lignes de 64 octets et sont associatifs à 2 entrées. Le cache L2 est lui unifié. Il a une taille de 512 Ko et est associatif à 16 entrées. Enfin, le cache L3 est partagé entre les quatre cœurs du processeur. Ce cache a une taille importante (6 Mo) et est associatif à 48 entrées. Le cache L3 est lui non inclusif. Cependant, AMD précise que le processeur essaie de déterminer si la donnée est partagée entre plusieurs cœurs. Si la donnée est partagée, le processeur essaiera de la laisser dans le L3 en plus du L2 ou du L1. Enfin, chaque nœud dispose également d'un contrôleur mémoire permettant l'accès à la mémoire locale.

Les nœuds sont interconnectés avec des liens *HyperTransport*. Ces liens à haut débit permettent de véhiculer des requêtes vers d'autres processeurs sous la forme de paquets. Cette interconnexion est proche de ce que l'on peut trouver dans les systèmes répartis, bien que les problématiques de la gestion du transport sur une puce ne sont pas les mêmes que celles sur un réseau local. L'utilisation des liens HyperTransport n'est pas limitée aux domaines des processeurs et peut être utilisée comme inter-

connexion haut débit entre plusieurs composants. Dans notre architecture, trois liens HyperTransport sont utilisés. Il est important de distinguer les liens HyperTransport gérant la communication avec les périphériques (nommés liens E/S sur la figure) et les liens HyperTransport connectant les processeurs entre eux. La topologie dans notre cas n'est pas complètement connectée, c'est-à-dire que les processeurs ne sont pas capables d'accéder à tous les nœuds en un saut. Dans notre cas, il n'y a pas de lien entre le nœud 0 et le nœud 3. En effet, sur ces deux nœuds un des liens est utilisé pour l'interconnexion avec les périphériques. L'utilisation des liens *HyperTransport* pour connecter les nœuds et la topologie potentiellement non complètement connectée introduisent des concepts de routage propres aux systèmes répartis. Le routage est effectué par chaque nœud. Notons que l'interconnexion aurait pu être construite différemment avec par exemple un bus d'E/S par nœud. Les processeurs Intel utilisent une structure d'interconnexion similaire nommée QPI (*QuickPath Interconnect*).

Les architectures NUMA telles que celles des nouveaux processeurs Intel et AMD proposent toujours le concept de mémoire partagée. Tout comme les architectures multi-cœurs classiques, ces nouvelles architectures utilisent le protocole de cohérence de caches pour garantir la cohérence des données même ci ces dernières sont copiées dans un cache distant. Ces architectures NUMA avec cohérence de cache sont appelées *ccNUMA (cache-coherent Non Uniform Memory Access)*. Cette cohérence de cache est implantée au-dessus du lien HyperTransport (*CHT*, Coherent HyperTransport).

Ce mécanisme de cohérence de cache est construit à base de messages. Une technique similaire à l'écoute des bus est mise en œuvre. Lorsqu'un cœur veut accéder à une zone mémoire et que cette zone n'est dans aucun des caches locaux (L1, L2 ou L3), il y a plusieurs possibilités. La première possibilité est que cette zone soit dans sa mémoire locale. Dans ce cas, il faut tout d'abord demander aux autres nœuds si cette donnée est stockée avec une valeur modifiée dans un de leurs caches. Après avoir reçu la réponse, le cœur accédera, si nécessaire, à sa mémoire locale. La seconde possibilité est que la donnée soit stockée dans une mémoire distante (gérée par exemple par le nœud X). Dans ce cas de figure, le cœur va faire une demande d'accès au nœud X qui gère la zone mémoire concernée (cette demande pourra éventuellement être routée par plusieurs nœuds). Le nœud X se chargera de faire la demande de valeur modifiée aux autres nœuds. Notons que ces autres nœuds répondront directement au cœur demandeur. Enfin, le nœud X va accéder à sa mémoire et envoyer la valeur au cœur demandeur. Enfin, le cœur demandeur enverra un message de signalisation au nœud X pour l'informer de l'état de la donnée. En écoutant ce qu'il se passe sur les liens, il sera donc possible de savoir si une donnée est partagée et donc, en cas de modification locale, d'invalider les autres copies.

1.4.2 Mesures de performances

Dans cette partie, nous allons nous intéresser à la latence d'accès aux différentes zones mémoire ainsi qu'au débit de la connexion mémoire locale et des bus HyperTransport. Pour cela, nous utilisons les mêmes outils que ceux décrits dans la section 1.3.2.

Le tableau 1.3 présente les mesures de latence effectuées sur l'architecture AMD précédemment décrite. Ce tableau présente les résultats lorsque l'on accède aux différentes zones de la mémoire. Ces zones peuvent être soit les zones de mémoire locales (cache L1, L2, L3 ou DRAM locale), soit les zones de mémoire distantes (cache L3 ou DRAM distante). Pour accéder à la DRAM distante, il faut soit un, soit deux sauts. Les écarts types observés sont généralement faibles (excepté pour les caches L1 et L3 locaux, mais l'ordre de grandeur est identique).

À partir de ces résultats, nous pouvons effectuer plusieurs observations. Premièrement, les caches locaux ont un coût d'accès très faible. Ces coûts sont comparables à ce qui a été observé sur les architectures multi-cœurs à accès mémoire uniforme. La seconde observation est que ces mesures confirment les schémas d'accès précédemment décrits. On voit notamment qu'un accès à la mémoire locale nécessite d'interroger les caches des nœuds distants. Cependant, de manière surprenante, le

	Latence d'accès (cycles) / Écart type	**Surcout du saut (cycles)**	**Surcout du saut (%)**
Cache L1 local	2 / 0.4	—	—
Cache L2 local	15 / 0.4	—	—
Cache L3 local	32 / 5.2	—	—
Mémoire locale	261 / 3.2	—	—
L3 distant de 1 saut	213 / 2.4	181	566 %
L3 distant de 2 sauts	306 / 3.1	274	856 %
Mémoire distante de 1 saut	286 / 2.3	26	10 %
Mémoire distante de 2 sauts	377 / 4.3	116	44 %

TABLE 1.3 – Latence des accès mémoire en fonction de la localisation de la donnée sur une architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs.

temps d'un accès à la mémoire locale est inférieur au temps de l'accès au L3 le plus lointain, alors qu'il est nécessaire d'interroger tous les caches distants pour vérifier si la donnée a été mise à jour. Nous n'avons pas d'explication pour ce comportement. Enfin, nous pouvons voir que, sans surprise, le coût d'un accès au L3 distant est très supérieur au coût d'un accès au L3 local. De même, nous pouvons évaluer le surcoût du bus HyperTransport à 10 % pour un accès nécessitant 1 saut et 40 % pour un accès nécessitant 2 sauts.

Nous avons également mesuré la bande passante de la mémoire locale et des mémoires distantes (1 saut et 2 sauts). Nous effectuons les tests en utilisant 1, 2 et 4 cœurs du même processeur. Les résultats obtenus sont présentés dans le tableau 1.4.

Mémoire accédée	**1 cœur, débit mémoire total (Mo/s)**	**2 cœurs, débit mémoire total (Mo/s)**	**4 cœurs, débit mémoire total (Mo/s)**
Mémoire locale	3318	6379	7574
Mémoire distante de 1 saut	2839	2849	2848
Mémoire distante de 2 sauts	2022	2732	2737

TABLE 1.4 – Débit des différentes mémoires en fonction du nombre de cœurs sur une architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs.

À partir de ces résultats, nous pouvons faire plusieurs observations. Premièrement, contrairement à l'architecture Intel précédemment décrite, un cœur ne suffit pas à saturer la bande passante mémoire disponible. Deuxièmement, le bus HyperTransport a un débit moins important que le bus mémoire local. Dans ce cas, en fonction du nombre de sauts, il faut un ou deux cœurs seulement pour saturer le bus HyperTransport.

Enfin, nous avons mesuré le débit mémoire atteint lorsque tous les cœurs accèdent à leur mémoire locale. La figure 1.3 présente le débit obtenu lorsqu'un, deux, trois puis quatre nœuds accèdent à leur mémoire locale. Lorsqu'un nœud accède à sa mémoire locale, cela signifie que tous les cœurs de ce nœud accèdent à la mémoire.

Nous pouvons observer sur cette figure que le débit total mesuré dévie de l'idéal au-delà de deux

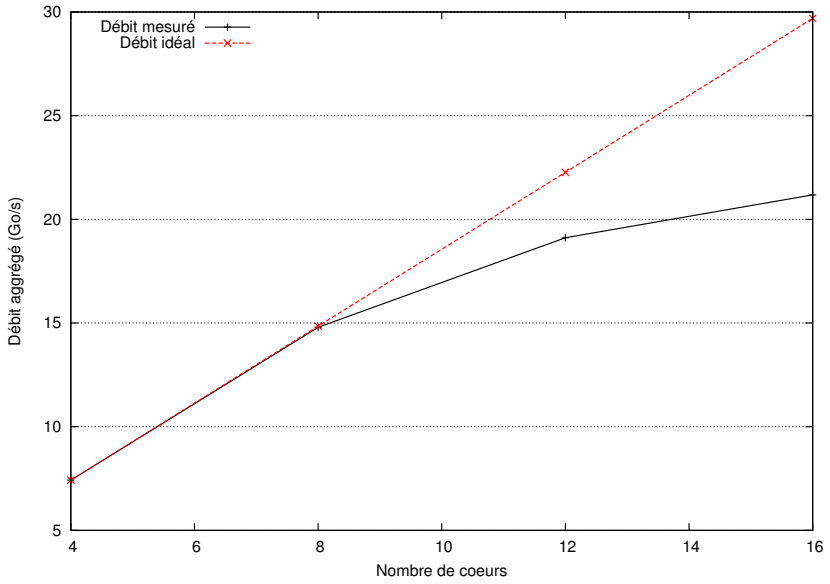


Figure 1.3 – Évolution du débit mémoire en fonction du nombre de nœuds accédant à leur mémoire locale sur une architecture ccNUMA avec 4 processeurs AMD Opteron 8380 à quatre cœurs.

nœuds accédant à leur mémoire locale. Nous attribuons ce résultat au protocole de cohérence de cache qui envoie des messages sur les liens HyperTransport pour avoir l'état des caches distants. Par conséquent, ces bus saturent et ralentissent le débit global de la mémoire.

1.4.3 Limitations

Les mesures de latence et de débit effectuées dans la partie précédente permettent de lever une série de problèmes. Le premier problème soulevé est qu'accéder à la mémoire distante est jusqu'à 40 % plus coûteux qu'accéder à sa propre mémoire. Un processeur en attente va donc attendre la donnée plus longtemps et va avoir encore plus de difficultés à recouvrir ces temps d'accès mémoire.

Ce problème de surcoût des accès distants est d'autant plus grand que le débit des bus HyperTransport est bien plus limité que celui de la mémoire. Nous avons pu voir que le débit de ces bus est environ 60 % plus faible que le débit du bus mémoire local. De plus, les accès à la mémoire locale contribuent également à la saturation des bus HyperTransport. Il devient donc très important :

- *De limiter au maximum les accès mémoire à une zone mémoire gérée par un autre processeur.*
- *De limiter les accès à la mémoire locale.* En plus de la charge accrue sur le bus HyperTransport, le coût d'un accès à la mémoire locale est 8 fois plus important que le coût d'accès au cache L3.

Cette architecture étant dérivée des architectures multi-cœurs à accès mémoire uniforme, elle hérite des problèmes de ces dernières telles que les problèmes de faux partage, les problèmes de compétition pour l'accès au cache partagé ou encore les problèmes de parallélisation des applications.

1.5 Processeurs du futur

Dans les sections précédentes, nous avons étudié les différentes architectures multi-cœurs actuellement adoptées par les fabricants. Ces architectures ont cependant des limitations les empêchant de

passer à l'échelle, c'est-à-dire être capable d'intégrer plusieurs centaines voire plusieurs milliers de cœurs. Différentes approches ont été étudiées pour outrepasser ces limitations. Certaines de ces architectures ne sont qu'à l'état d'idées ou de prototypes, d'autres sont actuellement disponibles. Parmi ces directions, potentiellement complémentaires, nous pouvons citer notamment :

- *La suppression du mécanisme de cohérence de cache.* Comme nous l'avons vu dans les parties précédentes, ce mécanisme est extrêmement coûteux et empêche une architecture NUMA de passer parfaitement à l'échelle. Cependant, la suppression du mécanisme de cohérence de cache implique une refonte des applications, car ces dernières se basent essentiellement dessus pour la communication inter-cœurs.

- *L'intégration d'un mécanisme de passage de message entre les cœurs.* Ce mécanisme de passage de message permettra de faire de la communication asynchrone efficace entre les cœurs. Une application peut notamment s'appuyer sur cette fonctionnalité pour implémenter un mécanisme de cohérence de cache logicielle.

- *L'intégration de cœurs hétérogènes.* Ces cœurs peuvent être hétérogènes en terme de fonctionnalités proposées. Par exemple, il est possible d'intégrer au sein d'une même puce des processeurs généralistes et des processeurs spécialisés (notamment pour les traitements liés aux domaines de l'audio, de la vidéo ou des communications réseau). Cela permet de faire cohabiter des processeurs complexes avec des processeurs plus simples. Il est également possible d'intégrer des processeurs identiques tournants à des fréquences différentes pour des raisons d'économies d'énergie.

Dans cette section, nous allons brièvement voir quels sont les prototypes et les processeurs existants qui pourraient être les prémisses des architectures futures.

1.5.1 Le projet Tera-scale

Intel est en train de mener des recherches, au travers de son projet Tera-scale [32], sur le futur des processeurs multi-cœurs et sur leur utilisation. Ces recherches portent à la fois sur des considérations matérielles et des considérations logicielles. L'objectif de ce projet est d'essayer de produire des processeurs avec un nombre de cœurs compris entre 10 et 100 cœurs par puce. Sur les parties matérielles, ces études ont abouti à la conception de deux prototypes. Le premier est le processeur Teraflops composé de 80 cœurs. Le second est le processeur *Single-chip Cloud Computer* (SCC) composé de 48 cœurs. Nous allons détailler par la suite les spécificités de ces deux processeurs.

Prototype Intel Teraflops à 80 cœurs. Azimi *et al.* ont présenté le projet Tera-scale dans [16]. Dans le projet Tera-scale, le processeur est vu comme un ensemble de cœurs simples connectés en utilisant une grille 2D. Chaque cœur dispose de 5 connexions (4 vers les cœurs les plus proches dans la grille et une vers la mémoire). Chaque cœur comporte un routeur qui permet la communication entre tous les cœurs du processeur.

Il est important de considérer que le processeur embarque également le contrôleur mémoire ainsi que la mémoire, ce qui permet une architecture originale d'empilement des couches processeurs et mémoire. Cette nouvelle architecture matérielle permet d'améliorer sensiblement les débits mémoire.

Le processeur, bien que l'architecture soit homogène initialement, peut également embarquer des processeurs spécialisés tels que des unités de calcul habituellement utilisées par les cartes graphiques. Le processeur peut avoir différentes hiérarchies de cache en fonction du type des cœurs utilisés. Cette architecture ne propose pas de passage de message efficace entre les cœurs, mais propose un

mécanisme de cohérence de cache basé non plus sur de l'écoute de bus mais sur un mécanisme de chemins permettant de garder un suivi des copies. Ce mécanisme, déjà utilisé sur des machines massivement multiprocesseurs, permet de conserver la compatibilité avec les applications existantes.

Enfin, la multiplication du nombre de cœurs sur une puce rend cette dernière plus sensible aux pannes. Les cœurs doivent donc pouvoir gérer les pannes. Un cœur en panne redistribuera ses travaux en attente aux autres cœurs. De même, le mécanisme de routage doit pouvoir être tolérant à la panne d'un cœur.

En février 2007, Intel a présenté son premier prototype correspondant à l'architecture Tera-scale, appelé Teraflops. Intel réussit avec ce processeur l'intégration de 80 cœurs sur une puce. Les 80 cœurs sont relativement simples, ce qui permet d'en intégrer autant sur une unique puce. Les cœurs sont uniquement capables de faire des calculs en virgule flottante. Ces cœurs sont interconnectés par une grille telle que décrite précédemment. Les cœurs sont répartis par blocs. Chaque bloc contient 8 cœurs. Chaque cœur dispose d'un cache L1 privé. Chaque bloc dispose d'un cache L2. Enfin, un cache L3 commun à tous les blocs est également présent. La mémoire est répartie sur tous les cœurs. Chaque cœur gère une partie (64 Mo) de la mémoire RAM, ce qui porte l'espace mémoire disponible à un peu plus de 5 Go. Enfin, la gestion des fautes est également présente.

Ce prototype a réussi à atteindre la performance d'un teraflops (nombre d'opérations flottantes exécutées par seconde) en ne consommant que 62W. Cependant, ce prototype n'a jamais été, à notre connaissance, disponible pour des expériences autres que celles menées par les équipes d'Intel.

Prototype Intel SCC à 48 cœurs. En décembre 2009, Intel a dévoilé un second prototype, appelé *Single-chip Cloud Computer*. Ce processeur embarque 48 cœurs. Les cœurs sont divisés en 4 groupes de 6 *tuiles* et sont interconnectés, comme précédemment, en utilisant une grille 2D. Une *tuile* est composé de deux cœurs qui disposent chacun de leur cache L2. Contrairement à l'architecture précédente, il n'y a pas un routeur par cœur, mais un routeur par *tuile*. Ce processeur marque également plusieurs différences par rapport au prototype précédent :

- *L'utilisation de cœurs complets.* Contrairement au prototype précédent, chaque cœur est basé sur l'architecture *Pentium* d'Intel.
- *L'utilisation de contrôleurs mémoires en lieu et place de l'empilement de la mémoire au dessus des cœurs.* Dans ce cas, il y a un contrôleur mémoire par groupe de *tuiles* (donc 4 contrôleurs mémoires au total).
- *L'abandon de la cohérence de cache matérielle.* À la place de la cohérence de cache matérielle, le processeur fournit la possibilité de faire passer efficacement des messages d'un cœur à un autre. Ce passage de message est très efficace au sein de la puce grâce à l'utilisation de tampons de messages. Avec ce mécanisme de passage de message, l'application pourra gérer au besoin de la cohérence de cache logicielle.
- *L'hétérogénéité de fréquence.* Pour des raisons de consommation mémoire, chaque *tuile* peut tourner à une fréquence différente.

Ce prototype devrait pouvoir être accessible aux collaborateurs d'Intel.

1.5.2 Processeurs hétérogènes

Une autre possibilité pour les processeurs du futur est d'évoluer vers des architectures hétérogènes. Un processeur peut contenir des cœurs hétérogènes du fait de leurs fonctionnalités ou de leur fréquence. L'idée est d'avoir des cœurs hétérogènes en fréquence et d'améliorer la consommation

énergétique globale du processeur. Il va ainsi être possible d'affecter aux cœurs puissants des tâches capables de les exploiter et aux cœurs peu puissants des tâches fréquemment en attente (par exemple d'accès mémoire ou d'E/S). De même, intégrer des cœurs spécialisés dans différentes opérations au sein d'une seule puce permet d'améliorer notamment la latence de communication entre les cœurs ainsi que la consommation énergétique (puisque les cœurs spécialisés consommeront moins pour une même opération).

Dans les parties suivantes, nous allons parler de l'architecture Cell ainsi que de la convergence des processeurs avec le domaine des processeurs graphiques (*GPU*).

1.5.2.1 Architecture Cell

Le Cell est un processeur développé conjointement par Sony, Toshiba et IBM. Ce processeur est bien connu puisque c'est celui qui équipe la PlayStation 3, la console de jeux de Sony. Plus généralement, ce processeur est utilisé notamment pour faire du calcul scientifique.

Le Cell est un processeur hétérogène, car il se base sur un cœur généraliste (appelé *PPE, Power Processor Element*) et huit cœurs spécialisés (appelés *SPE, Synergistic Processing Elements*). Les concepteurs ont choisi de simplifier leur processeur et de laisser le travail d'optimisation au programmeur. Ainsi, le PPE est un PowerPC simplifié (sans, par exemple, la gestion de l'exécution des instructions dans le désordre). Le PPE est capable d'exécuter deux threads en parallèle. Les SPE, quant à eux, sont des processeurs spécialisés avec un jeu d'instructions SIMD (une instruction est capable de traiter plusieurs données). Les SPE sont reliés entre eux, mais aussi au PPE, à la mémoire et aux E/S en utilisant un bus en forme d'anneau. Les SPE disposent d'une petite quantité de mémoire embarquée, mais ne peuvent pas accéder à la mémoire. Les SPE vont alors utiliser un contrôleur DMA (*Direct memory access*) pour faire des demandes asynchrones à la mémoire. Il est donc possible d'entrelacer les demandes avec le calcul. Le rôle du PPE est de traiter les calculs simples et de répartir le travail sur les SPE. Le PPE va pouvoir communiquer avec les SPE par messages.

Ce processeur permet d'atteindre une puissance de calcul importante. Cependant, ces choix architecturaux entraînent des difficultés de programmation. Par exemple, l'accès à la mémoire asynchrone oblige le programmeur à penser à l'entrelacement de ses calculs avec les accès mémoire pour que le processeur ait toujours du travail. Ces difficultés ont réservé ce processeur à des opérations spécifiques telles que le calcul scientifique, les jeux vidéo ou le décodage vidéo.

1.5.2.2 Synergie avec les processeurs graphiques

Les processeurs graphiques sont devenus des processeurs multi-cœurs spécialisés dans certains calculs. Par exemple, certains processeurs graphiques (*GPU, Graphics Processing Unit*) peuvent être utilisés comme accélérateurs pour décoder matériellement certains formats vidéo (notamment les formats vidéos haute définition). Nvidia a également intégré l'architecture CUDA (*Compute Unified Device Architecture*) à ses cartes graphiques depuis la série GeForce8. Cette architecture permet de programmer la carte graphique pour lui faire exécuter des opérations générales de calcul (nomée GPGPU, *General-Purpose computing on Graphics Processing Units*). Cependant, l'utilisation du GPU est encore limitée à certaines applications (notamment les applications scientifiques). OpenCL est une bibliothèque tierce permettant également de distribuer du calcul sur ces dernières. Cela montre bien l'importance prise par les GPGPU.

Intel et AMD ont aussi pour projet d'intégrer un GPU au sein de leurs CPU. L'objectif de cette intégration est de réduire les latences de communication entre le CPU et la mémoire et éventuellement de décharger certains calculs du CPU vers le GPU. Intel a concrétisé cet objectif au sein de certains de ces processeurs *i5* et *i7*. AMD a annoncé le projet *Fusion* qui devrait aboutir en 2011.

Enfin, Intel est allé un cran plus loin dans l'intégration des GPU et des CPU avec le projet Larrabee [83]. L'idée de ce projet n'est plus d'intégrer un CPU et un GPU mais de les fusionner. Le processeur est un ensemble de cœurs reliés entre eux, à la mémoire et aux E/S par un anneau. Chaque cœur dispose de son cache et un mécanisme de cohérence de cache est présent. Chaque cœur est composé d'une unité de calcul *scalaire*, qui est l'équivalent d'un processeur mono-cœur classique, et d'une unité de calcul *vectoriel* (VPU). Un VPU est proche des unités de traitement que l'on peut trouver dans un GPU ainsi que des instructions SSE des processeurs. Larrabee supporte un langage de programmation x86 avec des extensions spécifiques. Le support du x86 est un des gros avantages de ce projet. En effet, cela rend le processeur compatible avec les applications existantes (après recompilation avec le compilateur pour l'architecture Larrabee). De plus, le compilateur permet d'utiliser les unités vectorielles automatiquement. Cependant, les utiliser pleinement nécessite de produire du code spécifique. Bien qu'Intel ait démontré des résultats convaincants [83], l'avenir et le positionnement du projet sont flous.

1.6 Bilan

Dans ce chapitre, nous avons étudié l'évolution des différentes architectures de processeurs. Nous avons vu notamment les limitations des architectures mono-cœur et l'introduction progressive du parallélisme. Le premier parallélisme introduit l'a été au niveau des instructions. L'exécution en parallèle d'instructions a permis un gain important de performance au sein d'une application. Toutefois, il est difficile de faire évoluer les mécanismes de pipeline. Le parallélisme a donc évolué vers du parallélisme par threads. Plusieurs approches ont été étudiées : dupliquer une partie du processeur (SMT), dupliquer tout le processeur (processeurs multi-cœurs à accès mémoire homogène) ou encore dupliquer le processeur ainsi que la mémoire (processeurs multi-cœurs NUMA). Nous avons également vu que les grandes tendances pour les architectures du futur étaient la suppression du mécanisme de cohérence de cache, l'introduction de mécanisme de passage de message et l'hétérogénéité des processeurs.

Cette étude montre également qu'il est nécessaire de repenser les applications pour ces nouvelles architectures parallèles. Cela implique de repenser :

- *Le modèle de programmation des applications.* En effet traditionnellement, les applications ont été développées en utilisant un seul et unique flot d'exécution. Il est donc difficile (voire impossible) d'utiliser efficacement les architectures parallèles avec ce modèle de programmation. La probable suppression du mécanisme de cohérence de cache est également un défi pour les modèles de programmation.

- *La manière d'exécuter les applications.* Cela suppose notamment de repenser les systèmes d'exploitation existants pour qu'ils prennent en compte les nouvelles architectures ainsi que les environnements d'exécution. La répartition efficace des applications et de leurs threads est le problème majeur de ces deux composants. Dans un objectif d'efficacité, la localisation mémoire des données ainsi que les affinités entre threads sont notamment des paramètres à prendre en compte.

Chapitre 2

Modèles de programmation pour les architectures multi-cœurs

Sommaire

2.1 Modèles de programmation classiques 24
2.1.1 Programmation à base de threads 24
2.1.2 Améliorations du modèle de threads 27
2.1.3 Programmation événementielle 29
2.1.4 Amélioration du modèle d'événements 31
2.1.5 Approches visant à unifier les deux modèles 34
2.2 Nouveaux modèles de programmation multi-cœurs 35
2.2.1 Modèles de programmation *fork-join* 35
2.2.2 Programmation par étages 36
2.2.3 Files d'éléments 37
2.2.4 MapReduce 37
2.2.5 Grand central 38
2.3 Bilan 38

Nous avons pu voir dans le chapitre précédent que les processeurs ont évolué jusqu'à proposer un degré de parallélisme important. Ce parallélisme doit cependant être exploité au sein d'une application pour que cette dernière puisse bénéficier des améliorations futures. Cette utilisation nécessite de repenser les modèles de programmation pour utiliser au mieux les architectures multi-cœurs. Cette refonte des modèles de programmation doit être pensée selon trois axes principaux :

1. *La simplicité.* En effet, il doit pouvoir être aisé pour le programmeur de structurer son application en tâches parallèles. De plus, cette structuration doit s'effectuer de manière indépendante à l'architecture cible. Cela permet notamment de déployer l'application sur différentes architectures proposant différentes formes ou différents degrés de parallélisme.

2. *La correction.* Ce critère, intimement lié au précédent, juge si le modèle de programmation permet ou ne permet pas les erreurs inhérentes à la programmation concurrente.

3. *L'efficacité.* Le modèle de programmation et ses hypothèses impactent directement la performance de l'application sur les architectures parallèles. Par exemple, un modèle de programmation impliquant une communication fréquente entre les processeurs réduira d'autant ses performances.

Dans la suite de ce chapitre, nous allons proposer une vue d'ensemble des modèles de programmation pour les architectures multi-cœurs. Nous allons tout d'abord étudier les modèles de programmation utilisés traditionnellement dans les applications et verrons les tentatives d'amélioration de ces modèles en suivant les trois critères précédemment décrits. Nous nous intéresserons ensuite aux modèles de programmation conçus pour les architectures multi-cœurs qu'ils soient généralistes ou spécialisés pour un type d'application.

2.1 Modèles de programmation classiques

Nous avons vu dans le chapitre précédent que, vue du processeur, une application est un flot d'instructions. Cependant, du point de vue du système d'exploitation, l'application est contenue dans un *processus*. En plus de ce flot d'instruction, la notion de processus contient :

- *Une pile.* Cette pile est utilisée pour effectuer les appels de méthodes. En effet, la pile va contenir au moins (i) les paramètres d'appel, (ii) les valeurs des données locales à la fonction, (iii) le pointeur vers la fonction appelante, et (iiii) l'emplacement pour stocker une valeur de retour. La pile peut également contenir les valeurs des variables statiques.
- *Un tas.* C'est dans ce tas que seront effectuées les allocations dynamiques de mémoire demandées par le programme.
- *Une table des pages.* Pour gérer les problèmes de sécurité vis-à-vis des autres processus, nous avons vu précédemment que chaque processus disposait d'un espace de mémoire virtuelle. La table des pages est gérée par le système d'exploitation.
- *La liste des ressources utilisées.* Les ressources utilisées sont, par exemple, les descripteurs de fichiers ouverts. Ces fichiers peuvent être soit des fichiers disques traditionnels, soit des fichiers spéciaux tels des sockets ou des périphériques. Cette table des ressources utilisée est gérée, en général, par le système d'exploitation.

Les processus sont ordonnancés par le système d'exploitation sur les cœurs disponibles. L'ordonnancement des processus est fréquemment effectué par partage de temps, c'est-à-dire que les processus ont un quantum de temps qui leur est alloué et le système d'exploitation change le processus en cours lorsque ce temps est expiré. Des précisions sur l'ordonnancement des processus sont données dans le chapitre 3.

Au-dessus de ce modèle de processus, l'application choisit un modèle de programmation. Le choix du modèle de programmation impacte la structuration de cette dernière ainsi que les hypothèses formulées par le programmeur lors de la conception. Deux modèles de programmation sont principalement utilisés pour le développement d'applications. Le premier modèle est structuré autour d'un ou plusieurs processus légers (appelés *threads*). Le second va lui être construit autour d'événements et du traitement de ces événements. Les qualités respectives d'un modèle par rapport à un autre ont été longuement débattues et sont toujours un sujet de controverse [64, 74, 96, 89, 50].

Nous allons dans la suite de cette section détailler les spécificités ainsi que les points forts et les points faibles des modèles à base de threads et des modèles à base d'événements.

2.1.1 Programmation à base de threads

La première façon pour un programmeur est de structurer son application en utilisant des processus légers (appelés *threads*). Un thread est appelé processus léger car il a des caractéristiques proches d'un processus. L'application est ainsi découpée en flots d'exécution. Chaque thread dispose de sa

propre pile, mais les threads partagent les autres informations telles que le tas, la table des pages (et donc l'espace d'adressage) et les descripteurs de fichiers ouverts.

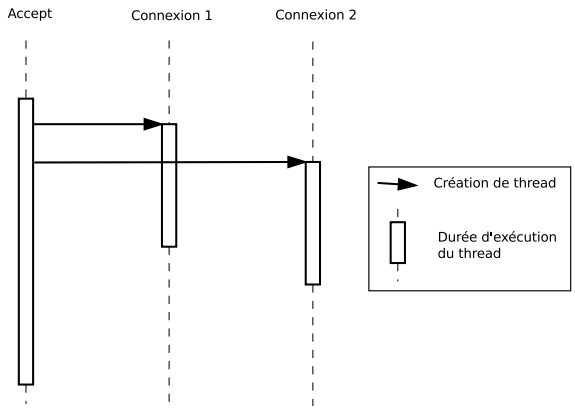
```
Thread d'acceptation de connexion :

  while (true) {
    Socket nouvelle_connexion = accept();
    créer_threads(nouvelle_connexion);
  }

Thread de traitement de la connexion :

  // La socket connexion à
  // passer en paramètre du thread

  while (! est_fermée(connexion)) {
    Message m = lire_message(m);
    // Traiter message m
  }
```



. (a) (b)

Figure 2.1 – (a) Pseudo-code d'un serveur de données utilisant le modèle de threads (b) diagramme de séquences du même serveur pour deux connexions reçues.

La figure 2.1 donne le pseudo-code ainsi qu'un exemple de séquence d'un serveur de données construit en utilisant des threads. Dans cet exemple, l'application gère deux connexions clientes. Pour ce serveur de données, il existe deux types de threads : le thread chargé de la gestion des connexions et les threads chargés de la gestion du dialogue sur les connexions acceptées. Lorsqu'une connexion a été acceptée, un nouveau thread est créé. L'identifiant de la connexion, c'est-à-dire le descripteur de fichier associé à la socket, est affecté au thread nouvellement crée pour qu'il s'occupe du dialogue avec le client. Lorsque la communication avec le client se termine, le thread est détruit. Ce mécanisme de délégation du travail à un thread distinct est très souvent utilisé dans les serveurs à base de threads. Apache [6] est un exemple de serveur Web largement déployé construit sur ce modèle [1].

Il est intéressant de noter que l'application aurait été développée de façon similaire en utilisant des processus en lieu et place des threads. Cependant, dans ce cas, le partage d'information (socket cliente, statistiques éventuelles du serveur...) entre le processus père et son processus fils aurait été plus complexe et plus coûteux vu qu'ils ne partagent ni espace d'adressage ni table des descripteurs de fichiers.

Les avantages de l'utilisation des threads sur les processus sont donc importants en terme de simplicité, d'utilisation et de coût. Les gains viennent du partage de l'espace d'adressage entre deux threads. Le partage de mémoire permet d'éviter des communications fastidieuses et coûteuses entre les processus (pas d'utilisation de *pipes* ou de sockets). De plus, le partage de l'espace d'adressage évite les vidages du TLB lors de changement de contexte avec un autre thread du même processus. Pour ces raisons, les threads sont considérés comme des processus « légers ».

Tout comme les processus, les threads peuvent être distribués sur les cœurs disponibles. Cette structuration de l'application permet donc l'utilisation des architectures multi-cœurs (à condition d'avoir au moins un thread par cœur). Les threads peuvent également être multiplexés sur un cœur, par exemple pour maximiser l'utilisation du cœur en présence de requêtes d'E/S bloquantes.

L'interface de programmation la plus utilisée pour programmer les applications à base de threads est la norme *POSIX pthread*, implantée par exemple par la bibliothèque *NPTL (Native POSIX Thread*

1. Plus précisément, la version *worker* du serveur Web Apache utilise un mélange de processus et de threads. Nous y reviendrons en détail dans le chapitre 6.

Library).

2.1.1.1 Problèmes principaux

L'utilisation de la mémoire partagée entre les threads engendre un nouveau problème : l'accès concurrent à des données partagées. En effet, si deux threads accèdent à la même zone mémoire simultanément, la valeur résultante pourra être incorrecte (malgré la cohérence des caches assurée par le matériel). Ce fonctionnement provient de deux raisons. La première est que certaines opérations sont transformées non pas en une instruction, mais en une suite d'instructions. Par exemple, l'affectation `i = i + 1` va générer trois instructions : une pour charger la donnée dans le registre, une pour effectuer l'addition et la dernière pour sauvegarder le résultat en mémoire. La deuxième est que les registres ne sont pas concernés par le protocole de cohérence de cache. Si l'instruction copie la donnée en registre avant de la traiter, comme dans le cas précédemment cité, elle va potentiellement travailler sur des données invalides, car mises à jour sur un autre cœur. Le même problème existe, à plus gros grain, si l'application fait une copie de la donnée dans une variable temporaire avant de la traiter.

La solution à ce problème de concurrence est d'utiliser des verrous. Les primitives de verrouillage permettent de garantir que seul un thread pourra accéder à une zone mémoire à un instant donné. Cette nécessité de prendre un verrou avant de mettre à jour des données partagées est probablement le point le plus problématique lors de l'utilisation de multiples threads. En effet, la prise de verrous est complexe. Il est, par exemple, possible que l'application se trouve dans une situation d'interblocage. Si le thread 1 possède le verrou A et cherche à acquérir le verrou B et que le thread 2 possède le verrou B et cherche à acquérir le verrou A, il n'est pas possible que les threads progressent. Une solution à ce problème consiste à acquérir les verrous toujours dans le même ordre. Cette solution est difficile à maître en œuvre en pratique. La composition avec des bibliothèques tierces peut également être problématique, car l'application ne sait pas quels verrous sont détenus par les différentes fonctions de la bibliothèque.

Lee [64] montre qu'il est très difficile, voire impossible, de maîtriser complètement l'ordonnancement des threads. Il prend notamment l'exemple du projet Ptolemy où un problème d'interblocage a été découvert quatre ans plus tard, malgré un travail de conception et de vérification rigoureux. Les processeurs multi-cœurs augmentent encore les possibilités d'ordonnancement et de nouveaux problèmes de concurrence apparaissent. En effet, si les threads sont exécutés sur un seul cœur, les threads sont en concurrence à cause du temps processeur partagé. A contrario, avec les processeurs multi-cœurs, il y a un réel parallélisme des threads et donc des accès aux zones partagées.

L'utilisation de verrous est très néfaste pour le passage à l'échelle d'une application. Lorsque plusieurs threads sont bloqués sur un verrou, aucun de ces threads (et donc des cœurs associés) ne peut faire de progrès. Pour résoudre ce problème, la solution est d'utiliser des verrous avec un grain le plus fin possible (pour la section critique et les données impactées). Cependant, plus les verrous ont un grain fin, plus les risques de tomber dans des cas d'interblocages augmentent.

La consommation mémoire importante est un problème généralement reproché à la programmation par threads. Comme nous l'avons vu précédemment, chaque thread dispose de sa pile pour pouvoir gérer ses appels de fonction. Cette pile a une taille initiale importante (8 Mo sur nos machines de tests). La consommation mémoire virtuelle de X threads est donc de X * 8 Mo. Cette consommation mémoire peut rendre ce modèle impropre pour certaines architectures (notamment les architectures embarquées). Notons cependant que la mémoire réellement utilisée par le thread peut être moins importante.

La gestion des threads (création, destruction, ordonnancement) est également très coûteuse. En effet, à chaque changement de contexte, il faut sauvegarder l'état du processeur pour pouvoir le res-

taurer par la suite. Sauvegarder l'état du processeur consiste essentiellement à sauvegarder l'état des registres. Il faut également restaurer le contexte du nouveau thread. En plus de son coût inhérent, ce mécanisme réduit l'efficacité des caches puisque le thread nouvellement ordonnancé va potentiellement remplacer les données du thread précédent en cache. Lorsque le thread précédent sera réordonnancé, ses données ne seront peut-être plus en cache.

La préemption automatique est un problème pour le programmeur. En effet, il peut être difficile de reproduire les problèmes de concurrence. L'ordonnanceur de threads est le seul responsable à la fois du quantum de temps affectés aux threads, de leur placement sur les différents cœurs ainsi que de leur exécution simultanée. Cela provoque un non-déterminisme de l'exécution qui rend la recherche d'erreurs plus compliquée.

Enfin, plus globalement, la gestion de la mémoire partagée se base sur les mécanismes de cohérence de cache. Ces mécanismes sont peu efficaces et ont un passage à l'échelle restreint. De plus, nous avons vu que ces mécanismes tendaient à disparaître dans les futures architectures de processeurs.

2.1.2 Améliorations du modèle de threads

Différentes approches ont essayé d'améliorer la programmation à base de threads. Ces approches ont deux objectifs principaux. Le premier est de simplifier la programmation parallèle en proposant des facilités syntaxiques pour gérer les sections critiques ou la parallélisation du code. Le second objectif est d'améliorer les performances.

L'ordonnancement coopératif. Une partie des problèmes liés aux threads provient de leur préemption automatique. Dans ce cas, les threads peuvent se faire retirer du processeur au milieu d'une section critique, obligeant le programmeur à gérer le verrouillage des données manipulées. La solution proposée par Adya *et al.* [10] consiste à utiliser un type d'ordonnancement dans lequel l'application redonne explicitement le processeur à l'ordonnanceur. Cette méthode, appelée *ordonnancement coopératif*, est à mi-chemin entre l'exécution *run-to-completion*, c'est-à-dire sans interruption, et l'exécution pouvant être préemptée à n'importe quel moment. Avec une méthode d'ordonnancement coopératif, le programmeur maîtrise les moments où le système peut suspendre un flot d'exécution pour attribuer le cœur à un autre flot et les moments où le programme ne doit pas être interrompu. Par exemple, une application ne redonnera pas la main à l'ordonnanceur au milieu d'une section critique. Au contraire, il est possible de demander un changement de contexte juste après l'envoi d'une demande d'E/S bloquante.

Cette méthode est très intéressante sur une machine mono-processeur car elle permet de résoudre les problèmes d'accès concurrents à des ressources partagées. Cependant, elle ne résout pas du tout le problème sur les machines multi-cœurs, sur lesquelles les threads sont en concurrence réelle pour l'accès à ces ressources. De plus, cette méthode n'est pas applicable si la section critique englobe des E/S bloquantes ou alors risque de dégrader sensiblement les performances.

Les mémoires transactionnelles. Les mémoires transactionnelles [52, 62] s'attaquent au problème des accès concurrents. L'idée est d'utiliser des transactions pour la gestion de la mémoire. Les transactions sont une technique empruntée aux systèmes de gestion de base de données. L'idée est de décharger le programmeur de la gestion des données partagées sur le système de gestion de transaction. Les transactions peuvent concerner un ou plusieurs accès à la mémoire ou la quasi-intégralité du code de l'application. La solution fréquemment choisie dans les mémoires transactionnelles est de déclarer des blocs de code *atomiques*. Lorsque l'application commence un bloc atomique, une nouvelle transaction est démarrée. Les opérations sont effectuées uniquement localement et ne sont pas

visibles des autres threads. À la fin du bloc atomique, la transaction est validée (*commit*) si possible, c'est-à-dire s'il n'y a pas eu de changement sur la donnée. Valider une transaction consiste à répercuter les modifications locales dans la mémoire globale de manière à les rendre visible depuis un autre thread. S'il n'est pas possible de valider une transaction, alors les modifications locales sont annulées et les valeurs remises à l'état initial (*rollback*).

Le gros avantage d'utiliser les mémoires transactionnelles pour la gestion de la concurrence est la simplicité d'expression, la correction ainsi que l'absence de possibilité d'interblocage. En effet, les algorithmes de mémoire transactionnelle n'utilisent pas de verrous et permettent donc une composition facile des transactions pour le programmeur. Le support d'exécution McRT [81] montre également que les mémoires transactionnelles permettent d'avoir un passage à l'échelle là où les verrous ne le permettent pas. Cela est vrai lorsqu'il y a peu de mises à jour et donc de possibilités de rollback. Dans ce cas, les mémoires transactionnelles permettent d'obtenir du parallélisme à grain très fin.

Cependant, de nombreux problèmes empêchent une adoption plus large des mémoires transactionnelles [28]. La performance de ces dernières est notamment un critère important. Pour les mémoires transactionnelles logicielles (*STM, Software Transactionnal Memory*), Larus *et al.* [62] parlent de ralentissement allant de 2 à 7 fois, mais avec un passage à l'échelle conservé. Inversement, Cascaval *et al.* mesurent sur un ensemble de tests un mauvais passage à l'échelle et des dégradations beaucoup plus importantes (entre 2 et 40 fois). Les mémoires transactionnelles peuvent être également gérées matériellement ou avec un mélange de mémoires transactionnelles logicielles et matérielles. L'utilisation de mémoires transactionnelles matérielles réduit significativement ce surcoût. Cependant, ces mémoires transactionnelles matérielles sont encore très peu déployées. Un autre défaut des STM est qu'elles ne surveillent que le code déclaré atomique. Si un code non atomique manipule une donnée d'un code atomique, le gestionnaire de transaction ne le verra pas et l'exécution sera potentiellement invalide. Notons que ce problème est également présent avec des verrous. Enfin, il est important de considérer que certaines opérations, telles que les E/S, sont très difficiles (voire impossible) à annuler.

OpenMP. OpenMP [2] est un ensemble de directives de pré-compilation ainsi qu'une bibliothèque de threads permettant de simplifier la programmation parallèle à base de threads. Les améliorations d'OpenMP (du point de vue du programmeur) sont de simplifier la programmation en permettant de générer automatiquement le code correspondant à une partie des opérations classiques. Par exemple, il est possible de définir facilement un fonctionnement de type *fork-join*. Ce mode de synchronisation permet d'effectuer des calculs concurrents et d'attendre le résultat de ces calculs avant de continuer la progression du fil d'exécution principal. Il est également possible de définir facilement le découpage d'une boucle sur plusieurs threads. Enfin, nous pouvons citer la possibilité de contrôler explicitement si une variable est partagée ou non.

Un autre avantage est de s'abstraire de la machine. Il n'est pas nécessaire de manipuler directement les threads donc le support d'exécution pourra s'adapter à la machine et démarrer, au besoin, le nombre de threads adéquats pour exploiter convenablement l'architecture matérielle.

Capriccio. L'objectif des auteurs de Capriccio [97] est de proposer une bibliothèque de threads malgré un nombre de threads importants. Pour cela, il a été nécessaire de considérer (i) le coût de l'ordonnancement, (ii) la surconsommation mémoire, et (iii) l'ordonnancement des threads en fonction des ressources disponibles.

Pour résoudre les coûts d'ordonnancement, Capriccio se base sur une bibliothèque de threads de niveau utilisateur, c'est-à-dire gérés au niveau utilisateur et pas par le noyau, ordonnancés de manière coopérative. Ce choix permet de réduire les coûts liés à la gestion des threads. Il n'est ainsi pas nécessaire d'effectuer un appel noyau (*syscall*) pour gérer certaines opérations telles que la création et la destruction de threads ou la prise de verrous. Les threads de niveau utilisateur présentent cependant

plusieurs inconvénients. Le principal est que le système d'exploitation n'est pas conscient que ces threads existent et ne les ordonnancera pas sur les machines multi-cœurs. Pour utiliser les machines multi-cœurs, il est nécessaire de définir une relation *N threads utilisateur* ⇒ *M threads noyau* où M est au minimum le nombre de cœurs de la machine (et éventuellement avec M = N). Notons que dans Capriccio, il n'existe pas de support des machines multi-cœurs.

Pour réduire les coûts de consommation mémoire, il est nécessaire de réduire la taille de la pile. Par défaut, la pile à une taille fixe. Cependant, cette taille est largement surdimensionnée pour pouvoir gérer tous les cas et notamment les applications récursives. Pour réduire la consommation mémoire de la pile, Capriccio se base sur un mélange d'analyse statique et de contrôle dynamique. Le compilateur crée un graphe où les nœuds sont les fonctions et les arcs les appels entre ses fonctions. Les nœuds sont décorés par la taille de la pile consommée par cette fonction. La taille de la pile à allouer initialement correspond au chemin consommant le plus de mémoire dans la pile. Pour gérer les cas de récursion, des points de vérification sont insérés pour vérifier à l'exécution s'il y a assez d'espace dans la pile. S'il n'y a plus d'espace disponible dans la pile, la taille de celle-ci est augmentée.

La dernière avancée de Capriccio consiste à ajuster l'ordonnancement en fonction des ressources disponibles. L'idée est de garder des informations sur l'utilisation des ressources globales ainsi que l'utilisation moyenne des ressources par un bloc. Un bloc représente la portion de code entre deux changements de contexte (volontaires). Ainsi, avec ces informations, si une ressource est surchargée (mémoire, nombre de descripteurs de fichier ouverts, etc.), il sera possible d'ordonnancer en priorité des blocs libérant cette ressource.

2.1.3 Programmation événementielle

Contrairement à la programmation à base de threads, la programmation événementielle utilise un seul fil d'exécution par coeur. Les traitements des événements sont entrelacés de manière coopérative. Ce choix implique que le traitement d'un événement ne sera jamais interrompu pour exécuter un autre traitant, mais devra explicitement redonner la main à l'ordonnanceur. Comme le traitement d'un événement est un appel de méthode, si un traitement complet nécessite une succession de traitements, le programmeur doit explicitement sauvegarder la partie du contexte nécessaire aux différents traitants. Cette sauvegarde est effectuée dans un *message* (ou *continuation*) [2]. Notons que les événements peuvent être soit des événements internes, générés par le traitement d'autres événements, soit des événements externes, générés par exemple par des périphériques d'E/S.

La figure 2.2 donne le pseudo-code de la boucle principale d'un serveur de données événementiel ainsi qu'une illustration du traitement d'un événement et de la génération d'un événement interne. Cette figure illustre la structuration d'un programme événementiel autour d'une file, stockant les événements à traiter (avec leurs continuations), et de traitants d'événements. Nous pouvons voir que dans le cas de ce serveur de données, nous avons un seul flot d'exécution qui gère l'ensemble des requêtes arrivant sur le serveur. Lorsqu'une demande de connexion ou une requête arrive, il s'agit d'une demande externe (émise depuis la carte réseau) [3]. Nous pouvons également voir des événements générés par le traitement d'autres événements. C'est le cas par exemple de l'envoi d'une réponse. Cet événement n'est pas déclenché à cause d'une interruption de la carte réseau, mais est posté par le traitement de la lecture de la requête. Notons que le découpage du serveur en un ensemble de traitants est arbitraire. Certains traitants sont liés aux E/S, d'autres sont choisis par le programmeur (par exemple, il est possible de fusionner *LectureRequête* et *EnvoiRéponse* en un unique traitant).

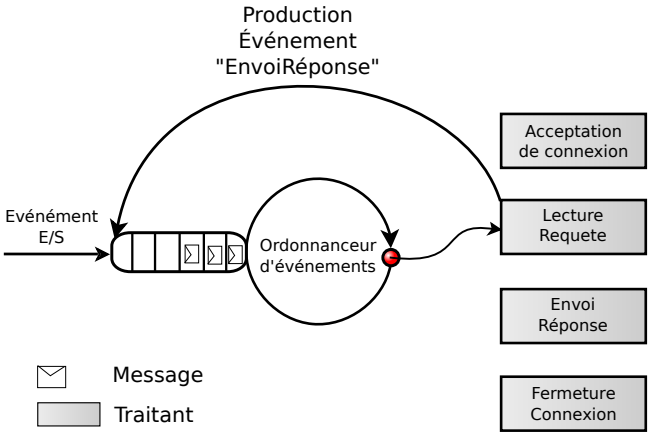
2. Nous utiliserons ces deux termes de manière interchangeable dans la suite de ce document

3. Plus précisément, ces événements réseau sont transformés en événement interne grâce à l'utilisation de primitives telles que *select* ou *epoll*. Nous décrirons ces mécanismes plus loin dans ce document.

```
Boucle principale :

while(true) {
  Événement e = récuperer_événement();
  switch(type_d_événement(e)){
    case "Nouvelle Connexion" :
      accepter_connexion(e);
    case "Nouvelle Requête" :
      lire_requête(e);
    case "Envoyer Réponse" :
      envoyer_réponse(e);
    case "Fermeture Connexion" :
      fermer_connexion(e);
  }
}
```

(a)



(b)

Figure 2.2 – (a) Pseudo-code de la boucle principale d'un serveur de données utilisant une programmation à base d'événements (b) Illustration de la production d'événements pour le même serveur. Les événements sont produits soit de manière interne soit depuis des requêtes réseau.

Il est intéressant de remarquer que la granularité du découpage est laissée au programmeur. En général, un traitant d'événements effectue une seule tâche (lire la requête, l'analyser ou encore envoyer la réponse). Cependant, il est possible de faire des événements très courts ou très longs.

Comme il n'est pas possible de traiter un autre événement si le serveur fait par exemple une E/S bloquante, il est nécessaire d'utiliser des opérations non bloquantes pour les E/S. Heureusement, la plupart des périphériques permettent d'utiliser des opérations non bloquantes.

Enfin, nous pouvons voir que, comme les connexions sont multiplexées au sein d'un même thread, les continuations associées aux événements doivent sauvegarder le contexte nécessaire pour le traitement de l'événement. Il faut sauvegarder, au minimum, l'identifiant de la socket concernée par l'événement, mais aussi les pointeurs vers les tampons de réception ou d'envoi.

Les avantages de ce modèle de programmation sont multiples. Le premier est qu'il est basé sur un seul flot d'exécution sans préemption. Cela permet de gérer aisément l'accès aux données partagées tout en délimitant finement les frontières des sections critiques (le début et la fin du traitement de l'événement). Dans l'exemple du serveur de données précédemment décrit, cet ordonnancement coopératif permet de proposer simplement du parallélisme entre les requêtes tout en gérant facilement les accès concurrents à la socket. Le fait de ne pas avoir de préemption permet de se passer de multiples piles de threads (utiles uniquement pour sauvegarder le contexte d'un thread lors d'un changement de contexte). La programmation événementielle bénéficie donc d'une empreinte mémoire réduite et est ainsi adaptée aux applications embarquées (ou aux serveurs de données avec des milliers de connexions en parallèle). De plus, la programmation événementielle s'utilise naturellement dans le cadre d'applications réagissant à des événements, telles que des serveurs de données ou des interfaces graphiques. Enfin, comme l'application ne dépend pas de spécificités du système d'exploitation, si ce n'est des E/S non bloquantes, la programmation événementielle est portable sur toutes les architectures.

Comme nous l'avons vu précédemment, les processeurs proposeront peut-être bientôt des mécanismes permettant de faire du passage de message efficace entre les cœurs. Dans cette vision, la programmation événementielle bénéficiera de ces améliorations en utilisant ces mécanismes lors de l'envoi d'événements d'un cœur à un autre.

2.1.3.1 Problèmes principaux

Le principal reproche fait à la programmation événementielle est connu sous le nom de *stack-ripping* [10]. Ce problème vient du fait que l'utilisateur doit gérer la sauvegarde, ainsi que la restauration du contexte manuellement lors du découpage de l'application en une succession de traitants. Ce découpage entraîne également une obfuscation du code de l'application. Contrairement à une approche séquentielle classique, il peut parfois être difficile de déterminer quel traitant engendre quels événements et d'avoir une vue d'ensemble de l'application. La gestion de la mémoire dynamique et des ressources partagées deviennent plus complexes. Par exemple, il est nécessaire de vérifier qu'il n'existe plus d'événement pouvant manipuler une donnée avant de la libérer. Une sémantique de passage de messages uniquement par copie permet de résoudre ce problème. Cependant, cette solution est moins flexible du point de vue du programmeur et plus coûteuse. La meilleure solution consiste à utiliser un mécanisme de ramasse-miettes qui libérera automatiquement les ressources qui ne peuvent plus être utilisées. Enfin, la granularité des événements est laissée à l'appréciation du programmeur. Dans l'exemple précédent du serveur de données, des tâches de petite granularité entraînent un surcoût de gestion, mais des tâches de grosse granularité ne permettent pas l'avancement en parallèle du traitement des connexions.

Deuxièmement, au milieu de ces nouvelles architectures parallèles, il est difficile d'imaginer un programme basé sur un seul thread. Ce choix architectural, s'il est très intéressant pour la gestion des données partagées, ne permet pas d'utiliser efficacement les processeurs. Il est uniquement possible de les utiliser partiellement puisque seuls les tâches du noyau ou d'autres processus pourront être mis sur les autres cœurs. La technique habituelle avec un système événementiel consiste à démarrer une instance de l'application par cœur. Cette technique est appelée *N-Copy*. Elle peut être mise en place lorsque les instances n'ont pas besoin de partager d'état. Par exemple, il est possible de mettre en place une telle technique dans le cas d'un serveur de données de type serveur Web où les données sont indépendantes, mais il n'est pas possible de le faire dans un serveur de type serveur de fichier où les écritures dans les fichiers doivent être cohérentes. Cette technique oblige également à dupliquer les données de l'application, ce qui augmente la consommation mémoire et peut, dans certains cas, réduire l'efficacité de l'application. Par exemple, dans un serveur Web, le cache applicatif sera dupliqué sur les différentes instances et cela réduira potentiellement la taille maximum de ce cache.

2.1.4 Amélioration du modèle d'événements

Les approches pour améliorer la programmation événementielle se concentrent surtout sur trois points essentiels : la facilité de programmation, la performance et le support des architectures multi-cœurs. Dans cette partie, nous allons détailler les différentes améliorations proposées.

Support d'exécution générique. Libasync [68] est une bibliothèque permettant de simplifier la programmation événementielle. Pour cela, Libasync propose un mécanisme générique d'enregistrement d'événements et d'appel de traitants. Il est également possible d'enregistrer des méthodes à appeler lors d'une activité sur un descripteur de fichier. Pour cela, Libasync utilise l'appel système *select*. Ce mécanisme, proposé par Linux, permet une surveillance des descripteurs de fichiers. Libasync offre également la possibilité de gérer facilement les continuations via un mécanisme de *wrap*. *Wrap* permet de créer les événements. Un événement est un pointeur sur le traitant à appeler ainsi que la continuation. *Wrap* permet d'associer facilement des continuations de taille variables à un événement. Il est également possible de déclencher un événement sur la sonnerie d'un compte à rebours. Enfin, Libasync offre la possibilité d'utiliser un mécanisme de ramasse-miettes. Ce mécanisme est utilisé de manière interne pour la gestion des événements (et des continuations) et est également proposé

au programmeur pour gérer ses données (une donnée peut être référencée par plusieurs événements). Une bibliothèque de fonctionnalités, déjà événementielles, est également fournie pour simplifier le développement d'applications. Il est ainsi possible d'insérer rapidement un appel DNS événementiel, tout en gardant la lisibilité de l'application.

Libevent [78] est une bibliothèque proposant des fonctionnalités similaires à Libasync sans les mécanismes de *wrap* et de ramasse-miettes. Libevent permet d'utiliser pour la partie réseau différents mécanismes, là où Libasync est limité au vieillissant *select*. Libevent supporte également les mécanismes d'envoi sans copie entre le code utilisateur et le noyau, tel que *sendfile*. Le code de Libevent permet d'utiliser plusieurs threads mais la synchronisation est laissée à la charge du programmeur.

Gestion des opérations bloquantes. Le projet Flash [75] consiste à écrire un serveur Web efficace. Ce serveur est construit autour d'une architecture événementielle. Cependant, les auteurs ont fait le constat que soit les primitives d'E/S non bloquantes n'étaient pas disponible à l'époque (1999) dans les systèmes d'exploitation, soit ces primitives n'étaient pas intégrées avec d'autres mécanismes tels que select. Ils ont donc inventé une nouvelle architecture pour Flash : l'architecture *AMPED, Asymmetric Multi-Process Event Driven*. Cette architecture est basée sur un cœur événementiel pour la gestion des connexions est des requêtes et délègue les E/S disque à des processus distincts. Lorsque l'E/S est terminée, le processus informe la boucle de contrôle de cette terminaison. Ce mécanisme permet de proposer une solution portable pour gérer efficacement les E/S bloquantes.

Amélioration des performances sur les architectures mono-cœur. Deux projets se sont intéressés à améliorer les performances des applications sur les architectures mono-cœur.

Le premier projet consiste en un mécanisme d'allocation mémoire efficace maximisant l'utilisation des caches [21]. L'idée de ce projet est de coupler l'allocateur mémoire et l'ordonnanceur d'événements. En utilisant une analyse statique, un graphe des relations entre les traitants est construit ainsi que la consommation mémoire de chacun des traitants. La consommation mémoire indique le nombre et la taille des objets que le traitant alloue ou libère. L'ordonnanceur va utiliser ces informations lors du choix du prochain événement à traiter. Ce choix sera effectué de manière à ne pas faire déborder l'utilisation du cache et à ne jamais accéder à la mémoire. Pour réaliser ce mécanisme, il est nécessaire que l'allocateur mémoire garde une trace de la quantité de mémoire allouée et que l'ordonnanceur dialogue avec le mécanisme d'allocation pour trouver la prochaine tâche à ordonnancer. Le programmeur utilisera donc les fonctions d'allocation mémoire du nouvel allocateur mémoire (*Stingy*) au lieu des traditionnels *malloc*. Pour gérer les allocations de mémoires dynamiques dont la taille n'est pas connue à l'avance, telles que les allocations dépendantes d'une requête, il est possible pour le programmeur de spécifier la taille probable grâce à des annotations. L'allocateur mémoire utilisé (Stingy ou l'allocateur du système) dépend de la justesse de la prédiction. L'interaction dans le cache entre les deux allocateurs n'est cependant pas détaillée. Toutefois, les auteurs argumentent que l'objectif est d'améliorer les serveurs de données et que de telles applications sont déterministes et que leur consommation mémoire peut être prévue.

Une autre possibilité pour réduire les coûts de la programmation événementielle est de réduire le nombre d'appels à des traitants d'événements. Ces appels peuvent être de deux types : synchrones, c'est-à-dire comme un appel de fonction, ou asynchrones, en postant un événement traité plus tard. Le problème est que le postage des événements introduit différentes sources de coûts telles que la sauvegarde et la restauration de la continuation, le coût des appels de méthodes ou encore la duplication des vérifications. Rajagopalan *et al.* [79] proposent d'essayer de fusionner les traitants pour réduire les coûts d'appel et mettre en commun les vérifications. Pour cela, ils se basent sur un premier profilage permettant de déterminer l'arbre d'appel des traitants. Ensuite, un arbre réduit est créé ne contenant que les chemins fréquemment accédés. Deux optimisations sont possibles. Lorsqu'un

événement nécessite systématiquement le passage par plusieurs traitants, ces derniers peuvent être fusionnés. Lorsqu'un événement provoque le traitement d'un autre événement de manière synchrone, dans ce cas également, il est possible de fusionner les événements. Pour gérer les cas non couverts par le profilage initial, les traitants non fusionnés sont également présents.

Amélioration du modèle de programmation. Le projet Tame [60] s'attaque au problème de *stack-ripping* et la facilité de programmation des programmes événementiels en général. Pour cela, un ensemble de structures syntaxiques ont été définies : (i) la possibilité de lire le code de l'application comme un programme séquentiel, (ii) la sauvegarde automatique des continuations (nommées *closures*), et (iii) la possibilité de se synchroniser sur le résultat d'un ou plusieurs événements avant de continuer l'exécution. Notons toutefois qu'une fonction asynchrone (*tamed*) ne peut pas retourner de valeur. Tame propose également un support rudimentaire des bibliothèques de threads coopératives. Cela permet de mélanger au sein de la même application la programmation événementielle et la programmation utilisant des threads coopératifs. Ainsi, cela facilite la transition du second modèle de programmation vers le premier. Du fait de l'utilisation de multiples threads, il est nécessaire de gérer correctement les accès concurrents à des ressources partagées. Pour simplifier la programmation, Tame gère automatiquement la mémoire et notamment les continuations. Notons que l'implantation de Tame se base sur une traduction du langage Tame vers du C++ et sur la bibliothèque Libasync, précédemment décrite, pour gérer les événements.

Utilisation des architectures multi-cœurs. Le gros problème de la programmation événementielle est de ne pas permettre l'utilisation des nouvelles architectures multi-cœurs. Libasync-smp [102, 35] est une extension de Libasync permettant d'offrir un support de ces architectures. Le principe est de répartir les événements sur les différents cœurs disponibles. Cependant, comme pour la programmation par threads, il est nécessaire de gérer la concurrence entre le traitement des différents événements. En effet, ce traitement peut concerner des données globales à l'application, par exemple un compteur, ou plusieurs continuations peuvent contenir des pointeurs sur la même ressource, ou encore une socket. Pour gérer ce problème, Libasync-smp s'appuie sur un mécanisme original de couleurs. L'idée est de décharger la gestion des verrous autant que possible du programmeur tout en garantissant la correction du programme. Le mécanisme de couleurs de Libasync-smp permet de fournir ces garanties. Le programmeur va *colorier* (ou annoter) ses événements. Le support d'exécution garantit que plusieurs événements de la même couleur ne seront jamais traités simultanément. Ce mécanisme permet de faire une exclusion mutuelle à l'échelle du traitant. Les sémantiques d'exclusion mutuelle complexes ne peuvent pas être exprimées avec la coloration d'événement. Cependant, les auteurs précisent que ces mécanismes ne sont généralement pas utiles et sont même à prohiber, car ils sont sources d'erreurs. Un des intérêts du mécanisme de couleurs est de pouvoir injecter progressivement du parallélisme. Plus de détails seront donnés sur Libasync-smp dans le chapitre 5. Il est important de noter que Libasync-smp garantit également la non-préemption du traitement d'un événement. Cette garantie simplifie grandement la gestion de l'exclusion mutuelle, mais implique d'avoir des traitants non bloquants.

La coloration des événements simplifie la programmation événementielle multi-cœurs. Ce mécanisme ne garantit cependant pas que l'exécution du programme soit correcte. En effet, le programmeur doit encore manuellement colorier ses événements, ce qui peut engendrer des problèmes de synchronisation, bien que le support d'exécution garantisse l'absence d'interblocages. Jannotti *et al.* [53] prônent la coloration automatique des événements pour garantir la sûreté d'exécution, même si cela doit se faire au détriment de la performance. Leur approche permet de détecter des contraintes entre traitants (par exemple pour l'accès à une variable globale) où entre événements (par exemple pour l'accès à une donnée partagée passée dans la continuation). Pour gérer une synchronisation efficace,

ils introduisent deux types d'annotations : les teintes pour l'exclusion entre traitants et les couleurs pour l'exclusion entre événements. Pour le moment, seul un prototype permettant de détecter les contraintes sur les données globales a été implanté sans évaluation de performance.

2.1.5 Approches visant à unifier les deux modèles

Les architectures événementielles et les architectures à base de threads sont très proches. Lauer et Needham [63] ont relevé cette équivalence dans les modèles de programmation et d'exécution. Adya *et al.* [10] résume la différence entre les deux modèles à la gestion de la pile, qui peut être soit automatique soit manuelle, et la gestion des tâches, qui peut également être automatique ou manuelle. Cela donne donc quatre choix :

- *Une gestion automatique des tâches et une gestion automatique de la pile.* C'est le modèle de thread préemptible classique.
- *Une gestion manuelle des tâches et une gestion automatique de la pile.* C'est le modèle de thread coopératif. Tame peut également rentrer dans cette catégorie.
- *Une gestion manuelle des tâches et une gestion manuelle de la pile.* C'est le modèle de programmation événementiel.
- *Une gestion automatique des tâches et une gestion manuelle de la pile.* C'est un modèle de programmation dans lequel lors de la préemption, la sauvegarde du contexte est laissée au programmeur. Ce modèle de programmation est impossible à mettre en place, car il nécessite de connaître à chaque instant les données importantes à sauvegarder.

Ces similarités ont fait que plusieurs approches ont essayé d'unifier la programmation événementielle et la programmation par threads. Nous allons les décrire dans la suite de cette section.

Les fibers. Après avoir fait le constat que ce qui différencie réellement la programmation événementielle et la programmation par threads est la gestion de la pile et des tâches, Adya *et al.* ont proposé un mécanisme permettant d'unifier la programmation par thread *coopérative* et la programmation par événements. Les deux types de programmation diffèrent uniquement par la méthode de gestion des tâches. Le programmeur utilise des *fibers* en utilisant le mode de gestion des taches qui lui convient (préemption ou coopératif). Une *fiber* est liée à un contexte d'exécution qui correspond soit à la pile d'exécution lorsque la sauvegarde automatique est choisie soit à une continuation lorsque le mode de sauvegarde manuel est choisi. L'ordonnanceur ne gère lui que des *fibers* quelque soit la méthode choisie. Lorsqu'une *fiber* est ordonnancée, le contexte (c'est-à-dire la pile ou la continuation) est restauré.

Une approche similaire a été proposée pour le langage fonctionnel Haskell [65]. Cette approche se base sur les *monads*, une unité d'abstraction des calculs en Haskell. Les événements comme les threads sont considérés comme des *monads*. L'évaluation des *monads* peut être faite de manière paresseuse, c'est-à-dire uniquement au besoin.

Erlang. Erlang [15] est un langage de programmation fonctionnel ayant mis au cœur de son architecture la robustesse et la répartition de charge. Ce langage est construit autour de processus légers, semblables à des threads, mais ne partageant pas de mémoire. La communication entre les processus d'une même application est effectuée par envoi de message. Chaque processus dispose de sa file d'événements dans laquelle le processus récupère les messages à traiter. L'envoi des messages est asynchrone, mais fiable, c'est-à-dire que tout message envoyé sera reçu, et FIFO (*First In First Out*),

c'est-à-dire que, pour un même émetteur, les messages seront reçus dans l'ordre d'envoi. Comme le partage de mémoire n'est pas autorisé dans le langage Erlang, tous les envois sont effectués par copie. La problématique d'envoi et de réception de messages étant au cœur d'Erlang, ces deux fonctionnalités sont nativement intégrées au langage.

Ce modèle de programmation encourage les concepteurs d'applications à penser leurs applications selon le modèle d'acteurs [11]. L'utilisation de multiples processus légers et l'interdiction du partage de mémoire entre ceux-ci permettent de distribuer facilement les processus entre plusieurs cœurs d'une même machine voire entre plusieurs machines. Contrairement au modèle à base de threads dans lequel les programmeurs doivent, pour des raisons de performances, limiter le nombre de threads simultanément utilisés, un très grand nombre de processus Erlang peut être employé puisque le modèle de programmation découple le modèle de processus de son exécution.

Erlang connaît un succès important dans le domaine des messageries instantanées dans lequel la possibilité de paralléliser de manière importante le traitement des discussions est un atout indéniable. Il est notamment utilisé par la messagerie instantanée du service Facebook.

Flux. Flux [27] propose de séparer le modèle de programmation du modèle d'exécution. Pour cela, le programmeur découpe son application en nœuds. Ces nœuds sont l'équivalent des traitants pour la programmation événementielle ou les fonctions de la programmation par threads. Ensuite le programmeur explicite les relations entre les nœuds et les contraintes d'atomicité (c'est-à-dire si un nœud doit être exécuté en exclusion mutuelle). Les contraintes d'atomicités ne peuvent être exprimées que sur des nœuds. Il n'est pas possible de définir une contrainte d'intégrité sur le traitement du nœud. Par exemple, il n'est pas possible, dans un serveur Web, de spécifier des contraintes d'atomicité en fonction du fichier accédé. L'avantage de définir les verrous dans un langage de haut niveau permet de ne pas se retrouver dans des situations d'interblocage. Notons toutefois que cela n'empêche pas les accès concurrents invalides à la mémoire si le programmeur ne spécifie pas bien ses contraintes. Le langage de Flux est par la suite traduit vers le langage C++. Il est possible de choisir entre différents supports d'exécution (événementiel, threads ou SEDA). Après transformation, le programmeur doit remplir le code des nœuds, sans se soucier du modèle d'exécution.

2.2 Nouveaux modèles de programmation multi-cœurs

En plus des améliorations aux modèles d'exécution traditionnels, différentes approches ont été étudiées pour proposer du parallélisme aux applications. Généralement, ces approches permettent de résoudre de manière efficace un type de problème bien défini (traitement de données de tailles importantes, algorithmes parallèles, etc.). Dans cette section, nous allons voir un ensemble de modèles de programmation spécialisés pour les architectures multi-cœurs (ou pour les architectures multi-processeurs).

2.2.1 Modèles de programmation *fork-join*

Cilk [23] est un modèle de programmation ainsi qu'un support d'exécution qui expose le parallélisme au programmeur. Ce dernier se concentre uniquement sur la structuration des calculs en opérations parallèles disjointes et laisse le support d'exécution ordonnancer efficacement ces opérations sur l'architecture multi-cœurs sous-jacente. Le modèle de calcul utilisé par Cilk est un modèle *fork-join*. En cela, Cilk ressemble au modèle de programmation OpenMP précédemment décrit (voir section 2.1.2). L'application est écrite de manière séquentielle. Sans les extensions spécifiques à Cilk, l'exécution d'un programme serait faite dans un seul thread. Cilk définit quatre extensions au langage de programmation C. La première consiste à pouvoir démarrer le traitement d'une fonction en

```
cilk int fib (int n) {
  if(n < 2) return n;
  else{
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);

    sync;

    return (x*y);
  }
}
```

```
cilk int main (int argc, char * argv){
  int n, result;
  n = atoi(argv[1]);

  result = spawn (fib(n));
  sync;

  printf("result = %d\n", result);
  return result;
}
```

Figure 2.3 – Exemple de programme Cilk permettant de calculer la suite de Fibonacci.

parallèle (*spawn*). Le mot clé *sync* permet d'attendre la fin des tâches lancées en parallèle avant de continuer l'exécution. Comme les fonctions lancées en parallèle sont considérées comme des fonctions classiques, il est possible de récupérer la valeur de retour de la fonction. Cependant, fort logiquement, cette valeur n'a de sens qu'après l'exécution d'un *sync* qui assure que la fonction s'est bien entièrement exécutée. Il est également possible de déclencher une fonction lorsqu'une fonction lancée en parallèle a fini de s'exécuter (*inlet*). Le modèle de programmation de Cilk crée ainsi des graphes d'exécution. Les nœuds de ces graphes (c'est-à-dire les threads) seront ordonnancés sur les cœurs disponibles. Il est important de noter que Cilk propose d'utiliser en dernier recours des verrous pour gérer les accès concurrents aux données partagées, mais que le langage et le modèle d'exécution ne sont pas pensés pour utiliser de telles primitives. Le programmeur doit explicitement structurer ses calculs pour travailler sur des données indépendantes.

La figure 2.3 présente l'exemple classique du calcul de la suite de Fibonacci en utilisant le langage Cilk [4]. Nous pouvons voir que la fonction `fib` est déclarée comme `cilk`, c'est-à-dire pouvant être appelée dans un thread séparé. Chaque appel à `fib` crée deux nouveaux threads. Le résultat de ces deux threads est attendu avant de continuer le calcul. La structure de l'exécution du programme est donc bien un arbre chaque nœud ayant deux fils, la seule exception étant pour le calcul d'un n inférieur à 2.

TBB [61] est un projet initié par Intel pour programmer efficacement les architectures multi-cœurs. Intel est basé sur les concepts de Cilk. Cependant, contrairement à Cilk qui nécessite un compilateur modifié, Intel se base sur des *templates* C++. L'avantage est que le mécanisme fourni fonctionne sur n'importe quel compilateur C++. L'inconvénient est que la syntaxe est légèrement plus compliquée.

2.2.2 Programmation par étages

SEDA [98] (*Staged-Event Driven Architecture*) est un modèle de programmation pour construire des serveurs d'applications résistants à la charge. Le postulat de base est que les serveurs Web à base de threads ne résistent pas à la charge dès que le nombre de clients devient trop important, principalement à cause du coût de la gestion des threads. La programmation événementielle est plus adaptée pour le développement des serveurs de données. Cependant, elle souffre aussi de problèmes tels que la nécessité d'avoir des E/S non bloquantes ou la difficulté d'avoir des stratégies efficaces d'ordonnancement des événements puisque tous les événements sont dans une même file. SEDA est donc une nouvelle façon de structurer les applications. L'application est découpée en un ensemble d'étages. L'interaction entre les étages est effectuée de manière asynchrone. Ces étages sont ana-

4. Cet exemple est tiré du papier décrivant la version 5 du langage Cilk [43]

logues aux traitants dans la programmation événementielle. Cependant, chaque étage dispose de sa propre file d'événements et d'un ensemble de ressources d'exécution (threads). Le gros intérêt de cette architecture est qu'il est possible d'affecter plus de threads à un étage qui serait saturé pour lui permettre de se désengorger, tout en ralentissant la production des étages précédents en diminuant le nombre de threads correspondant. Plus d'informations sur l'ordonnancement de SEDA sont données dans le chapitre 3. Cependant, ce modèle de programmation amène d'importantes difficultés pour le programmeur. En effet, le programmeur doit gérer les accès concurrents aux données partagées à la fois intra- et inter-étages. Nous retrouvons ici toute la difficulté du modèle de programmation par thread en terme de concurrence. SEDA n'est pas un modèle créé spécialement pour les architectures multi-cœurs mais permet de les utiliser du fait de la présence de plusieurs threads.

2.2.3 Files d'éléments

Click [71] est un langage de programmation permettant de construire des routeurs logiciels modulaires. Les routeurs logiciels ont l'avantage de présenter un potentiel d'évolution important par rapport aux routeurs matériels. Le gros problème des routeurs logiciels est leur performance. Click est donc un projet visant à fournir une architecture pour construire des routeurs performants et modulaires. L'application est structurée autour d'*éléments*. Une analogie peut être faite entre les éléments et les traitants de la programmation événementielle. Cependant, la *connexion* entre ces événements est réalisée de manière synchrone avec des *ports*. Un élément peut donc avoir des ports d'entrée et de sortie. Les points d'entrée et de sortie d'une chaîne d'éléments sont les interfaces réseau du routeur. Il est possible de rendre la communication entre éléments asynchrone en insérant explicitement des files de messages. Click dispose de son propre langage pour décrire les éléments, leurs ports ainsi que les connexions.

SMP Click [31] intègre un support des architectures multi-cœurs au sein de Click. Le principe consiste à répartir les blocs d'exécution synchrones. Un bloc d'exécution synchrone regroupe tous les éléments interagissant de manière synchrone entre deux files de messages. Les blocs sont répartis sur les cœurs. Cette affectation est réévaluée périodiquement. Lorsque le thread ordonnance un bloc, il vérifie s'il a des messages dans sa file d'entrée. Deux types de parallélisme sont disponibles. Le premier consiste à exploiter le parallélisme entre blocs. Dans ce cas, un paquet est traité par le premier bloc puis est passé au second bloc en changeant potentiellement de processeur. Le second type de parallélisme consiste à dupliquer les blocs pour avoir du parallélisme entre paquets. Ici, un paquet ne changera pas de processeur lors de son traitement. Le projet RouteBricks [38] a montré que la seconde option est plus performante pour construire des routeurs logiciels.

2.2.4 MapReduce

```
Map (String key, String value) :

  for each work w in value :
    EmitIntermediate(w, 1);
```

```
Reduce (String key, Interator values) :

  int result = 0;
  for each value v in values :
    result += v;

  Emit(Key, result);
```

Figure 2.4 – Exemple de phase Map et Reduce permettant de compter les mots d'un texte.

MapReduce [37] est une adaptation pour les architectures réparties d'un modèle de programmation fonctionnel. L'objectif de ce modèle de programmation est de pouvoir traiter efficacement

de larges quantités de données. Le traitement est alors réparti sur un ensemble de machines d'une grappe. Ranger *et al.* [80] ont proposé un support pour ce modèle de programmation sur les architectures multi-cœurs. Les cœurs sont considérés comme étant les nœuds de la grappe. Pour traiter les données, différentes étapes sont appliquées. Pour illustrer le fonctionnement de MapReduce, nous allons prendre l'exemple d'une application comptant les mots dans un texte. La figure 2.4 présente le pseudo-code pour les phases *Map* et *Reduce* de cette application. La première étape consiste à découper les données par bloc (*split*). Dans notre exemple, la taille d'un bloc peut être de X caractères (en considérant toujours des mots complets). Ces blocs sont répartis sur chacun des cœurs. Chaque cœur va compter les mots de chacun des blocs (phase de *Map*). A chaque mot trouvé, une valeur est ajoutée dans une liste associée au mot (*EmitIntermediate*. Ici, la valeur ajoutée est 1 puisqu'un mot a été identifié. La phase de *map* génère donc un ensemble de couples (clé, valeur). La phase suivante consiste à combiner pour une même clé les valeurs pour n'avoir plus que des couples (clé, valeur) avec une seule occurrence de chaque clé (*combine*). Cette étape est effectuée sur chacun des processeurs. Elle n'est pas représentée dans le pseudo-code mais est équivalente, dans ce cas, à la phase de *reduce*. Pour chaque clé, le couple (clé,valeur) est envoyé au processeur chargé de la phase de *reduce*. Cette phase consiste, pour chaque mot, à combiner le nombre d'occurrences remontées par les différents cœurs. Pour un mot donné, il n'y a qu'un seul cœur chargé de la phase de *reduce*. Ce cœur est choisi en utilisant une fonction de *modulo*. Enfin, les résultats sont fusionnés (*merge*) dans un ordre défini. MapReduce est un modèle très efficace pour gérer en parallèle de grandes quantités de données. Le modèle n'utilise pas de verrous puisque les cœurs manipulent des données disjointes. La grande partie des manipulations étant locales, les caches processeurs ont une bonne efficacité. Ranger *et al.* ont pu ainsi montrer des améliorations de performances super-linéaires. Notons que la version MapReduce distribuée gère également les fautes des machines. Ce support n'est pas forcement nécessaire sur les architectures multi-cœurs.

2.2.5 Grand central

Grand Central [26] est une technologie permettant de programmer efficacement les architectures multi-cœurs. Cette technologie a été introduite dans MacOS X 10.6 et est également disponible en partie dans FreeBSD. Le modèle de programmation de Grand Central est très proche de celui proposé par Libasync-smp avec du sucre syntaxique. L'application est structurée en blocs. Les blocs sont l'équivalent des traitants d'événements. Tout comme le traitement d'un événement, l'exécution d'un bloc n'est pas préemptée. Il est possible de créer facilement des continuations. Pour gérer la concurrence entre les événements, ces derniers sont placés dans des files par le programmeur. Deux types de files existent : les files globales qui ne donnent aucune garantie sur l'exécution parallèle de deux événements et les files privées, gérées par le programmeur, qui garantissent que les événements dans la file sont exécutés en série. Notons également l'existence de la file principale fonctionnant également en série. Les files se rapprochent du modèle de couleur défini par Libasync-smp mais les traitants peuvent être bloquants puisqu'il y a un ensemble de threads par cœur.

2.3 Bilan

Nous avons vu dans ce chapitre que les modèles les plus utilisés pour développer des applications sont basés sur les threads et les événements. De nombreux travaux se sont intéressés à améliorer, soit la performance, soit l'expressivité et la simplicité d'utilisation de ces modèles. Nous avons vu notamment que la difficulté majeure de la programmation concurrente se situe dans la gestion correcte des accès concurrents à des données partagées. La programmation événementielle ainsi que la

programmation par threads ont des extensions permettant une gestion plus aisée de ces accès concurrents. Nous avons vu également que des approches essayent d'unifier les deux approches. En effet, des équivalences existent entre les deux modèles. La différence se joue essentiellement sur la gestion des tâches et la gestion de la pile. D'autres approches ont essayé de fournir des modèles de programmation efficaces pour la programmation d'applications concurrentes. Ces modèles sont généralement spécialisés pour un type de parallélisme.

Nous pensons que le modèle de programmation événementiel *colorié* est un modèle efficace pour le développement d'applications concurrentes. Plusieurs avantages existent pour ce modèle. Premièrement, les couleurs permettent d'exprimer de façon simple et efficace les opérations à sérialiser. Le programmeur peut injecter incrémentalement du parallélisme dans l'application. Les couleurs permettent d'exprimer plusieurs types de synchronisation. Il serait cependant intéressant de pouvoir spécifier des relations du type *lecteurs-rédacteur*. Un autre avantage des couleurs est de découpler la spécification de l'exclusion mutuelle de l'exécution. Le support d'exécution est chargé d'ordonnancer les événements en tenant compte des contraintes exprimées par le programmeur. Une version de Tame, adaptée pour gérer le modèle des couleurs, apporterait le sucre syntaxique nécessaire à une utilisation efficace du modèle événementiel. Enfin, le modèle de programmation événementiel serait adapté pour l'utilisation des mécanismes de passage de messages entre les cœurs, si cela devait être introduit dans les futurs processeurs.

Chapitre 3

Environnements d'exécution pour architectures multi-cœurs

Sommaire

3.1 Repenser les systèmes d'exploitation pour les architectures multi-cœurs 41
3.1.1 Rôles principaux du système d'exploitation 42
3.1.2 Architecture des systèmes d'exploitation 44
3.1.3 Systèmes d'exploitations pour les architectures multi-cœurs 46
3.2 Repenser l'exécution des tâches sur les architectures multi-cœurs 51
3.2.1 Équilibrage de charge 52
3.2.2 Affinités et compétition 54
3.2.3 Allocation mémoire efficace 56
3.3 Bilan 58

Dans le chapitre précédent, nous avons étudié les modèles de programmation existants. Ces modèles ont en commun le fait de créer des tâches (qui peuvent être des threads ou des événements). Le traitement de ces tâches doit être affecté de manière efficace aux cœurs disponibles. Pour cela, l'application repose à la fois sur des mécanismes fournis par le système d'exploitation, ainsi que sur des environnements d'exécution de niveau utilisateur.

Dans cette section, nous allons étudier les différentes couches interagissant pour exécuter une application. Dans une première partie, nous ferons un tour d'horizon de l'architecture des systèmes d'exploitation. Nous expliquerons notamment les problématiques multi-cœurs dans ces systèmes. Dans une seconde partie, nous nous intéresserons spécifiquement à la problématique de l'ordonnancement des tâches sur les cœurs disponibles. Cet ordonnancement peut être soit de niveau utilisateur, soit de niveau noyau. Enfin, nous verrons quels sont les points que le développeur doit considérer lors de la conception d'une application multi-cœurs.

3.1 Repenser les systèmes d'exploitation pour les architectures multi-cœurs

Le système d'exploitation est composé d'un noyau et de bibliothèques permettant l'exécution d'applications. Pour bien comprendre les améliorations apportées aux systèmes d'exploitation dans le contexte multi-cœurs, nous allons premièrement détailler les rôles d'un système d'exploitation, ainsi que les différentes architectures possibles pour le noyau. Nous allons voir notamment que, en

fonction de l'architecture du noyau, le rôle de celui-ci peut être plus ou moins complexe. Enfin, nous verrons pourquoi l'apparition des architectures multi-cœurs nécessite de repenser les systèmes d'exploitation, puis nous présenterons les travaux menés sur ce sujet.

3.1.1 Rôles principaux du système d'exploitation

Deux rôles principaux sont généralement accordés au système d'exploitation. Le premier rôle consiste à fournir une abstraction de l'architecture matérielle (processeurs, périphériques) de la machine pour permettre à un concepteur d'application d'éviter le plus possible les considérations de très bas niveau. Le second rôle consiste à multiplexer de façon sûre les ressources disponibles entre les différentes applications. La figure 3.1 présente une vue schématisée du système d'exploitation.

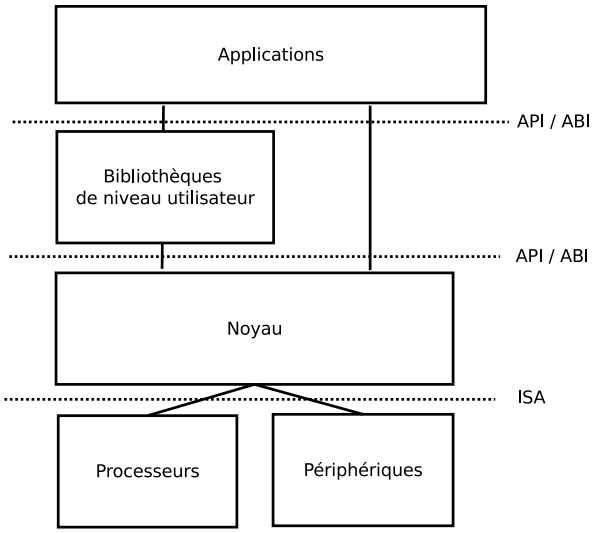


Figure 3.1 – Architecture classique d'un système d'exploitation.

Cette figure illustre le fonctionnement en couches d'un système d'exploitation. Chaque couche présente une interface à la couche de plus haut niveau et utilise les interfaces exposées par les couches de plus bas niveau. Par exemple, comme nous l'avons vu dans le chapitre 1, le processeur présente au système d'exploitation une interface appelée *ISA (Instruction Set Architecture)*. Le processeur expose au noyau uniquement ce qui lui est nécessaire pour son utilisation. Il est par exemple indispensable d'exposer au programme les registres et le jeu d'instructions, mais aussi des informations telles que le nombre de cœurs. Inversement, les détails de l'architecture interne du processeur (ALU, FPU, pipeline, exécution désordonnée, etc.) ne sont pas exposés au noyau. L'ISA est ainsi déjà une couche d'abstraction de la micro-architecture du processeur. Le noyau présente aux couches supérieures une interface de programmation (*API, Application Programming Interface*). Cette interface est utilisée par le programmeur lors du développement de l'application pour effectuer les appels systèmes (*syscalls*). Le noyau du système définit également une interface binaire (*ABI, Application Binary Interface*). Cette interface est utilisée à l'exécution pour définir des standards nécessaires à la cohabitation entre le noyau et les couches plus hautes. Par exemple, l'ABI définit la sémantique du passage de paramètres à une fonction (e.g. passage des paramètres par la pile ou par les registres) ou encore le format binaire des applications pour pouvoir les charger. L'application peut directement se servir de l'interface proposée par le noyau ou utiliser d'autres couches d'abstraction. Une couche fréquemment utilisée est la bibliothèque standard du langage C *(libc, C standard library)* qui possède différentes implantations. Ces bibliothèques de plus haut niveau proposent également une API. L'ABI est néces-

sairement commune au système d'exploitation pour la cohabitation des applications.

Le premier rôle du noyau est d'abstraire la gestion des ressources. Pour cela, il est chargé du dialogue avec le processeur et les périphériques. Ces composants communiquent avec le noyau au moyen d'interruptions et d'exceptions. Les interruptions sont générées soit par les processeurs, soit par les périphériques. Par exemple, le minuteur (*timer*) du processeur génère périodiquement des interruptions. Pour un périphérique, tel qu'une carte réseau, lorsque celui-ci reçoit une nouvelle information, une interruption est générée pour informer le noyau que des données sont arrivées. Les requêtes d'interruption (*IRQ, Interrupt Request*) sont générées sur une ligne d'IRQ. Le nombre de lignes d'IRQ est limité, ce qui fait que deux périphériques peuvent partager une même ligne. Les exceptions (*trap*) sont générées par le processeur. Une exception peut être déclenchée par le processeur lui-même sur une faute (défaut de page, division par zéro, etc.) ou être déclenchée manuellement par l'application (par exemple pour un appel système). Lorsqu'une interruption ou une exception est générée, le flot d'exécution est dérouté vers le traitant préalablement enregistré.

Le noyau du système d'exploitation tourne avec le mode de privilège le plus élevé, c'est-à-dire qu'il a accès à toutes les possibilités proposées par le processeur. Le processeur fournit plusieurs modes de protection. Les plus couramment utilisés sont le mode superviseur et le mode protégé. Dans le mode superviseur, il est possible d'avoir accès à toutes les instructions et de modifier tous les registres (y compris les registres d'état de l'application en cours). Inversement, en mode protégé, l'application n'a pas accès à toutes les instructions et ne peut manipuler que de la mémoire virtuelle. Nous avons vu dans le chapitre 1 que pour garantir qu'une application ne peut pas lire ou modifier les données d'une autre application, une solution est de s'appuyer sur un mécanisme de pagination. Le système d'exploitation est responsable du maintien de la table des pages de chaque processus. Lorsque le processeur ne trouve pas la correspondance pour une adresse virtuelle dans le TLB, il consulte la table des pages du processus. L'emplacement de la table des pages est indiqué dans un registre d'état. S'il n'y a pas d'adresse physique associée à l'adresse virtuelle, le processeur génère une exception qui sera traitée par le noyau. Dans ce cas, le rôle du noyau consiste à allouer une nouvelle page physique pour prendre en compte cette adresse virtuelle. Inversement, lorsqu'une page mémoire est libérée par le processus et retirée de sa table des pages, il est nécessaire de supprimer l'entrée correspondante dans le TLB du processeur si elle y est présente (*TLB shootdown*).

Comme nous l'avons vu dans le chapitre 2, le système d'exploitation définit deux abstractions : les processus et les threads. Un processus est composé d'un ou plusieurs threads. Dans ce cas, les threads se partagent certaines propriétés du processus telles que l'espace de mémoire virtuelle, ou encore la liste des descripteurs de fichiers ouverts. Les threads composant le processus peuvent être de niveau utilisateur ou de niveau noyau. Lorsque les threads sont de niveau noyau, cela veut dire qu'ils sont ordonnancés par le noyau. Un processus est toujours composé d'au moins un thread. Pour être conscient de la présence de plusieurs threads, le noyau maintient une table de tous les threads créés. Cette table contient notamment l'état des threads noyau. Le thread peut être soit en cours d'exécution sur un des cœurs, soit bloqué (par exemple en attente d'une E/S), soit prêt à être ordonnancé. L'ordonnanceur choisit le prochain thread à exécuter dans la liste des threads prêts. Pour garantir la progression de tous les threads, l'ordonnanceur affecte un quantum de temps à chacun. Lorsque le quantum est excédé, le thread suivant est ordonnancé. Il est possible de modifier les priorités d'exécution associées aux threads. Avec le noyau *Linux*, modifier la priorité d'un processus n'implique pas que ce processus va être ordonnancé plus fréquemment qu'un autre processus, mais que ce processus aura un quantum de temps associé plus important. Comme le noyau ordonnance uniquement des threads et non pas des processus, ce modèle d'ordonnancement ne garantit pas l'équité de temps entre les applications, mais entre les threads.

Lors d'un changement de contexte, il est nécessaire de sauvegarder l'état du thread en cours. Cela implique de sauvegarder notamment le compteur ordinal (c'est-à-dire l'adresse mémoire de

l'instruction en cours d'exécution), l'état des registres généraux ou spéciaux tels que le pointeur de pile. Si le changement de contexte est effectué pour un thread d'un autre processus, il est également nécessaire de changer le pointeur vers la table des pages ainsi que de vider le TLB s'il n'est pas annoté avec le numéro de processus.

Dans un système complètement préemptible, l'exécution peut être interrompue à n'importe quel instant (y compris dans le noyau). Cela nécessite pour le noyau d'apporter un soin tout particulier à la gestion des accès concurrents aux ressources partagées. Par exemple, lorsque le noyau affecte de la mémoire virtuelle à une application, il est nécessaire de garantir que, même si le noyau est préempté, le même espace de mémoire physique n'est pas affecté à deux processus distincts.

Pour effectuer un appel système, l'application génère tout d'abord une exception. Le noyau est dérouté sur le traitant de l'exception. Il va donc passer en adressage physique. Pour manipuler les données passées en paramètre de l'appel système, le noyau fait une copie de ces données dans son espace d'adressage (c'est-à-dire l'espace de mémoire réel). Le passage en mode noyau nécessite un changement de contexte et il est donc nécessaire de sauvegarder l'état des registres ainsi que le compteur ordinal. Le noyau traite ensuite l'appel système puis restaure le thread appelant en lui restaurant son contexte ainsi que son espace d'adressage. Un appel système a donc un coût très élevé du fait des coûts inhérents aux changements de contextes ainsi que de la copie de données entre les modes utilisateur et noyau. Certains appels système permettent de ne pas effectuer cette copie pour certaines opérations réduisant ainsi nettement les coûts. Cette méthode est appelée *zero-copy*. C'est le cas de l'appel système `sendfile`, disponible dans le noyau Linux, qui est utilisé pour transférer des données d'un descripteur de fichier à un autre. Le principe de la solution est d'éviter la copie en effectuant cette opération uniquement en mode noyau, alors que les primitives traditionnelles de manipulation de descripteurs de fichier nécessitent un passage par le mode utilisateur. Cette solution est fréquemment employée pour envoyer un fichier à un client via une socket ou pour faire de la copie de fichiers.

3.1.2 Architecture des systèmes d'exploitation

Dans cette section, nous allons détailler l'organisation d'un noyau. Trois classes d'architectures existent : le noyau monolithique, le micro-noyau et l'exo-noyau. Ces architectures diffèrent essentiellement par les services qui sont proposés par le noyau et par les modes d'interaction entre ces services.

3.1.2.1 Noyaux monolithiques

Un système d'exploitation peut se baser sur un noyau monolithique. Un noyau monolithique propose un ensemble de services (gestion des pages, gestion des pilotes de périphériques, pile IP, etc.) disponibles dans l'espace mémoire du noyau. Ces services sont en fait des appels de fonctions. Par exemple, en plus de la gestion basique de la carte réseau, le noyau peut être en charge différents protocoles de la couche *OSI (Open Systems Interconnection)*. Dans le modèle OSI, les protocoles sont vus comme une pile, des couches les plus basses (matériels) aux couches les plus hautes (applications). Dans ce modèle, le noyau peut être en charge de la gestion de la couche réseau comme les protocoles IP ou ARP ou de la couche transport comme TCP ou UDP. Les couches plus hautes peuvent également être gérées par le noyau (tels que les protocoles applicatifs comme NFS). La gestion des systèmes de fichiers est un autre exemple de service pouvant être fourni par le noyau. Par exemple, sous Linux, le noyau fournit une vue virtuelle de tous les systèmes de fichiers, appelée *VFS (Virtual File System)*. En dessous de cette vue virtuelle, le noyau utilise le système de fichiers adapté pour l'opération. Le noyau est également chargé de la gestion des pilotes de périphériques.

Le problème des noyaux monolithiques est essentiellement leur taille ainsi que le partage de l'espace d'adressage par tous les services. Par exemple, le noyau Linux est composé de plus de 8 millions de lignes de code dont plus 4 de millions de lignes sont réparties entre les pilotes. Cette taille astronomique rend le noyau complexe et sa maintenance difficile. L'inclusion de très nombreux pilotes rend le noyau plus sensible à un pilote ayant potentiellement un fonctionnement anormal (problème de sûreté et de sécurité). Le problème des noyaux monolithiques est que le noyau ainsi que les autres services tels que les pilotes partagent le même espace d'adressage. Par conséquent, la faute d'un service peut affecter le bon fonctionnement et la sécurité des autres parties du noyau.

3.1.2.2 Micro-noyaux

Les micro-noyaux sont issus du constat que les noyaux monolithiques sont trop gros, trop complexes et trop sensibles aux fautes. Un micro-noyau considère que le rôle du noyau est d'offrir l'abstraction du matériel et la gestion transparente du partage des ressources. Cependant, les autres services doivent être exécutés dans un espace de mémoire séparé. Les avantages de cette architecture sont multiples. Premièrement, le noyau devient plus petit et sa maintenance plus aisée. Deuxièmement, le noyau est moins sensible à la faute d'un service. Enfin, après la faute d'un service, tel qu'un pilote de périphérique, il est possible de redémarrer ce service (bien que la récupération de l'état du service soit compliquée, voire impossible).

Pour fournir ces garanties, un système d'exploitation utilisant un micro-noyau est basé sur un noyau fournissant un ensemble minimal de fonctionnalités. Ces fonctionnalités comprennent la gestion des pages, la gestion de canaux de communication sûrs entre processus, ainsi éventuellement qu'un ordonnanceur. Chacun des autres services est vu comme un serveur avec son propre espace mémoire. La communication entre les serveurs est effectuée au moyen d'*IPC (Inter-Process Communication)*. Cette communication peut être synchrone ou asynchrone.

Les premiers micro-noyaux, tels que Mach [8], n'ont pas permis de résoudre les problèmes de complexité. Un des problèmes majeurs est que ces micro-noyaux sont génériques pour supporter plusieurs architectures. La généricité complique grandement le code du noyau, des interfaces et a également un impact sur la performance des applications. De plus, les micro-noyaux reposant essentiellement sur la communication entre processus, il est nécessaire que les IPC soient efficaces. Cependant, dans le noyau Mach, cette communication nécessite de faire des copies entre les différents espaces d'adressage et a par conséquent de mauvaises performances.

Une seconde génération de micro-noyaux, tels que L4 [66], a expérimenté les noyaux optimisés pour une architecture donnée. Le micro-noyau est écrit en prenant en considération les spécificités de l'architecture matérielle sur laquelle il va être déployé. Le couplage fort entre le noyau et l'architecture sous-jacente a permis de mettre en place des IPC et d'améliorer ainsi les performances globales du noyau. Le principe de la solution est de coupler les changements de contextes avec les IPC. Cette synergie permet, par exemple, de faire passer une petite quantité de données sans copie entre l'appelant et le serveur appelé. Le passage de données va s'effectuer en utilisant les registres. L'appelant doit copier les données concernées dans les registres et le noyau effectue un changement de contexte en faveur du serveur appelé qui pourra récupérer directement la valeur des données dans les registres.

3.1.2.3 Exo-noyaux

L'idée des exo-noyaux (*exokernel* [39, 40]) est de pousser la simplification du noyau au maximum. Comme nous l'avons vu, dans les noyaux classiques, le noyau est responsable de (i) multiplexer les ressources entre les applications et de (ii) fournir une abstraction des ressources. Cependant, fournir une abstraction nécessite une grande quantité de code, ce qui accroît la complexité du noyau. La

gestion des abstractions réduit à la fois la sûreté du noyau, ainsi que sa maintenance. De plus, définir des abstractions génériques dégrade généralement les performances. Enfin, il est complexe, voire impossible, de fournir des abstractions génériques correspondant à tous les cas.

Un exo-noyau ne s'occupe donc que du multiplexage sécurisé des ressources pour les applications. Un exo-noyau n'essaie pas de masquer la complexité des ressources aux bibliothèques de niveau utilisateur. Au contraire, l'exo-noyau va affecter une partie des ressources à chaque processus. Les ressources sont notamment les interruptions, les blocs de disque ou encore les quanta de temps. L'exo-noyau est chargé de gérer les privilèges d'accès d'un processus à une ressource. Il est également chargé de la gestion du TLB. Enfin, le noyau offre un support minimal pour la communication entre processus.

Il est important de noter que les auteurs d'*exokernel* encouragent le développement de couches d'abstractions mais au niveau utilisateur. Contrairement à un micro-noyau, un système d'exploitation utilisant un exo-noyau ne se base pas sur des services proposés par un ensemble de serveurs, mais sur des bibliothèques de niveau utilisateur.

3.1.3 Systèmes d'exploitations pour les architectures multi-cœurs

Sur les systèmes mono-cœur, un noyau complètement préemptible doit être capable de gérer les accès concurrents aux ressources partagées puisque l'exécution du noyau peut être interrompue à chaque instant. Comme nous l'avons déjà vu dans le chapitre 2, l'arrivée des architectures multi-cœurs aggrave encore les difficultés puisque le noyau doit faire face à du parallélisme réel. Les difficultés engendrées par le parallélisme réel sont multiples :

- *Gestion des accès concurrents à des données partagées.* Tout comme dans un noyau mono-cœur, il est nécessaire de gérer correctement les accès aux structures de données internes du noyau. Par exemple, si deux threads d'un même processus demandent simultanément l'allocation d'une socket pour gérer une connexion TCP, il est impératif qu'ils n'obtiennent pas le même descripteur de fichier. Le parallélisme réel engendre plus souvent des problèmes d'accès concurrents puisque sur un système mono-cœur, cela n'arrive qu'aux frontières des changements de contextes. La prise de verrou a un impact très négatif en termes de performances, car si plusieurs cœurs sont bloqués, tous ces cœurs ne feront pas de progrès. Enfin, si les cœurs partagent des données, à chaque mise à jour d'une donnée, il faut déclencher le protocole de cohérence de cache, ce qui réduit également les performances.

- *Gestion des interruptions.* Contrairement à un système mono-cœur, une interruption d'un périphérique doit être affectée à un cœur. Grâce à l'*IO-APIC (I/O Advanced Programmable Interrupt Controller)*, il est possible de fixer l'association entre cœurs et lignes d'interruption ou de laisser l'*IO-APIC* équilibrer la charge d'interruption sur les cœurs. Il est important d'essayer de faire gérer l'interruption par le cœur qui va ensuite manipuler les données, pour maximiser la localité et l'utilisation des caches.

- *Allocation mémoire.* Sur les architectures NUMA, il est intéressant de maximiser la localité des données. Pour cela, lors d'une faute de page, une solution efficace est d'allouer les données dans la zone mémoire de celui qui a déclenché la faute de page. Ainsi, la donnée sera au plus près du flot d'exécution demandeur.

- *Allocation de ressources processeurs aux applications.* Précédemment, le noyau se basait uniquement sur des notions de partage de temps (*time-sharing*) pour affecter des ressources aux différents threads. Avec l'apparition des architectures parallèles, il devient nécessaire également de faire du partage de cœurs entre les threads (*space-sharing*).

Dans la suite de cette section, nous allons étudier les récents travaux qui ont eu lieu dans le domaine des systèmes d'exploitation pour les architectures multi-cœurs. La figure 3.2 résume les différentes architectures possibles pour les systèmes d'exploitation. Cette figure est inspirée de celle présentée dans les travaux sur le système d'exploitation Barrelfish [18]. Le noyau Linux est l'un des plus utilisés sur les machines multi-processeurs. Depuis sa version 2.6, Linux fournit un support efficace des nouvelles architectures multi-cœurs grâce notamment à l'utilisation de verrous à grain fin. Linux gère également les architectures NUMA en utilisant la technique d'allocation de pages précédemment décrite. Cependant, différents problèmes subsistent pour une utilisation efficace des machines massivement multi-cœurs. Les solutions que nous présentons dans la suite de cette section empruntent des techniques venues du domaine des systèmes d'exploitation distribués.

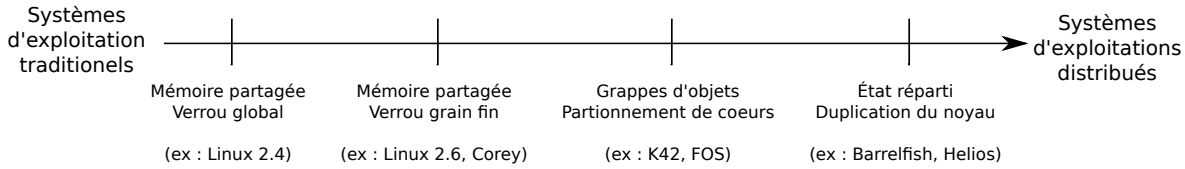


Figure 3.2 – Possibilités d'architecture pour les systèmes d'exploitation multi-cœurs.

3.1.3.1 Corey

Corey [100] est un système d'exploitation pour les architectures multi-cœurs. Tout comme les exo-noyaux, Corey propose de redéfinir les abstractions fournies au programmeur d'application pour pouvoir améliorer les performances. L'objectif de Corey n'est pas de changer l'architecture du noyau pour améliorer son passage à l'échelle sur les architectures multi-cœurs, mais de proposer au programmeur un ensemble de primitives pour lui permettre de gérer finement le partage entre les threads d'une même application. En effet, les concepteurs de Corey ont fait trois constats :

- *Les applications ne peuvent pas gérer explicitement le partage de mémoire entre leurs threads.* Dans les systèmes d'exploitation actuels, l'intégralité de la mémoire virtuelle est partagée entre les threads d'un même processus. Cette obligation engendre deux problèmes principaux. Premièrement, les threads d'un même processus manipulent la même table des pages. Par conséquent, si plusieurs de ces threads font une faute de page simultanément sur deux cœurs différents, ils vont entrer en contention pour obtenir une nouvelle page mémoire. Deuxièmement, lors d'une éviction d'une association du TLB à la suite de la libération d'une zone mémoire, il est impératif de vider les TLB des autres cœurs qui possédaient l'entrée. Cette nécessité réduit l'efficacité du TLB. Pour ne pas partager de mémoire par défaut, une solution consiste à utiliser des processus à la place des threads, puis à créer une zone de mémoire commune grâce à la fonction *mmap*. Cependant, comme il y a une table des pages par processus, chaque accès à la zone commune provoque une faute de page sur chacun des processus pour remplir sa table des pages, multipliant ainsi les traitements et réduisant la performance.

- *Les threads d'une même application partagent automatiquement les mêmes ressources.* C'est par exemple le cas de la table des descripteurs de fichiers. Lors de l'allocation d'un nouveau descripteur de fichier, il est nécessaire de verrouiller l'accès à la table, réduisant ainsi le passage à l'échelle de cette fonction.

- *Tous les cœurs font des traitements symétriques.* Cependant, ce type d'exécution nécessite de prendre des verrous sur les données globales, puisque chaque cœur peut faire le même traite-

ment. De plus, il peut être nécessaire de rapatrier les données depuis le cache d'un autre cœur, déclenchant ainsi le protocole de cohérence de cache.

Pour résoudre ces problèmes, Corey propose de redéfinir les interfaces que le noyau exporte à l'application. Corey propose notamment de permettre à l'application d'expliciter le partage des ressources. Pour chaque point décrit ci-dessus, les auteurs de Corey fournissent une nouvelle abstraction.

- *Plages d'adresses.* Chaque thread déclare explicitement dans quelle plage d'adresses (*address range*) la donnée doit être placée. Il est possible de créer des plages d'adresses qui sont privées au thread ou des plages d'adresses qui sont partagées entre les threads. Concrètement, cela nécessite une gestion des tables des pages à deux niveaux : la table des pages de niveau 0 qui contient les associations entre adresses virtuelles et adresses physiques pour les plages d'adresses partagées et celle de niveau 1 qui contient les liens pour les plages d'adresses privées. Notons qu'il y a une table des pages de niveau 1 par thread. L'avantage de cette solution est d'éviter la contention pour les données non partagées, tout en ne conservant qu'une faute par page sur les données partagées.

- *Zones de partage.* L'idée est de permettre au programmeur de définir des descripteurs de fichiers privés à un thread. Tout comme pour les plages d'adresses, il existe donc les tables des descripteurs de fichiers privés aux threads et la table des descripteurs de fichiers partagés. L'avantage de ces zones de partage (*shares*) est de ne pas avoir de contention lors de la gestion d'un descripteur de fichier s'il est traité par le thread qui l'a créé.

- *Cœurs dédiés.* L'idée de cette solution est de dédier des cœurs (*kernel cores*) à la gestion de certaines tâches du système d'exploitation pour réduire la contention sur les données manipulées et améliorer la localité de ces données dans le cache des cœurs. Par exemple, il est possible de dédier un cœur à la gestion des interfaces réseau pour éviter la contention dans le pilote de ces cartes.

3.1.3.2 K42 et Tornado

K42 [58] est un système d'exploitation basé sur Tornado [44]. L'objectif est de construire un noyau qui soit modulaire, adaptable et ayant un bon passage à l'échelle sur les architectures multi-processeurs (en supportant une grande variété d'architectures). La structure de K42 est orientée objet, c'est-à-dire que chacun de ses composants est un objet avec ses données et ses fonctions. L'avantage de la programmation orientée objet se situe essentiellement dans l'adaptabilité et la modularité. K42 fournit un support de l'ABI et de l'API de Linux, ce qui permet de supporter les applications développées pour ce système. Tornado et K42 ont apporté différentes améliorations pour supporter les architectures multi-processeurs de manière efficace.

Architecture du système d'exploitation. Le système d'exploitation est basé sur un exo-noyau et des serveurs. La partie noyau est chargée de fournir notamment des primitives de communication efficaces entre les serveurs. Ces primitives sont appelées *PPC (Protected Procedure Call)* et sont des appels synchrones à des fonctions fournies par des serveurs tournants dans un autre espace d'adressage. Pour réduire les coûts, la communication est effectuée si possible en utilisant les registres. Pour cela, l'appel doit être effectué sur le même cœur que l'appelant. K42 propose également un mécanisme d'appel de méthodes asynchrone. Dans ce cas, les appels peuvent être effectués sur d'autres cœurs. Les serveurs sont, comme tout le reste du système d'exploitation, des objets et fournissent des services (système de fichiers, allocation mémoire, etc.).

Objets en grappe. Pour résister à la montée en charge, il est possible de définir des grappes d'objets *(clustered objects)*. Le serveur est un objet proposant une interface. Cependant, cet objet peut être dupliqué et réparti sur plusieurs cœurs pour résister à la charge. La cohérence entre ces multiples copies peut être assurée au besoin par les mécanismes de communication asynchrones décrits précédemment.

Autres propositions. En plus de l'organisation en objets et de la réplication de ces objets, K42 propose différents mécanismes pour améliorer le passage à l'échelle du système. Premièrement, K42 se base sur un gestionnaire de mémoire efficace sur les architectures multi-cœurs et adapté aux architectures NUMA. Nous verrons dans la section 3.2.3 les détails des allocateurs mémoire efficaces. Pour éviter la contention sur le système de fichiers, K42 propose un système de fichiers (*KFS*) utilisant des verrous à grain fin. Enfin, il est possible d'observer de manière efficace le comportement d'une application en utilisant les compteurs de performance fournis par le processeur. La collecte des traces est effectuée par processeur pour réduire au minimum l'impact sur les performances lorsque le système possède un grand nombre de processeurs.

3.1.3.3 Fos, un système d'exploitation pour processeurs massivement multi-cœurs.

Fos [99] (*Factored Operating System*) est un système d'exploitation qui vise les architectures massivement multi-cœurs. Les auteurs parlent d'architectures à plus d'un millier de cœurs. Pour ces architectures, un système d'exploitation classique ne permettra pas un passage à l'échelle efficace. En effet, comme nous l'avons vu précédemment, plusieurs problèmes apparaissent dont les principaux sont la difficulté de prendre des verrous à grain très fin, ainsi que la probable suppression du mécanisme de cohérence de cache dans les processeurs.

Fos est basé sur du partitionnement dans l'espace plutôt que sur du partage de temps comme les systèmes d'exploitation traditionnels. Ce choix est raisonnable avec un grand nombre de cœurs, car il est possible d'avoir un ou plusieurs cœurs par application. Fos est basé sur un micro-noyau et un ensemble de serveurs. L'architecture de Fos est donc très proche de celle de K42, excepté que le système d'exploitation n'est pas orienté objet et ne nécessite pas obligatoirement de mémoire partagée.

Micro-noyau. Le micro-noyau a pour rôle de fournir les mécanismes nécessaires à la communication entre les serveurs, c'est-à-dire un service de noms et un mécanisme de communication basé sur du passage de messages. Il y a un micro-noyau par cœur. La communication entre les serveurs est effectuée de manière asynchrone par passage de messages. Les arguments sont passés par copie. Pour faire du passage de message efficace, il est possible d'utiliser les mécanismes qui seront peut-être présents dans les prochains processeurs.

Serveurs. Fos intègre dans ses serveurs des mécanismes venus du domaine des systèmes répartis à grande échelle. Les serveurs fournissent un ensemble de services aux applications. Par exemple, lors de la manipulation de fichiers, l'application va contacter puis dialoguer avec le serveur gérant le système de fichiers. Ce serveur va lui même éventuellement interagir avec le serveur gérant les accès physiques au disque dur. Un serveur s'exécute sur un cœur et il n'y a qu'un seul serveur par cœur. Un serveur peut également se situer sur une machine distante [36]. Tout comme dans les systèmes répartis traditionnels, les serveurs peuvent être répliqués pour mieux résister à la charge. Dans ce cas, l'ensemble des serveurs fournissant un même service est appelé une *flotte* de serveurs. Lorsqu'un client requiert un service, il va contacter le serveur le plus proche parmi la flotte de serveurs. Pour

cela, le micro-noyau fournit un service de nommage. Il est possible de faire grandir ou rétrécir dynamiquement la flotte de serveurs associée à un service en fonction de la demande. Pour simplifier la gestion de la flotte, les serveurs sont sans état, c'est-à-dire qu'ils ne maintiennent pas d'état entre deux requêtes. Toutes les informations nécessaires au traitement de la demande sont donc passées en paramètre de la requête. Au sein d'un serveur, le modèle d'exécution est événementiel. Si une prise de verrou est nécessaire pour assurer l'exclusion mutuelle sur une ressource, alors un service de verrou est utilisé.

L'architecture de Fos a été conçue pour fournir un passage à l'échelle idéal sur les architectures massivement multi-cœurs. Cependant, le passage à l'échelle sur de telles architectures n'a pas été étudié et les surcoûts liés à Fos sur les architectures actuelles sont importants (en 10 et 40 fois plus lents que Linux). La validité de l'architecture reste donc à démontrer.

3.1.3.4 Barrelfish

Alors que les systèmes d'exploitation précédemment cités étudient le passage à l'échelle sur les architectures multi-cœurs soit par le raffinage des verrous ou des interfaces fournies, soit par la duplication et le partitionnement des services, Barrelfish [19, 18] explore l'extrémité du spectre en choisissant de dupliquer le noyau du système d'exploitation sur chacun des cœurs. Le système d'exploitation est vu réellement comme un système distribué et les algorithmes employés sont également tirés du domaine des systèmes distribués. Par rapport au système d'exploitation Fos, la principale différence est que sur un cœur, le noyau gère l'intégralité des services, alors que dans Fos chaque cœur gère un service spécifique.

Noyau. Le noyau (appelé *CPU driver*) tournant sur chacun des cœurs est similaire à un exo-noyau. Comme nous l'avons vu dans la partie précédente, cela veut dire que le noyau ne fournit pas d'abstraction aux couches supérieures. Dans Barrelfish, le noyau fournit des canaux de communication simples entre les cœurs. Au-dessus de ces canaux pourront être implantés des mécanismes de communication plus élaborés. La communication entre les cœurs est effectuée de manière asynchrone en utilisant des messages. Dans l'implantation actuelle, le mécanisme de passage de messages est effectué en utilisant la mémoire partagée. Il est envisagé d'utiliser le passage de messages natif si les processeurs le proposent dans le futur. Pour des raisons de performances, le mécanisme de passage de messages est effectué différemment pour un appel local ou pour un appel distant (c'est-à-dire sur un autre cœur). Cependant, cela est transparent pour l'application. Notons que le noyau est lié à une architecture spécifique. La structuration du système d'exploitation en un noyau par cœur permet également l'utilisation des architectures hétérogènes. Pour le moment, cette possibilité n'a pas été exploitée. Il n'est également pas précisé la manière dont une application sera compilée pour une ou plusieurs architectures.

Moniteurs. Les moniteurs sont une couche située au-dessus de l'exo-noyau. Les moniteurs sont dupliqués sur chacun des cœurs et sont responsables notamment de l'allocation mémoire ou encore de l'ordonnancement. Comme l'état des moniteurs est répliqué entre les cœurs, il est nécessaire d'utiliser un protocole de consensus entre ces cœurs. Par exemple, pour allouer une nouvelle région de la mémoire, le moniteur doit d'abord acquérir une capacité associée à une partie de la mémoire pour mettre à jour sa table des pages.

Applications. Selon le modèle de Barrelfish, une application peut être distribuée sur plusieurs cœurs (éventuellement de types différents). Une application peut être développée en ayant conscience de sa

répartition ou en se basant sur un mécanisme transparent de cohérence de cache logicielle implantée avec des messages.

3.1.3.5 Helios

Helios [73] est un système d'exploitation qui a pour but de permettre l'utilisation efficace des architectures hétérogènes. Le constat réalisé est que de plus en plus de périphériques permettent de faire tourner des applications (cartes programmables, générations futures de GPU, etc.). L'avantage de faire tourner l'application qui traite les données au plus près de la carte est d'éviter des communications coûteuses et d'améliorer la localité. Helios est basé sur la machine virtuelle Singularity [41]. Il y a deux différences majeures par rapport au système d'exploitation Barrelfish précédemment cité. Premièrement, bien qu'il y ait un noyau par carte programmable, il n'y a pas forcément un noyau par cœur. Deuxièmement, l'utilisation d'un langage compilé pour une machine virtuelle rend l'utilisation des architectures hétérogènes plus transparente au programmeur.

Noyau. Helios est construit autour d'un noyau principal (*coordinator kernel*) et de noyaux satellites (*satellite kernels*). Il y a un noyau satellite par carte programmable. Dans les systèmes NUMA, il y a un noyau satellite par nœud. Les noyaux satellites peuvent être supportés sur une carte programmable si celle-ci fournit au moins (i) un minuteur, (ii) un gestionnaire d'interruptions, et (iii) un gestionnaire d'exception. Les noyaux satellites, à la manière des exo-noyaux ne sont responsables que de la gestion des quanta de temps et de la mémoire. La communication entre les noyaux est effectuée par passage de messages. Comme pour le noyau Barrelfish, Helios distingue l'envoi de messages locaux de l'envoi de messages distants pour améliorer les performances du mécanisme de passage de messages. Le noyau principal est responsable du gestionnaire de noms.

Indépendance vis-à-vis de l'architecture. Un des intérêts d'Helios est de se baser sur la machine virtuelle Singularity. Le programmeur est ainsi capable d'exécuter ses applications sur Helios sans aucun changement. La programmation est donc indépendante de l'architecture et Helios va exécuter du code intermédiaire (*byte-code*). Le programmeur va cependant indiquer les affinités d'un processus vis-à-vis d'une architecture ou d'un autre processus. Par exemple, il est possible d'indiquer que le pilote gérant les interfaces réseau soit placé sur celles-ci, si elles satisfont les trois critères précédemment énoncés. Il est également possible de définir des affinités négatives, c'est-à-dire qu'un processus ne souhaite pas être colocalisé avec un autre processus ou être placé sur une architecture donnée. Si les affinités ne sont pas spécifiées, Helios essaie d'équilibrer la charge sur les processeurs.

3.2 Repenser l'exécution des tâches sur les architectures multi-cœurs

Dans la section précédente, nous avons décrit l'architecture des systèmes d'exploitation et avons étudié les évolutions proposées pour faire face à l'apparition des architectures multi-cœurs. Le système d'exploitation est responsable de l'affectation des ressources processeurs aux applications. Nous avons vu que cette affection peut être soit du temporel soit spatial soit une combinaison des deux. Nous avons vu dans la section 3.1.2 qu'en fonction de l'architecture choisie pour le noyau, l'ordonnancement des tâches sur les processeurs est effectué soit par le noyau du système d'exploitation, soit par un environnement d'exécution de niveau applicatif. Dans le premier cas, le noyau est responsable d'ordonnancer les tâches à la fois dans le temps et dans l'espace. Dans le second cas, le noyau est uniquement chargé d'affecter un ensemble de ressources à cet environnement d'exécution et de l'informer lorsque ces ressources changent [12].

Qu'il soit de niveau utilisateur ou de niveau noyau, un ordonnanceur doit considérer un certain nombre de paramètres avant d'affecter les tâches aux cœurs disponibles. Nous ne parlerons pas ici des problématiques liées à l'ordonnancement sur un cœur (équité, famine, ordonnancement en fonction des ressources disponibles, etc.), mais des problématiques spécifiques aux architectures multi-cœurs. Ces questions concernent essentiellement le choix du cœur qui va traiter une tâche donnée. Ce choix est effectué en fonction de deux critères principaux :

- *La répartition de charge.* L'idéal pour les applications multi-cœurs est de ne jamais laisser un cœur inactif. L'ordonnanceur est ainsi responsable de garantir que tous les cœurs seront utilisés autant que possible. Il est important de considérer que l'ordonnanceur ne peut charger efficacement les cœurs que s'il dispose de suffisamment de tâches fournies directement ou indirectement (e.g. traitement des interruptions) par les applications. Une bonne utilisation des architectures multi-cœurs relève donc d'une responsabilité commune entre l'application et l'ordonnanceur.

- *La gestion des affinités.* Utiliser un maximum les processeurs disponibles est une condition nécessaire pour obtenir un bon passage à l'échelle. Cependant, il est également nécessaire de minimiser les surcoûts en évitant par exemple au maximum de colocaliser des tâches qui peuvent avoir un effet néfaste l'une sur l'autre ou encore de minimiser les communications entre les cœurs.

En plus de l'ordonnancement, l'allocation mémoire au niveau applicatif, c'est-à-dire le découpage de la mémoire virtuelle affectée aux threads d'un même processus, est une opération nécessitant un soin tout particulier pour éviter au maximum la prise de verrous, ainsi que le faux partage.

Dans la suite de cette section, nous allons tout d'abord étudier les améliorations proposées pour optimiser la répartition de charge, ainsi que la gestion des affinités. Nous allons ensuite présenter différents mécanismes d'allocation mémoire conçus pour fournir un bon passage à l'échelle. Enfin, nous décrirons brièvement deux environnements d'exécution optimisés pour les architectures multi-cœurs.

3.2.1 Équilibrage de charge

L'ordonnanceur est responsable du partage efficace des ressources entre les différents threads. L'apparition des architectures multi-cœurs nécessite de réaliser une répartition spatiale des tâches en plus de la classique répartition temporelle. Pour obtenir une utilisation maximale de ces architectures, il est crucial de répartir uniformément la charge entre les cœurs. Les tâches à exécuter, c'est-à-dire les threads ou les événements à traiter, sont stockées généralement dans des files d'attente de tâches prêtes. La figure 3.3 présente deux organisations possibles classiquement utilisées dans les ordonnanceurs de threads ou d'événements.

Sur cette figure, nous pouvons voir que l'organisation des tâches peut s'effectuer soit en utilisant une file globale soit en utilisant une file par cœur. Nous ne représentons ici que les files de tâches actives. Le même choix se pose pour les files de tâches bloquées (par exemple en attente d'une E/S). L'avantage d'utiliser une file globale est qu'il n'y a pas besoin d'effectuer d'équilibrage de charge entre les cœurs, puisque tous les cœurs peuvent ordonnancer toutes les tâches. Cependant, pour ajouter ou pour enlever une tâche dans une file, il est nécessaire de prendre un verrou pour assurer la cohérence de cette file. Cette prise de verrou est un frein au passage à l'échelle du noyau ou de l'application. Ce mécanisme utilisant une file globale a été utilisé par exemple dans la version 2.4 du noyau Linux. Une autre solution est d'utiliser une file de tâches par cœur. Dans ce cas, l'ordonnanceur peut proposer un passage à l'échelle idéal. Cependant, il est nécessaire de gérer la répartition des

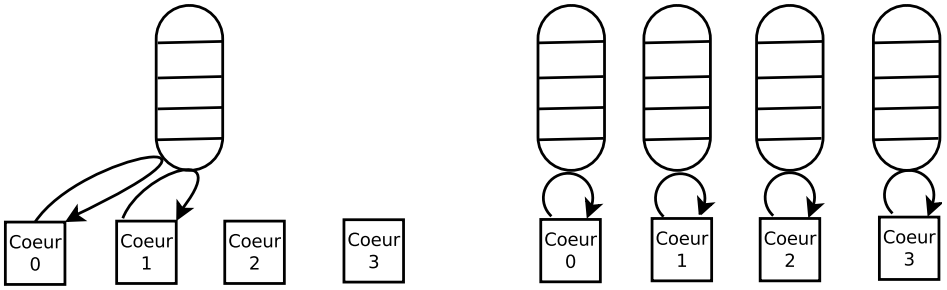


Figure 3.3 – Deux organisations typiques en files d'un ordonnanceur de tâches.

tâches sur les cœurs. Comme nous l'expliquons par la suite, le mécanisme de répartition des tâches peut entraîner des déséquilibres de charge entre les cœurs, ce qui réduit les performances. Ce type d'organisation est utilisée dans la version 2.6 du noyau Linux et est celui généralement mis en œuvre dans les supports d'exécution multi-cœurs. Notons que la structure des files de tâches peut être plus complexe que celles représentées sur la figure 3.3. Par exemple, il est possible d'avoir des files hiérarchiques, structurées par priorité. Cependant, la structure de la file n'a pas un impact sur le passage à l'échelle de l'ordonnanceur, mais sur ses performances.

Un déséquilibre de charge vient essentiellement du fait que les tâches sont initialement placées arbitrairement sur les cœurs. Cependant, les durées de vie des tâches ne sont pas équivalentes. Certaines tâches sont très longues, alors que d'autres, au contraire, ont une durée de vie très courte. Par conséquent, un cœur aura peut-être plus de tâches courtes qu'un autre cœur, se retrouvant ainsi plus souvent en situation d'attente. Pour répartir les tâches sur les cœurs disponibles, plusieurs solutions existent. La première solution consiste à avoir un gestionnaire qui est exécuté périodiquement sur un cœur et dont le rôle est d'équilibrer la charge entre les différents cœurs. Le gestionnaire est responsable de déterminer quels sont les cœurs surchargés et ceux sous-chargés et de migrer une partie des tâches des premiers vers les deuxièmes. C'est le fonctionnement choisi pour le support d'exécution Click-SMP, décrit dans la section 2.2.1.

Une autre solution pour équilibrer la charge sur les cœurs consiste à faire du vol de tâches. Ce mécanisme est utilisé notamment dans le support d'exécution Cilk [22, 24] dont le modèle de programmation est décrit dans la section 2.2.1, mais également par d'autres supports d'exécution tels que Libasync-smp [102] ou encore dans le noyau Linux 2.6. Le principe du vol de tâches est de décentraliser la gestion de la répartition des tâches sur chacun des cœurs actifs. Lorsqu'un cœur est en sous-charge, il va alors voler du travail chez un autre cœur. Le choix du cœur *victime* est généralement effectué aléatoirement pour éviter une trop grosse contention sur un même cœur. L'algorithme initial consiste à prendre un verrou sur le coeur victime pour garantir une manipulation correcte de la file de celui-ci. Chaque cœur prend également un verrou lorsqu'il récupère une tâche dans sa file. Chase et Lev [30] ont montré dans leurs travaux qu'il est possible dans le cas de Cilk de réaliser un vol de tâche sans prise de verrou sur la victime. Le principe est d'utiliser des files particulières, appelées *DEqueue*. Ces files ont une taille dynamique. La production d'une nouvelle tâche entraîne un ajout en tête de la file. La récupération de la prochaine tâche à exécuter est également effectuée en tête de file. Enfin, le vol est effectué en queue de la file. Le point le plus important est qu'une tâche est toujours ajoutée à la file du cœur qui l'a produite. Par conséquent, il n'y a toujours qu'un seul thread qui manipule la tête de la file donc il n'est pas nécessaire de prendre un verrou. Lorsque deux cœurs peuvent manipuler la même entrée de la file, par exemple lorsque deux cœurs volent en même temps ou lorsqu'il n'y a plus qu'un élément dans la file, la manipulation est effectuée au moyen d'une opération atomique. Il est important de considérer que ce mécanisme n'est pas applicable si une tâche peut être produite sur un autre cœur (ce qui est le cas par exemple de Libasync-smp).

Que ce soit périodiquement ou au moyen d'un mécanisme de vol de tâches, le déplacement d'une tâche d'un cœur à un autre est très coûteux. Le déplacement implique de devoir recharger les données dans un autre cache et ainsi de ne pas bénéficier d'un cache *chaud*. Cette constatation est à nuancer avec les architectures multi-cœurs dans lesquelles les caches peuvent être partagés. Le coût logiciel de la migration de thread peut également être un problème en fonction de la granularité de la tâche déplacée. Strong *et al.* [88] ont étudié la possibilité de migrer efficacement un thread d'un cœur à un autre. Ils proposent notamment plusieurs idées telles que faire de l'attente active à la place de l'utilisation d'interruptions ou encore des améliorations permettant de réduire le nombre de changements de contexte nécessaires à la migration.

3.2.2 Affinités et compétition

En plus de la notion de charge équilibrée, il est nécessaire de prendre en compte les affinités entre les processus (ou les threads) lors de leur affectation sur les cœurs. Il est possible de placer deux tâches sur le même cœur, sur deux cœurs d'un même processeur partageant un cache, sur deux cœurs d'un même processeur ne partageant pas de caches ou sur deux cœurs de deux processeurs distincts (distants d'un ou plusieurs sauts).

La colocalisation des threads ou des processus sur un même cœur peut être une source importante d'amélioration de performances ou inversement de ralentissement. Il convient de distinguer deux types de colocalisation : la colocalisation de threads et la colocalisation de processus. Dans le premier cas, comme les threads peuvent accéder à des données partagées, le placement devra tenir compte du partage éventuel de mémoire entre deux threads. Inversement, le choix du positionnement de deux processus ne tiendra pas compte d'un espace mémoire partagé, mais plutôt de la contention d'accès aux ressources.

Placement des threads sur les cœurs. Le placement des threads sur les cœurs disponibles est effectué en considérant l'équilibrage de charge, mais également les affinités entre les threads. Ces affinités sont liées au partage de mémoire entre les threads d'un même processus. Tam *et al.* [91] ont montré qu'il est possible d'utiliser les informations remontées par le processeur, c'est-à-dire les compteurs de performance *(HPC, Hardware Performance Counters)*, pour détecter le partage de mémoire entre les threads. L'idée est d'intégrer cette détection au support d'exécution pour éviter au programmeur de spécifier manuellement quelles sont les affinités entre les threads. En effet, le concepteur de l'application n'est pas forcément conscient du partage entre les threads, notamment si ceux-ci utilisent des bibliothèques tierces. Leur solution s'effectue en quatre étapes. La première étape est de vérifier que des accès aux caches distants sont effectués et que ces accès ont un impact sur les performances. Si les accès aux caches distants ont un impact sur les performances, alors la deuxième étape est de construire la carte des zones de mémoire partagées *(shMaps)*. Cette carte est construite en utilisant les spécificités des compteurs de l'architecture *Power* qui permettent de connaître l'adresse de la dernière donnée ayant engendré une faute de cache en L1. En couplant cette information avec les compteurs de performances indiquant le nombre de fautes de caches ayant entraîné un rapatriement de données depuis un cache distant, il est possible d'avoir une approximation fiable des adresses partagées entre les threads. Notons qu'avec ce mécanisme, il n'est pas possible de déterminer une affinité entre deux threads placés sur des cœurs partageant un cache. Pour réduire les coûts, seulement une partie de la mémoire virtuelle est observée (échantillonnage spatial) tous les N accès (échantillonnage temporel). À partir de ces informations, le support d'exécution réalise des groupes de threads si le partage entre ces threads est supérieur à une valeur définie. Il répartit ensuite ces groupes sur les cœurs disponibles. Cette évaluation est effectuée périodiquement.

Une autre solution présentée est de repenser l'ordonnancement des threads. Dans ce cas, on ne

place pas les threads en fonction de la charge, mais en fonction des ressources manipulées [25]. Alors que dans un système traditionnel les données sont déplacées entre les caches des cœurs en fonction de leur accès, leur solution consiste à migrer le thread voulant accéder à une donnée sur le cœur où elle est présente. Par exemple, dans un serveur Web, le thread sera placé en fonction du dossier accédé, ce qui permet de faire tenir le parcours de ces dossiers en cache. Cependant, à notre connaissance, les idées présentées dans ce papier n'ont pas donné lieu à une évaluation complète. De plus, si cette méthode d'ordonnancement peut être mise en œuvre dans un serveur Web, sa validité sur d'autres applications reste à démontrer. Enfin, les coûts de migration des threads peuvent également être importants.

Placement des processus sur les cœurs. Contrairement à la problématique du placement des threads sur les cœurs disponibles, il n'est pas intéressant de prendre en compte un éventuel partage de mémoire puisque, par défaut, les processus manipulent des espaces mémoire différents. Dans le placement des processus, outre la répartition de charge, le paramètre essentiel est la contention sur les ressources partagées. Le principal problème est le partage d'un ou plusieurs caches. Deux processus consommant une partie importante du cache, s'ils sont colocalisés sur deux cœurs partageant un cache, vont probablement chacun expulser les données de l'autre, réduisant ainsi grandement l'efficacité des caches. Différents travaux se sont attachés à classifier et à ordonnancer efficacement les processus. Il est important de noter que les placements étudiés par la suite font deux suppositions principales : (i) il n'y a qu'un seul processus par cœur et un seul thread par processus, et (ii) il n'y a qu'un seul cache partagé entre les cœurs. Ce cache partagé est le cache de plus haut niveau *(LLC, Last Level Cache)*.

Chandra *et al.* [29] ont présenté trois modèles permettant de prédire l'impact de la colocalisation de deux processus sur deux cœurs partageant un cache. Ces modèles sont basés sur l'écart entre deux accès à une même donnée (*stack distance*) ainsi que sur l'associativité du cache partagé. En utilisant ces données, une courbe est créée modélisant le nombre d'accès en fonction de l'écart entre deux accès, ce qui donne le profil de l'application. À partir de ces données, trois modèles sont présentés : *FOA (Frequency Of Access), SDC (Stack Distance Competition)* et un modèle probabiliste. Le modèle probabiliste est celui donnant les résultats les plus précis. Cependant, ce modèle est compliqué à mettre en œuvre.

Jiang *et al.* [54] proposent une solution permettant d'évaluer le placement optimal des processus. La solution est trouvée en créant des graphes de tâches, les arcs étant annotés avec la dégradation de performance si les deux tâches sont ordonnancées sur des cœurs partageant un cache. Notons qu'un des prérequis de l'algorithme est donc de connaître la dégradation de chaque couple de processus. Jiang *et al.* montrent que l'algorithme de recherche du placement idéal est polynomial si deux cœurs au maximum partagent un LLC et *NP-complet* si plus de deux cœurs partagent un LLC. Dans ce deuxième cas, ils proposent également une approximation permettant de fournir des résultats proches de l'optimal en conservant un temps polynomial.

En utilisant la méthode de Jiang *et al.*, il est possible de fournir des algorithmes d'ordonnancement s'approchant de l'idéal. Différentes solutions ont été proposées. La première est la caractérisation des programmes en classes [101]. Les classes sont construites en utilisant à la fois le taux de fautes de cache de l'application ainsi que le profil d'accès au cache. À partir de ces classes, il est possible de déterminer que le processus a ne doit pas être colocalisé avec le processus b. Une autre solution est d'utiliser les taux de fautes du cache de dernier niveau lorsque l'application tourne seule [57]. Dans ce cas, l'ordonnanceur place les processus avec le taux de fautes le plus important sur des cœurs ne partageant pas de caches. Comme décrit par Zhuravlev *et al.* [104], les avantages de cette méthode sont (i) la possibilité de mise en œuvre par un ordonnanceur en récupérant les mesures à l'exécution, ce qui avec leurs applications donne de bons résultats, et (ii) la prise en compte de la contention sur les autres composants (bus mémoire, contrôleur mémoire, et mécanisme de *prefetch*), car une faute

de cache entraîne forcement une requête à ces composants.

Cas des architectures hétérogènes. En plus de l'affinité entre les threads, les architectures hétérogènes nécessitent de prendre en compte les affinités entre les threads et les cœurs (ou les autres composants programmables).

Knauerhase *et al.* [57] proposent une méthode pour ordonnancer les threads sur des cœurs ayant des fonctionnalités différentes. Cette méthode est basée sur l'essai et l'apprentissage. Le thread est ordonnancé sur un cœur. Si ce cœur génère une exception à cause d'une fonctionnalité manquante, le thread est déplacé sur un autre cœur. Si le thread génère plusieurs fois une exception pour un cœur donné, le thread ne sera plus ordonnancé sur ce cœur.

Shelepov *et al.* [84] étudient le cas des architectures asymétriques non pas en terme de fonctionnalité, mais en terme de performances (tout particulièrement en fréquence). L'idée est de générer un profil des threads pour déterminer si un thread est souvent en attente de la mémoire. Ce profil est généré grâce à une observation du comportement des threads effectuée sur une première exécution. Lorsqu'un thread est créé, il est assigné à un processeur en fonction de son profil et des capacités du cœur. Par exemple, il est plus intéressant de placer un thread effectuant beaucoup d'accès mémoire sur un cœur avec une fréquence moindre puisqu'il n'est pas capable de tirer parti d'une fréquence plus importante en étant en attente de la mémoire. La distribution des threads est réévaluée périodiquement.

3.2.3 Allocation mémoire efficace

Faux partage. Un des premiers problèmes exposés au concepteur d'applications multi-cœurs est le problème du faux partage. Ce problème a été décrit dans la section 1.3.3 et provient du fait que deux données disjointes manipulées par deux threads séparés déclenchent inutilement le mécanisme de cohérence de cache, car elles partagent la même ligne de cache. La solution à ce problème consiste à remplir les lignes de caches avec des données inutilisées (*padding*). Dans le cas précédent, après chaque donnée, la ligne de cache est remplie avec des octets inutilisés. Deux données vont donc remplir deux lignes de cache distinctes. La contrepartie de cette méthode est qu'elle remplit le cache de données inutiles expulsant ainsi des données valides et réduisant l'efficacité des caches. Il convient donc d'essayer de grouper les données privées pour réduire au minimum la perte et de limiter soigneusement le nombre de lignes de cache artificiellement remplies.

Bibliothèques d'allocation mémoire. L'allocation de la mémoire est à la charge du programme (ou des bibliothèques d'allocation mémoire utilisées). Le noyau s'occupe uniquement de la correspondance entre adresses virtuelles et adresses physiques et alloue des pages à la demande (lors d'un défaut de page). L'allocateur mémoire de l'application est responsable de découper les zones de mémoire affectées par le noyau en fonction de la demande de l'application.

Comme décrit par Hoard [20], un mécanisme d'allocation mémoire peut être jugé selon plusieurs critères :

- *La vitesse d'allocation.* Le mécanisme d'allocation est une fonction pouvant être très fréquemment appelée, il est donc essentiel de minimiser son coût.

- *Le passage à l'échelle.* Cela implique de minimiser (i) la prise de verrous qui réduit le passage à l'échelle, et (ii) la manipulation de données globales qui déclenche fréquemment le mécanisme de cohérence de cache.

- *La présence de faux partage.* Un allocateur mémoire doit placer les données en mémoire de manière à éviter le faux partage et à libérer le programmeur de la contrainte de remplissage artificiel des lignes de cache pour les données dynamiques.

- *La fragmentation et la consommation mémoire.* La fragmentation apparaît lorsque les zones allouées sont libérées. La capacité d'un allocateur mémoire à réutiliser ces zones libérées est donc essentielle pour limiter la consommation mémoire de l'application.

Ces problèmes sont communs à tous les allocateurs. Pour les architectures multi-cœurs, il est nécessaire de considérer le passage à l'échelle et la gestion du faux partage. Il est également important de ne pas augmenter trop sensiblement la consommation mémoire.

L'allocateur mémoire fréquemment utilisé est celui de la *GlibC (GNU C Library)*, une implantation de la bibliothèque standard du C. Les fonctions d'allocation de la bibliothèque *GlibC* sont tirées de la version 2 de la bibliothèque *ptmalloc*. Cette bibliothèque est très optimisée en ce qui concerne la gestion de la fragmentation mémoire, mais nous ne nous y intéresserons pas ici. L'allocation mémoire de la bibliothèque *GlibC* est basée sur le concept d'*arena*. Une *arena* est une zone de mémoire qui va servir de base pour l'allocation mémoire, ainsi qu'un verrou pour protéger l'allocation depuis cette zone. Lorsqu'un thread demande une nouvelle zone de mémoire au mécanisme d'allocation, il va tout d'abord essayer d'obtenir un verrou sur une *arena* (en partant de la dernière utilisée). S'il ne réussit pas à obtenir un verrou, il va alors créer une nouvelle *arena*. Si une *arena* ne contient pas assez de mémoire, sa capacité peut être augmentée. Enfin, lors de la libération d'une zone allouée, il est nécessaire de prendre le verrou sur l'*arena* depuis laquelle la zone a été allouée. Si cette *arena* ne contient plus de zones allouées, elle peut être libérée.

Ce mécanisme d'allocation mémoire souffre de plusieurs problèmes sur les architectures multi-cœurs. Premièrement, pour allouer une zone mémoire, il est nécessaire de prendre un verrou sur une *arena* réduisant ainsi le passage à l'échelle de l'allocation. De plus, bien qu'un thread essaie de réutiliser la dernière *arena* obtenue, plusieurs threads peuvent se servir de la même arena. Cela implique (i) la possibilité d'un faux partage entre les threads, et (ii) le déclenchement du protocole de cohérence de cache puisque plusieurs cœurs peuvent mettre à jour les données de l'*arena*.

Plusieurs mécanismes d'allocation mémoire permettant d'obtenir un bon passage à l'échelle ont été étudiés [48, 20, 61, 13, 70]. L'idée de ces mécanismes d'allocation est d'effectuer une allocation par thread. Le mécanisme de gestion de la mémoire de TBB [61], que nous utilisons dans le chapitre 5, utilise les mécanismes suivants pour allouer ou libérer de la mémoire :

- *L'utilisation d'un tas local à chaque thread.* L'allocation est effectuée si possible depuis ce tas et la libération remet les données dans ce tas. Comme un seul thread manipule ce tas, il n'est pas nécessaire de prendre un verrou. Dans le cas de TBB, le tas est découpé en *blocs* de différentes tailles pour éviter la fragmentation. Chaque bloc gère deux listes d'éléments libres : une liste privée pour les éléments alloués et libérés par ce cœur, qui ne nécessite par conséquent pas de prise de verrou, et une liste publique qui est utilisée par le cœur qui libère la zone mémoire, s'il n'est pas celui qui l'a allouée. Dans ce second cas, l'insertion dans la liste publique est effectuée grâce aux opérations atomiques. Lorsqu'un cœur effectue une allocation mémoire, il l'effectue préférentiellement depuis sa liste privée pour éviter le coût des opérations atomiques.

- *L'utilisation d'un tas global.* Le tas global permet de centraliser la mémoire allouée par le système et d'éviter que les tas locaux gardent trop de données libérées. Le tas global est alimenté par des appels systèmes lorsqu'il ne contient plus suffisamment de données libres et par les tas locaux lorsque ces tas contiennent trop de zones inutilisées. Pour manipuler le tas global, il est nécessaire d'utiliser un verrou.

L'avantage de ce type d'allocation est de fournir une allocation utilisant un minimum de verrous. De plus, comme les zones mémoires affectées à chaque tas local sont alignées sur une ligne de cache et ont une taille multiple de cette ligne, cela permet d'éviter le faux partage. Cependant, il est important de noter qu'utiliser des tas locaux à chaque thread peut engendrer une surconsommation mémoire importante si le nombre de threads est grand.

3.3 Bilan

Dans cette partie, nous avons présenté l'impact des architectures multi-cœurs sur les environnements d'exécution. Ces environnements d'exécution vont du noyau du système d'exploitation aux bibliothèques de niveau utilisateur.

Nous avons notamment mis en avant le fait que les systèmes d'exploitation connaissent une évolution, à la fois en terme d'architecture et d'abstraction fournies aux couches supérieures. La tendance pour les prochains systèmes d'exploitation semble être l'affectation de cœurs aux différents services du système d'exploitation et la réplication de ces services. La réplication peut être complète, c'est-à-dire que tout le noyau est répliqué comme dans les cas de Barrelfish et d'Helios, ou partielle, c'est-à-dire que seulement les services sont répliqués comme dans les cas de Fos et de K42.

Nous avons également étudié les évolutions des supports d'exécution. Le principal composant affecté par l'apparition des architectures multi-cœurs est l'ordonnanceur. L'ordonnanceur doit ainsi être capable de gérer la répartition spatiale des tâches. Cette répartition doit prendre en compte l'équilibrage de charge, mais aussi les affinités (ou les incompatibilités) entre les threads. Les affinités sont essentiellement dues au partage de mémoire entre les threads d'une même application. Les incompatibilités sont liées à la contention sur les composants partagés tels que les caches, le bus mémoire ou encore le contrôleur mémoire. Il est très compliqué de prendre en compte à la fois les affinités entre les threads, leurs incompatibilités et l'équilibrage de charge. À notre connaissance, les ordonnanceurs choisissent toujours une version simplifiée d'un des trois critères.

Enfin, nous avons mis en avant le fait que les bibliothèques de niveau utilisateur ainsi que les applications doivent également s'adapter pour gérer efficacement les architectures multi-cœurs. Les problématiques principales concernent la gestion de la mémoire. Cette gestion de la mémoire doit être capable de passer à l'échelle et doit également éviter le faux partage entre les cœurs.

Notons que des environnements d'exécution tels que McRT [81] ou TBB [61] proposent un support efficace des multi-cœurs en jouant à la fois sur l'ordonnancement et sur la gestion de la mémoire.

Chapitre 4

Positionnement de la contribution

Sommaire

4.1 Contexte de la contribution 59

4.2 Vol de tâches efficace pour les systèmes événementiels multi-cœurs 60

4.3 Étude et amélioration des performances du serveur Web Apache sur les architectures multi-cœurs 61

Dans les chapitres précédents, nous avons tout d'abord présenté l'apparition des architectures parallèles et tout particulièrement celle des multi-cœurs qu'ils soient homogènes ou hétérogènes. Nous avons ensuite abordé plusieurs axes de recherche pour une exploitation de ces nouvelles architectures. Ces travaux autour des architectures multi-cœurs portent à la fois sur les langages de programmation et les modèles d'exécution, et sur la construction d'applications passant à l'échelle. Dans la suite de ce document, nous ne nous intéressons qu'à l'aspect exécution d'applications en fixant à chaque fois le modèle de programmation. Dans ce chapitre, nous allons détailler le contexte de nos travaux, ainsi que le positionnement de nos contributions.

4.1 Contexte de la contribution

Les contextes généralement étudiés pour les applications multi-cœurs sont multiples. Ils balayent un spectre allant du décodage vidéo aux serveurs de données, en passant par les applications de calcul scientifiques. Ces dernières sont les plus prisées par la communauté du calcul haute performance (*HPC, High-Performance Computing*). Les serveurs de données ont été intensivement étudiés sur les architectures mono-cœurs mais très peu sur les architectures multi-cœurs. Nous allons donc, dans la suite, restreindre notre domaine d'étude à cette classe d'application. Ces applications ont l'avantage de permettre en théorie un passage à l'échelle idéal si les requêtes sont indépendantes. En effet, plusieurs architectures de serveurs permettent d'utiliser les processeurs multi-cœurs. Par exemple, il est possible d'affecter une requête par cœur, ce qui assure du travail pour tous les cœurs si le nombre de requêtes est élevé. Une autre possibilité est d'affecter aux différents cœurs une partie de la chaîne de traitement d'une requête (lecture de la requête, vérification et remplissage du cache de fichier, etc.).

Notre définition de serveur de données englobe tous les serveurs fournissant un service accessible au travers d'une interface réseau. Par exemple, nous incluons, dans cette catégorie, les serveurs Web fournissant un accès à des fichiers statiques ou dynamiques uniquement en lecture, les serveurs de fichiers permettant un accès en lecture et en écriture à un ensemble de fichiers, ou encore les serveurs vidéos permettant d'envoyer aux clients des flux vidéos avec différentes qualités.

Pour nos expériences, nous limitons le contexte d'étude aux architectures multi-cœurs homogènes en terme de fonctionnalité et en terme de fréquence. Nous disposons de deux machines de test : une machine composée de 8 cœurs Intel UMA et une machine composée de 16 cœurs AMD NUMA (voir chapitre 1 pour une description plus détaillée de ces architectures).

L'apparition des multi-cœurs implique de repenser à la fois l'environnement d'exécution des serveurs de données et leur architecture. Dans les chapitres suivants, nous nous intéressons à deux problématiques : (i) l'impact du mécanisme de vol de tâches sur la performance de deux serveurs de données événementiels sur l'architecture Intel à 8 cœurs, et (ii) le passage à l'échelle du serveur Web Apache sur l'architecture NUMA à 16 cœurs.

4.2 Vol de tâches efficace pour les systèmes événementiels multi-cœurs

Le modèle de programmation événementiel est un modèle de programmation fréquemment utilisé pour la conception de serveur de données [5, 9, 34, 35, 42, 59, 75, 103]. Nous avons vu dans le chapitre 2 que les avantages du modèle de programmation événementiel se situent essentiellement dans (i) la gestion à grain fin de la concurrence sur les ressources (notamment les E/S disques et réseau asynchrone), (ii) une empreinte mémoire plus faible que le modèle à base de threads, et (iii) des performances plus élevées que les serveurs utilisant des threads, principalement dues à l'absence de créations de threads et de changements de contextes. En outre, la programmation événementielle est bien adaptée à ce type d'applications étant donné qu'elles réagissent à des arrivées de requêtes (considérées comme des événements).

Nous avons pu également voir que le problème majeur de la programmation événementielle réside dans son incapacité à utiliser efficacement les architectures multi-cœurs. Cette incapacité est due au seul et unique thread utilisé pour l'exécution. Une solution, appelée *N-Copy*, existe. Cette solution consiste à utiliser une instance du serveur par cœur. Cependant, si cette solution est adaptée à un serveur dans lequel les requêtes sont toutes indépendantes, comme un serveur Web, il est compliqué (voire impossible) de la mettre en œuvre dans un système où les requêtes ne sont pas indépendantes, par exemple un serveur de fichiers dans lequel les fichiers peuvent être manipulés en écriture de manière concurrente. Un autre problème de cette solution réside dans la duplication des données. Cette duplication entraîne une surconsommation de mémoire et une réduction de certains composants de l'application tels que les caches de fichiers.

Une autre solution est l'introduction d'annotations (les *couleurs*). Les couleurs permettent de décrire l'exclusion mutuelle entre les traitements des événements et expriment ainsi le parallélisme de l'application. Ce mécanisme de couleurs nous semble très prometteur. En effet, il permet d'injecter incrémentalement du parallélisme dans l'application et à grain fin (au grain du traitement d'un événement). Les couleurs évitent également les situations d'interblocage puisque la gestion de l'exclusion mutuelle est laissée au support d'exécution. Il est à noter que les erreurs sont toujours possibles si le programmeur colorie ses événements de manière erronée. Néanmoins, ce problème est présent dans la plupart des modèles qui n'assurent pas une gestion automatique des accès concurrents à une ressource. Enfin, la garantie que le traitement d'un événement ne sera pas préempté ni déplacé sur un autre cœur durant son exécution simplifie grandement la mise en place du parallélisme. Contrairement à l'approche *N-Copy*, la programmation événementielle colorée ne duplique pas les données et bénéficie donc d'une empreinte mémoire limitée. Enfin, notons que la programmation événementielle, en raison de sa structuration en traitants d'événements, est adaptée aux nouvelles architectures multi-cœurs proposant du passage de messages efficace.

Dans ces travaux, nous nous intéressons à la programmation événementielle utilisant des couleurs pour exprimer le parallélisme. Nous étudions le support d'exécution Libasync-smp et tout particuliè-

rement le mécanisme d'équilibrage de charge utilisant le vol de tâches. En effet, le mécanisme de coloration permet une expression simple de la concurrence, mais peut toutefois provoquer des déséquilibres de charge en fonction des couleurs choisies par le programmeur. L'impact de ce mécanisme sur les performances de l'application n'a pas été évalué exhaustivement dans la publication initiale de Libasync-smp. La seule évaluation a consisté sur un test simple de rééquilibrer sur quatre cœurs des événements distribués sur trois cœurs. L'impact sur les performances du serveur Web n'a pas été étudié.

Les travaux présentés dans le chapitre 5 s'intéressent à l'impact sur les performances du mécanisme de vol de tâches de Libasync-smp. Nous montrons dans la suite de ce document que le mécanisme de vol de tâches peut être soit très bénéfique (+35 % d'amélioration de performances sur le serveur de fichier sécurisé *SFS (Secured File Server)* [69]) soit être responsable d'importantes dégradations de performances (33 % de réduction de performance sur un serveur Web similaire à celui décrit par Libasync-smp [102]).

Nous introduisons par la suite deux améliorations pour réduire l'impact négatif du vol de tâches dans le cas du serveur Web tout en conservant ses performances dans le cas de SFS. La première consiste à proposer un ensemble d'heuristiques permettant de guider le mécanisme de vol de tâches dans sa prise de décision. Ces heuristiques essaient notamment d'améliorer la localité des caches et de se préserver des vols non rentables. La seconde consiste à proposer un nouveau support d'exécution pour la programmation événementielle colorée. Ce nouveau support d'exécution multi-cœurs est appelé *Mely (Multi-core Event LibrarY)* et est compatible avec les applications développées précédemment pour Libasync-smp. Contrairement à Libasync-smp, l'architecture de *Mely* a été étudiée pour minimiser les coûts liés au vol de tâches. Par conséquent, Mely a un coût de vol très faible, ce qui le rend utilisable même avec des événements très courts.

Nous présenterons ensuite une évaluation complète de Mely en utilisant à la fois un ensemble de micro-tests et une évaluation sur le serveur Web et SFS. Nous montrerons notamment que notre solution égale ou améliore les performances par rapport à Libasync-smp avec ou sans vol de tâches. Nous montrons principalement que le serveur Web s'exécutant avec Mely a des performances 73 % meilleures que le serveur Web s'exécutant au-dessus de Libasync-smp avec vol de tâches et 25 % meilleures que celui s'exécutant au dessus de Libasync-smp sans vol de tâches.

4.3 Étude et amélioration des performances du serveur Web Apache sur les architectures multi-cœurs

La seconde partie de nos travaux porte sur l'étude et l'amélioration du passage à l'échelle du serveur Web Apache. Nous nous intéressons ici au passage à l'échelle du serveur Web Apache car c'est le serveur Web le plus déployé à l'heure actuelle. Ce serveur est utilisé pour héberger plus de la moitié des sites Web.

Le serveur Web est une classe d'application dont le passage à l'échelle est normalement aisé puisqu'il est possible de distribuer les connexions indépendantes sur les différents cœurs disponibles. La plupart des études sur le passage à l'échelle des serveurs de données se limitent à des charges fictives (c'est-à-dire différentes des charges observées sur les serveurs Web déployés dans l'industrie) ou des serveurs Web expérimentaux [18, 100]. Pariag *et al.* ont comparé les performances de différentes architectures de serveur Web sous une charge réelle sans toutefois s'intéresser au passage à l'échelle de l'ensemble.

Veal et Foong [95] ont réalisé la seule étude complète du passage à l'échelle d'un serveur Web sur les nouvelles architectures multi-cœurs. Cette étude a été effectuée en utilisant la version 2005 de l'injecteur de charge SpecWeb [86]. Cet injecteur de charge permet de simuler différents modèles de

charges tels que la possibilité de simuler un site marchand générant des pages Web dynamiques (*E-commerce*), un site de banque impliquant beaucoup de connexions sécurisées (*Banking*) ou encore un site de support en ligne proposant des documents de différentes tailles et quelques pages dynamiques (*Support*). Veal et Foong ont étudié les performances du serveur Web Apache en utilisant le modèle de charge *Support*. Ils ont mesuré que le serveur Web Apache ne passe pas à l'échelle sur une machine avec 8 cœurs équipée de 4 cartes réseau. Ils ont conclu que les problèmes de passage à l'échelle rencontrés ne proviennent ni d'un mauvais passage à l'échelle de la pile réseau de Linux ni du mauvais passage à l'échelle d'autres fonctions. Selon leurs mesures, le problème majeur vient de la contention sur le bus d'adressage commun aux cœurs. Il est important de considérer que ces résultats ne sont pas applicables sur une architecture NUMA puisque dans une telle architecture, la mémoire est distribuée sur les cœurs et une interconnexion remplace les précédents bus d'adresse et bus de données.

Dans le chapitre 6, nous étudions le passage à l'échelle du serveur Web Apache sur une architecture NUMA à 16 cœurs équipée de 16 cartes réseau. Pour cela, nous utilisons l'injecteur de charge SpecWeb 2005 et la charge *Support*. Nous montrons notamment que le serveur Web Apache ne présente pas un passage à l'échelle idéal et qu'il ne permet pas de saturer l'ensemble de ses 16 cartes réseau. La baisse de performance entre le passage à l'échelle idéal et les performances obtenues est de 31 %. Nous montrons à l'aide d'un ensemble d'indicateur que ces mauvais résultats sont dus à la baisse de l'efficacité des accès mémoire et à l'augmentation du temps passé dans les verrous du noyau.

À partir de ces constatations, nous proposons deux solutions. La première consiste à imposer qu'un processus Apache ne communique qu'avec un processus PHP local. En effet, nous montrons que les communications distantes jouent un rôle important dans la baisse de la performance des accès mémoire. Cette solution n'étant pas suffisante pour redresser le passage à l'échelle du serveur, nous montrons ensuite qu'il est nécessaire de considérer l'interconnexion entre les processeurs dans le mécanisme d'équilibrage de charge. En combinant ces deux propositions, les performances du serveur Web sont augmentées de 20 %. Toutefois, nous n'atteignons pas le passage à l'échelle idéal. Par conséquent, nous terminons ces travaux en proposant des pistes d'études supplémentaires et en discutant des possibilités de gains restant.

Chapitre 5

Vol de tâches efficace pour les systèmes événementiels multi-cœurs

Sommaire

5.1 Description du support d'exécution Libasync-smp 64
5.1.1 Programmation événementielle colorée 64
5.1.2 Architecture du support d'exécution Libasync-smp 67
5.1.3 Algorithme de vol de tâches 69
5.1.4 Évaluation des performances du vol de tâches 71
5.2 Un nouvel algorithme de vol de tâches 73
5.2.1 Vol de tâches sensible à la localité 74
5.2.2 Vol de tâches avec analyse de la rentabilité 74
5.2.3 Vol de tâches avec pénalité 75
5.3 L'environnement d'exécution Mely 76
5.3.1 Architecture de Mely 76
5.3.2 Mise en œuvre du vol de tâches 78
5.3.3 Détails supplémentaires de mise en œuvre 79
5.4 Évaluation du mécanisme de vol de tâches de Mely 82
5.4.1 Conditions expérimentales 82
5.4.2 Micro-tests 82
5.4.3 Serveurs de données 85
5.5 Conclusion 91

Nous avons vu dans les chapitres précédents que le modèle de programmation événementiel est un modèle fréquemment employé pour concevoir des serveurs de données. Les processeurs multi-cœurs sont de plus en plus répandus car l'augmentation du parallélisme est la seule possibilité d'amélioration de leurs performances puisque les fréquences d'horloge stagnent. Par conséquent, il est crucial pour ce modèle de programmation d'exploiter efficacement le parallélisme offert par ces architectures matérielles. Pour cela, la méthode de coloration des événements introduite par Libasync-smp est une approche prometteuse. En effet, elle permet à un programmeur d'injecter incrémentalement du parallélisme dans son application. Pour obtenir un maximum de parallélisme lors de l'exécution de l'application, l'environnement d'exécution de Libasync-smp utilise un mécanisme de vol de tâches permettant d'équilibrer dynamiquement la charge sur les cœurs disponibles.

Dans ce chapitre, nous étudions tout d'abord l'impact du mécanisme de vol de tâches sur les performances des applications événementielles multi-cœurs. Nous montrons notamment que le mécanisme de vol de tâches peut dégrader les performances et illustrons ce problème avec un serveur Web. Nous introduisons par la suite plusieurs améliorations pour augmenter les performances du vol de tâches. Ces améliorations sont implantées au sein d'un environnement d'exécution compatible avec Libasync-smp, appelé Mely. Nous avons évalué la performance de nos propositions à la fois sur des micro-tests et sur des serveurs de données réalistes. Tout particulièrement, nous améliorons les performances d'un serveur Web jusqu'à 73 % par rapport au support d'exécution Libasync-smp avec vol de tâches.

5.1 Description du support d'exécution Libasync-smp

Dans cette section, nous introduisons l'environnement d'exécution Libasync-smp [102]. Nous donnons quelques précisions supplémentaires sur le modèle de programmation événementiel par rapport à ce qui a été vu dans le chapitre 2. Nous décrivons ensuite l'architecture de Libasync-smp et le mécanisme de vol de tâches. Enfin, nous concluons cette section avec une évaluation de ce mécanisme.

5.1.1 Programmation événementielle colorée

Le modèle de programmation de Libasync-smp est basé sur celui de Libasync. Libasync est une bibliothèque écrite en C++ qui offre la possibilité de construire facilement des applications événementielles efficaces. Pour cela, Libasync se base sur un ensemble de fonctions et de *templates* C++ qui permettent de gérer aisément la création d'un événement associé à son traitant (*callback*), l'association de traitants à des opérations d'E/S, ou encore la prise en charge des signaux. L'interface de programmation exposée par Libasync est résumée dans le tableau 5.1. La fonction `wrap` est utilisée pour créer un objet *callback* qui est une structure représentant un événement. Cette fonction prend en paramètres le traitant associé à l'événement, ainsi que les paramètres de l'événement (continuation). Le nombre de ces paramètres est variable et les paramètres peuvent avoir des types différents. `fdcb` est une fonction qui permet de déclencher l'exécution d'un traitant d'événements suite à un changement d'état lié aux opérations d'E/S. Il est possible de différencier un événement de lecture d'un événement d'écriture. `sigcb` propose la même fonctionnalité pour la gestion des signaux. Enfin, il est possible en utilisant les fonctions `delaycb` et `timecb` de spécifier des contraintes temporelles sur l'exécution des événements. Notons que, comme un traitement d'événement ne peut pas être préempté, il ne peut pas y avoir de garantie forte sur l'instant d'exécution de l'événement. Tout au plus, le système garantit que l'événement ne sera pas exécuté avant l'échéance spécifiée par le programmeur.

Libasync fait un ensemble d'hypothèses sur le contenu des traitants ainsi que sur la gestion des événements. Dans le modèle de programmation événementiel, il est convenu que le traitement d'un événement ne sera jamais interrompu par l'exécution d'un autre traitant. Cela facilite grandement la gestion des accès concurrents à des données partagées. Toutefois, cela implique d'utiliser uniquement des E/S non bloquantes. En effet, si le traitement d'un événement est bloqué sur une E/S, aucun autre traitement ne pourra faire de progrès réduisant ainsi grandement les performances de l'application. La gestion du cycle de vie des continuations est effectuée de manière automatique par un mécanisme de ramasse-miettes utilisant un comptage de références. Ce mécanisme est également fourni à l'application pour l'aider à gérer les zones allouées dynamiquement. En effet, une continuation peut comporter des paramètres alloués dynamiquement et manipulés éventuellement par plusieurs événements. Dans ce cas, il est difficile de savoir quand tous les événements référençant ces zones de mémoire ont été

`événement = wrap (traitant, paramètres du traitant )`	Création d'un événement
fdcb `(descripteur de fichier, opération, événement)`	Association d'un événement à un descripteur de fichier
sigcb `(signal, événement)`	Association d'un événement à l'arrivée d'un signal
timecb `(temps absolu, événement)`	Lancement d'un événement à une heure donnée
delaycb `(délai, événement)`	Lancement d'un événement dans un délai donné

TABLE 5.1 – Interface de programmation de Libasync.

traités. Pour gérer ces zones de mémoire, un mécanisme de ramasse-miettes simplifie grandement la tâche du programmeur.

Comme nous l'avons introduit dans le chapitre 2, Libasync-smp est une version étendue de l'environnement d'exécution Libasync, permettant l'utilisation des architectures parallèles. Pour cela, Libasync-smp apporte quelques extensions au modèle événementiel classique, ainsi qu'à l'interface de programmation, tout en restant compatible avec le modèle initial. Pour gérer la concurrence, Libasync-smp introduit la notion de coloration des événements qui permet de décrire les traitements des événements devant être sérialisés et les traitements pouvant être effectués en parallèle. Une couleur est affectée à chaque événement. Le support d'exécution est chargé d'assurer que deux événements de la même couleur ne seront jamais exécutés simultanément. Cette modification du modèle de programmation entraîne une modification de l'API fournie au programmeur. Le tableau 5.2 décrit ces changements.

`événement =` **cwrap** `(traitant, paramètres du traitant,` *couleur*`)`	Création d'un événement associé à une couleur
`événement =` **cpwrap** `(traitant, paramètres du traitant,` *couleur, priorité*`)`	Création d'un événement associé à une couleur et une priorité
cpucb `(événement)`	Enregistrement d'un événement en tête de file
cpucb_tail `(événement)`	Enregistrement d'un événement en queue de file

TABLE 5.2 – Ajouts à l'API de Libasync pour prendre en compte la coloration d'événements.

Premièrement, nous pouvons constater que, en plus de la fonction `wrap`, deux nouvelles fonctions ont été ajoutées pour créer des événements. La première, nommée `cwrap`, permet de créer des événements avec une couleur. La couleur est représentée par un entier sur 2 octets. 65536 couleurs sont ainsi utilisables. La seconde permet, en plus de la couleur, d'affecter une priorité à l'événement. Par défaut, pour préserver la compatibilité avec la version initiale de Libasync-smp, la fonction `wrap` créer des événements de couleur 0. Ainsi, tous les événements créés par la fonction `wrap` ont la même couleur et sont donc sérialisés. Contrairement à Libasync qui permet uniquement d'appeler un traitant d'événements lors de l'apparition d'une E/S, d'un signal, ou lors de l'expiration d'un compte à rebours, Libasync-smp permet de créer des événements internes pour augmenter le parallélisme. Le

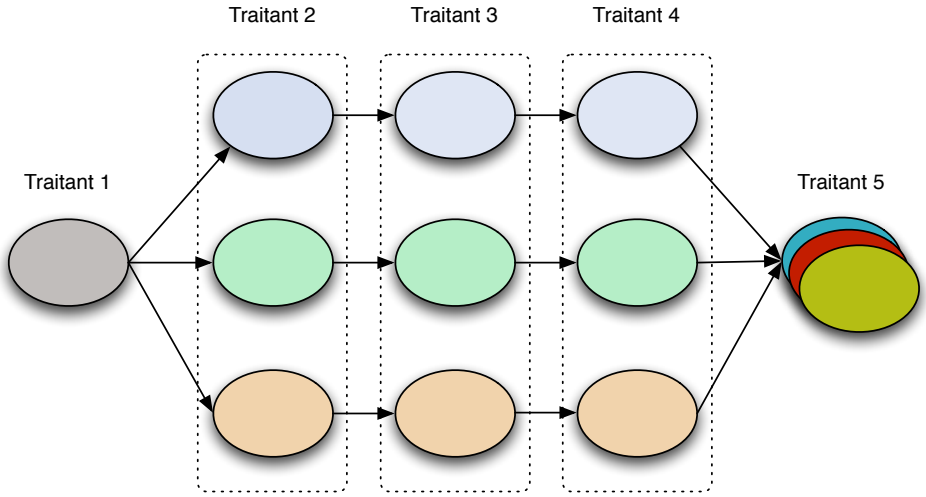


Figure 5.1 – Un exemple de coloration d'une architecture découpée en 4 traitants d'événements et utilisant 4 couleurs.

traitement d'un événement d'E/S ou d'un minuteur est découpé en plusieurs traitements successifs. L'idée est de permettre au programmeur de découper son travail pour permettre l'avancement des autres tâches. Dans ce cas, le modèle de programmation ressemble à l'ordonnancement coopératif que nous avons décrit dans la section 2.1.2. Ce découpage est effectué avec les fonctions `cpucb` (ajout en tête de file pour être la prochaine tâche à traiter) et `cpucb_tail` (ajout en queue de file).

La figure 5.1 présente un exemple de coloration d'événements. Nous pouvons voir sur cette figure que le mécanisme de coloration permet d'exprimer différentes formes de parallélisme. Par exemple, il est possible d'autoriser différentes instances d'un même traitant à travailler sur des données disjointes (coloration par flot) en les coloriant avec des couleurs différentes. Dans la figure, cette méthode de coloration est utilisée sur les événements des traitants 2, 3 et 4. Cette solution est fréquemment adoptée pour les serveurs de données dans lesquels les événements sont coloriés avec le numéro de connexion, pour permettre de traiter plusieurs connexions en parallèle. Il est aussi possible de garantir que toutes les instances d'un même traitant seront exécutées en exclusion mutuelle (coloration par traitant) en leur assignant la même couleur. Ce type de coloration est utilisé dans la figure 5.1 pour le traitant 1. Ceci est utilisé lorsqu'un traitant maintient un état global (par exemple, le nombre de clients connectés). Enfin, le traitant 5 combine les deux aspects. Les événements liés au traitant 5 sont coloriés avec une des trois couleurs. Ce type de coloration est généralement mis en place lorsqu'il est nécessaire d'avoir plusieurs instances d'un même traitant, c'est-à-dire que les données liées au traitant sont dupliquées bien que toutes les instances partagent le même code. L'objectif de cette coloration est d'augmenter la résistance à la charge tout en garantissant que chaque répliqua est manipulé en exclusion mutuelle. Par exemple, un serveur Web possède généralement plusieurs instances du cache de fichiers. En effet, comme toutes les requêtes valides doivent consulter le cache, avoir une seule instance de celui-ci pourrait être un goulot d'étranglement. Chaque instance gère un sous-ensemble disjoint des fichiers demandés et doit être manipulée en exclusion mutuelle afin d'éviter des opérations de consultation et de remplissage concurrente. Pour cela, les événements liés aux traitants de cache sont coloriés en fonction du fichier demandé. Une couleur par instance du cache de fichiers est utilisée.

Il est important de considérer que si les couleurs permettent d'exprimer de nombreux types de parallélisme, elles sont cependant moins expressives que des verrous. Il n'est par exemple pas pos-

sible d'exprimer une sémantique de type *lecteurs-rédacteurs*. Toutefois, elles ont l'avantage d'être moins sources d'erreur que les verrous. En effet, lors de l'utilisation de verrous, le principal problème est l'apparition d'interblocages si les verrous ne sont pas pris dans un ordre correct. Ce problème est exacerbé par la nécessité de l'utilisation de verrous à grain fin pour obtenir le maximum de parallélisme. La délégation de la gestion de l'exclusion mutuelle au support d'exécution permet d'éviter ce problème d'interblocages. De plus, le type de parallélisme proposé par la coloration d'événements est généralement suffisant pour les serveurs de données. En effet dans ce type d'application, le parallélisme se fait généralement entre les différentes requêtes. Enfin, pour les rares cas où le parallélisme proposé par les couleurs n'est pas suffisant, il est possible de réintroduire précautionneusement des verrous adaptés. Il est toutefois important de prendre en compte le fait que si un des cœurs se bloque sur un verrou, ce cœur n'effectuera plus d'opérations jusqu'à l'obtention du verrou, car il n'y a qu'un seul thread par cœur et le traitement d'un événement ne peut pas être préempté.

Le pseudo-code donné dans la figure 5.2 est un exemple d'implantation d'un serveur de données en utilisant Libasync (colonne de gauche), ainsi que les changements effectués pour rendre ce code parallèle (colonne de droite). Le serveur implanté est un serveur d'écho. Ce serveur accepte les connexions TCP, lit caractère par caractère sur la connexion et renvoie le caractère lu au client. Ce serveur montre les mécanismes d'enregistrement d'événements et d'association d'un événement à une activité de lecture sur une socket. La première étape est d'enregistrer une socket d'écoute sur un port donné et de l'associer à un traitant d'acceptation des connexions. Lorsque cette socket reçoit une demande de connexion, elle l'accepte et associe la connexion au traitant de lecture. Le traitant de lecture est appelé lors de la réception d'une donnée et enregistre le traitant d'écriture pour chaque octet lu. Lorsque la connexion est détectée comme fermée, la socket est fermée côté serveur et la surveillance de cette socket est arrêtée en supprimant le traitant associé. Notons que cette implantation est très simplifiée. Par exemple, la fermeture de connexion est uniquement à l'initiative du client. Elle est également sensible à certaines erreurs. Dans un serveur réel, avant de fermer la connexion, il est impératif de s'assurer qu'aucun événement d'écriture n'a été enregistré (voir section 5.4.3.1).

La colonne de droite illustre les changements nécessaires à l'utilisation des architectures multi-cœurs. Nous pouvons voir que ces changements sont minimes et concernent la coloration des événements. Dans notre exemple, tous les événements liés au traitant d'acceptation des connexions ont la couleur 1. Les événements liés à une connexion sont coloriés avec le numéro de descripteur de fichier associé à la connexion, ce qui assure que plusieurs connexions peuvent être traitées en parallèle, tout en garantissant une seule opération à la fois sur une même connexion (par exemple, pas de lectures concurrentes sur la même socket).

5.1.2 Architecture du support d'exécution Libasync-smp

Après avoir décrit et illustré le modèle de programmation de Libasync-smp, nous allons maintenant nous intéresser à sa mise en œuvre. Libasync-smp est un environnement d'exécution pour la programmation événementielle multi-cœurs. Pour cette raison, l'architecture Libasync-smp est structurée autour d'un thread et d'une file d'événements par cœur. La figure 5.3 illustre l'architecture du support d'exécution Libasync-smp. Un événement est représenté par une structure de données contenant un pointeur vers le traitant associé, les paramètres d'appel de ce traitant, ainsi que les informations de couleur et de priorité. Le traitement de l'événement est effectué par le cœur qui le possède dans sa file. Un seul thread par cœur est nécessaire, car le modèle de programmation impose que les traitements des événements soient non bloquants. La solution d'une file par cœur a été choisie pour réduire la contention sur les files. Sur la figure, nous pouvons voir trois cœurs avec leur thread et leur file d'événements. Le cœur 3 n'a plus de travail à effectuer tandis que les cœurs 1 et 2 ont encore des événements à traiter en attente. Trois couleurs sont utilisées pour colorier les événements.

```
// Declaration des traitants
void Accept(int fd);
void ReadOneByte(int fd);
void WriteResponse(int fd, char * buffer);

int main(int argc, char **argv) {

   int as = create_accept_socket(8080);

   fdcb(as, selread, wrap(Accept, as));

   amain();
}

void Accept(int fd) {
   int s = accepter_connexion(fd);

   if (s == -1 && errno!= EAGAIN) {
      // Erreur
   }
   else if (s > -1) {
      int ret = setNonblocking(s);
      if (ret < 0) {
         // Erreur
      }

      fdcb(s, selread, wrap(ReadOneByte, s));
   }
}

void ReadOneByte(int fd) {
   char* buf =
      (char *) malloc(sizeof(char));

   int rd = read(fd, buf, sizeof(char));

   if (rd == -1 && errno != ECONNRESET) {
      //Erreur
   }

   else if (rd == -1 || rd == 0) {
      // Connexion fermee
      fdcb(fd, selread, NULL);
      close(fd);
      return;
   }

   cpucb_tail(wrap(WriteResponse,fd, buf));
}

void WriteResponse(int fd, char * buffer) {
   int wr = write(fd, buffer, sizeof(char));

   if (wr == -1  ) {
      // Erreur
   }
   else if (wr == -1 && errno == EAGAIN) {
      wr = 0;
   }

   free(buffer);
}
```

```
fdcb(as, selread, cwrap(Accept, as, 1));

fdcb(s, selread, cwrap(ReadOneByte, s, s));

cpucb_tail(cwrap(WriteResponse,fd, buf, fd));
```

Version mono-cœur | Version multi-cœur

Figure 5.2 – Code d'un serveur écho simple dans sa version mono-cœur ainsi que les changements nécessaires pour le paralléliser en utilisant des couleurs.

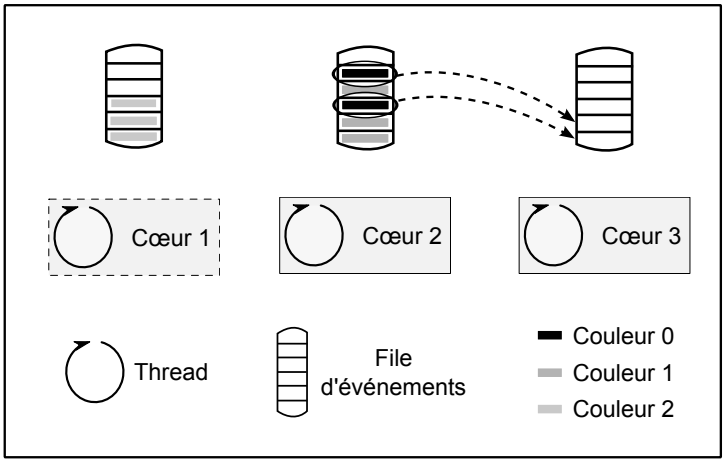


Figure 5.3 – Architecture du support d'exécution événementielle multi-cœurs Libasync-smp.

Comme nous l'avons vu dans la section précédente, étant donné que plusieurs threads manipulent simultanément des événements, il est nécessaire d'utiliser les couleurs pour exprimer l'exclusion mutuelle entre deux événements. Le support d'exécution doit garantir que deux événements de la même couleur ne peuvent pas être traités simultanément. Pour assurer cette propriété, Libasync-smp affecte les événements de la même couleur sur le même cœur donc dans la même file d'événements. Comme les événements ne sont pas préemptibles, cela permet d'assurer la garantie d'exclusion mutuelle. Initialement, les couleurs sont affectées aux cœurs en utilisant une fonction de répartition simple :

```
cœur de destination = couleur % nombre de cœurs
```

L'équilibrage de charge est ensuite assuré par un mécanisme de vol de tâches, que nous détaillons dans la section 5.1.3.

En plus de la notion de couleur, le programmeur peut affecter des priorités aux événements. Il est important de savoir que ces priorités sont uniquement données à titre indicatif pour le support d'exécution, mais sans garantie que les priorités seront respectées. Plus précisément, pour éviter les coûts de recherche dans la file, la recherche de l'événement de plus grande priorité est limitée aux 5 premiers événements de la file. De plus, cette priorité est pondérée par la présence d'un événement de la couleur précédemment exécutée. En effet, l'ordonnanceur d'événements essaie d'exécuter successivement des événements de la même couleur. L'objectif de cette politique est d'améliorer l'efficacité des caches processeurs puisque deux événements de la même couleur ont une grande probabilité de manipuler les mêmes données.

Les files d'événements peuvent être manipulées par plusieurs cœurs en même temps, car plusieurs cœurs peuvent produire un événement de la même couleur en même temps. Il est donc nécessaire de protéger l'accès à ces files (et aux métadonnées associées telles que le nombre d'événements présents dans la file) en utilisant des verrous. Les verrous utilisés dans Libasync-smp font de l'attente active avec une instruction de *test-and-set* atomique (ces verrous sont appelés *spinlocks*). En effet, il n'y a pas d'intérêt à relâcher le cœur puisqu'il n'y a pas d'autres threads à ordonnancer. Notons que cette constatation est valide si les critères de consommation énergétique ne sont pas une préoccupation et si l'application est seule à être ordonnancée sur la machine.

5.1.3 Algorithme de vol de tâches

Comme nous l'avons vu dans la partie précédente, les événements colorés sont répartis sur les cœurs en utilisant une fonction de *modulo*. Cette fonction ignore le fait que le traitement d'événe-

ments de certaines couleurs peut demander plus de temps que pour d'autres (par exemple, lorsque le nombre d'événements par couleur est très déséquilibré) ou que les durées d'exécution des traitants d'événements sont très variables. Par conséquent, comme nous le montre la figure 5.3, la charge de travail sur les cœurs peut être déséquilibrée réduisant ainsi les performances. Pour résoudre ce problème, Libasync-smp base l'équilibrage de charge sur un mécanisme de vol de tâches (dont l'idée de base a été présentée en section 3.2.1). Avec ce mécanisme, lorsqu'un cœur n'a plus d'événement à traiter dans sa file, il essaie de récupérer des événements dans la file des autres cœurs. Par exemple, dans la figure 5.3, le cœur 3 qui ne possède plus de tâches vole des événements dans la file du cœur 2.

```
core_set = construct_core_set();                    (1)
foreach(core c in core_set) {
   LOCK(c);
   if(can_be_stolen(c)) {                           (2)
      color = choose_colors_to_steal(c);            (3)
      event_set = construct_event_set(c, color);    (4)
   }
   UNLOCK(c);
   if(!is_empty(event_set)) {
      LOCK(myself);
      migrate(event_set);                           (5)
      UNLOCK(myself);
      exit;
   }
}
```

Figure 5.4 – Pseudo-code illustrant le mécanisme de vol de tâches de Libasync-smp.

Le mécanisme de vol de tâches de Libasync-smp est représenté (en pseudo-code) par la figure 5.4. Nous avons numéroté les différentes fonctions de cet algorithme. Tout d'abord, l'algorithme choisit l'ordre dans lequel le vol sur les cœurs sera réalisé. Cela est effectué en appelant la fonction `construct_event_set`. Cette fonction construit une série de cœurs sur lesquels voler des tâches (`core_set`). Nous ne détaillons dans ce paragraphe que l'enchaînement des fonctions. Le contenu précis des fonctions est décrit dans le paragraphe suivant. Pour chacun des cœurs sélectionnés, le cœur voleur vérifie s'il peut être ciblé en utilisant la fonction `can_be_stolen`. Si des événements peuvent être volés, le voleur choisit une couleur à voler en utilisant la fonction `choose_color_to_steal` et construit la liste de tous les événements de cette couleur (`construct_event_set`). Il est nécessaire de voler l'intégralité des événements de la couleur choisie pour garantir la contrainte d'exclusion mutuelle entre événements d'une même couleur. Si cette liste n'est pas vide, le cœur voleur la fait ensuite migrer dans sa propre file (fonction `migrate`) et sort de la fonction de vol.

Nous allons maintenant décrire plus précisément le principe de chacune des fonctions citées ci-dessus. La fonction `construct_core_set` permet de construire une liste de cœurs à voler. Cette liste commence par le cœur qui possède le plus d'événements dans sa file. Ensuite, la suite de la liste contient les cœurs successifs (en prenant en compte le numéro du cœur). Par exemple, sur une machine à 8 cœurs, si le cœur 6 est celui possédant le plus d'éléments dans sa file, la liste ainsi construite contiendra la suite {6, 7, 0, 1, 2, 3, 4, 5}. La fonction `can_be_stolen` retourne « vrai » lorsque le cœur a des événements d'au moins deux couleurs différentes dans sa file. En effet, pour garantir l'exclusion mutuelle entre deux événements d'une même couleur, il n'est pas possible de voler les événements coloriés avec la couleur en cours d'exécution. Un vol ne peut donc avoir lieu que s'il y a des événements d'une autre couleur dans la file. La fonction `choose_color_to_steal` parcourt l'intégralité de la file à la recherche de la couleur à voler. En plus de la contrainte citée ci-dessus sur l'impossibilité de voler la couleur en cours d'exécution, il n'est pas possible de voler

une couleur dont le nombre d'événements associés dans la file représente plus de la moitié de la file. L'objectif de cette mesure est d'éviter que le cœur qui a été volé se retrouve sans tâches. Cependant, une couleur représentant moins de la moitié des événements de la file du cœur et n'étant pas la couleur active peut ne pas exister dans la file. Dans ce cas, bien que `can_be_stolen` ait répondu « vrai », le cœur n'est pas une bonne cible, ce qui nécessite de passer à un autre cœur. La fonction `construct_event_set` construit la liste des événements présents dans la file associés à la couleur choisie. Lorsqu'un événement est ajouté à cette liste, il est retiré de la file d'événements du cœur victime. Cette fonction nécessite de parcourir l'intégralité de la file d'événements pour récupérer tous les événements d'une couleur donnée. C'est notamment le cas lorsque le dernier élément de la file est de la couleur choisie par le voleur. Pour réduire le coût de l'opération, un compteur d'évènements est maintenu pour chaque couleur. Comme les couleurs sont représentées par un entier sur 2 octets, cela nécessite un tableau de 65535 entrées. Ce tableau permet de déterminer si tous les événements ont été volés. Lorsque cette condition est vérifiée, il n'est pas nécessaire de continuer à parcourir la file de la victime. La fonction `migrate` transfère simplement la liste temporaire d'événements au sein de la file d'événements du cœur voleur.

Pour manipuler les files d'événements lors du vol, il est impératif de prendre un verrou sur le cœur cible et un verrou sur le cœur voleur. Les opérations à protéger par le verrou du cœur cible sont les numéros 2, 3 et 4 qui manipulent la file de ce cœur. Il est nécessaire de les protéger contre d'autres vols pouvant s'effectuer en parallèle, mais également contre les ajouts et les retraits dans la file d'événements. Ensuite, pour la même raison, il est impératif que le voleur prenne un verrou sur sa propre file. Il n'est pas possible dans le cas de la programmation événementielle colorée d'utiliser des structures sans verrous [30]. Comme nous l'avons vu dans la section 3.2.1, ces files ne peuvent pas être mises en place parce qu'un thread peut produire des tâches à destination d'un autre thread, alors que ces files exigent qu'un thread produise uniquement des tâches pour lui-même.

Enfin, si sans vol de tâches l'association entre un cœur et une couleur peut être déterminée simplement grâce à une fonction de *modulo*, l'introduction du vol de tâches oblige Libasync-smp à maintenir un tableau fournissant cette association.

5.1.4 Évaluation des performances du vol de tâches

Les auteurs de Libasync-smp ont évalué les performances de Libasync-smp en utilisant deux serveurs de données : un serveur de fichiers sécurisé (*SFS*) [69] et un serveur Web. Le serveur Web n'est pas disponible publiquement, mais Zeldovich *et al.* décrivent son architecture dans l'article sur Libasync-smp. Ces travaux ont montré que les applications développées en utilisant le modèle de programmation événementiel coloré sont capables de tirer parti des architectures multi-cœurs. Toutefois, l'impact du mécanisme de vol de tâches sur la performance de ces serveurs de données n'a pas été étudié. Les performances de ce mécanisme de rééquilibrage de charge ont seulement été évaluées sur un micro-test simple.

Nous avons donc décidé d'étudier l'impact du mécanisme de vol de tâches sur la performance des serveurs de données dans le contexte de la programmation événementielle colorée. Pour cela, nous avons développé un serveur Web réaliste basé sur l'architecture de celui utilisé pour les résultats initiaux. Nous avons évalué SFS et notre serveur Web en activant puis en désactivant le mécanisme de vol de tâches. Des détails sur la configuration du serveur Web, ainsi que sur la configuration utilisée pour les tests sont fournis dans la section 5.4. Toutes nos expériences ont été reproduites plusieurs fois et nous observons une variation très faible (moins de 1 %).

La figure 5.5 présente les résultats obtenus sur SFS lorsque 16 clients font des requêtes de lecture sur un fichier de 200 Mo. Ce test est similaire à celui de Zeldovich *et al.*. Nous pouvons voir que le vol de tâches améliore sensiblement le débit observé du serveur (+35 %). La raison est que ce serveur

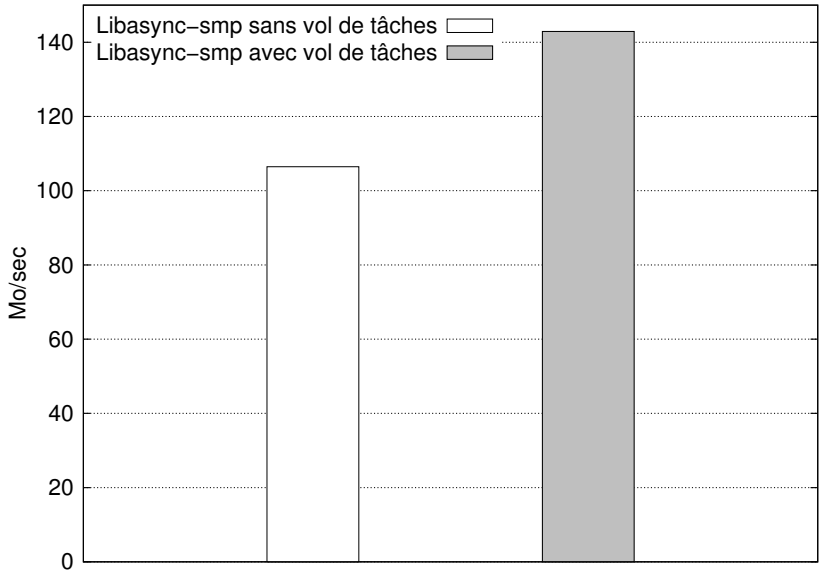


Figure 5.5 – Performances du serveur de fichiers sécurisé SFS en utilisant Libasync-smp avec et sans mécanisme de vol de tâches.

passe la majeure partie de son temps à effectuer des opérations cryptographiques coûteuses. Ainsi, il est très rentable de bien rééquilibrer la charge.

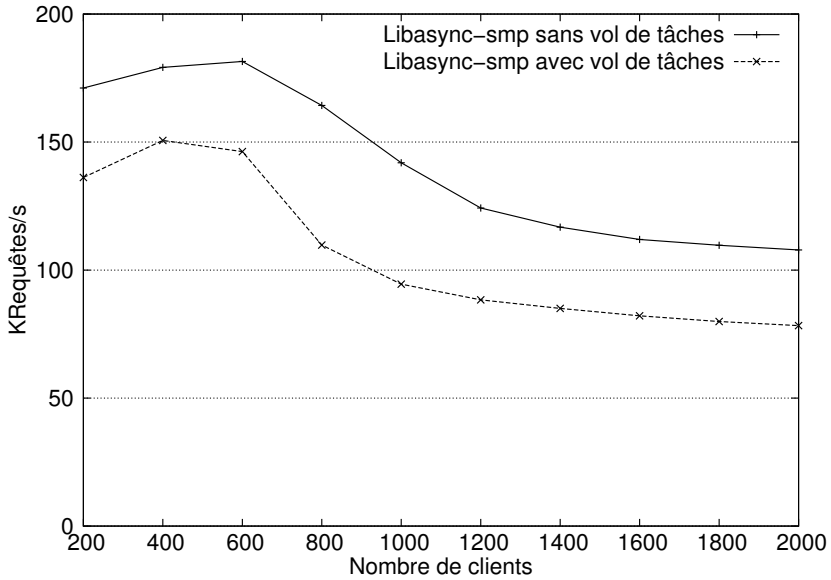


Figure 5.6 – Performance du serveur Web SWS en utilisant Libasync-smp avec et sans vol de tâches.

La figure 5.6 montre le débit du serveur Web lorsqu'un nombre de clients variable accèdent à des fichiers de 1 Ko. Cette figure montre bien que, contrairement au cas précédent, la performance du serveur Web est impactée très négativement par le mécanisme de vol de tâches (jusqu'à 33 % de baisse de performance). La raison est que le serveur Web manipule des événements dont le temps de traitement est bien plus court que les opérations cryptographiques de SFS. Par conséquent, les coûts liés au vol de tâches sont proportionnellement bien plus importants.

Pour comprendre de façon plus précise ce résultat, nous avons comparé, pour le serveur Web et pour SFS, le temps nécessaire pour voler un ensemble d'événements avec le temps moyen passé à exécuter cet ensemble d'événements volés. Les résultats obtenus sont présentés dans le tableau 5.3. Nous observons notamment que le temps nécessaire pour réaliser un vol est bien plus élevé dans le cas du serveur Web que dans le cas de SFS (plus de 40 fois plus long). Cela contraste avec la quantité de temps de traitement volé, qui est bien moins importante dans le cas du serveur Web que dans le cas de SFS. Cela explique pourquoi le vol est plus rentable dans le cas du serveur de fichiers sécurisé.

Serveur étudié	**Temps moyen d'un vol (cycles)**	**Temps moyen de traitement volé (cycles)**
SFS	4.8K	1200K
Serveur Web	197K	20K

TABLE 5.3 – Comparaison du temps passé à voler des événements par rapport au temps passé à exécuter ces événements volés dans le cas du serveur Web SWS et du serveur de fichiers SFS.

Nous attribuons ces piètres performances du vol de tâches dans le cas du serveur Web à deux raisons principales. La première est que l'algorithme de vol est relativement naïf : un voleur ne vérifie pas avant de voler si ce vol est réellement intéressant et va être amorti. Plus précisément, les fonctions `construct_core_set`, `can_be_stolen` et `choose_color_to_steal` ne prennent en compte ni le coût intrinsèque du vol ni la quantité de temps de traitement volé pour vérifier si le vol est pertinent.

De plus, la fonction `construct_core_set`, qui construit la liste des cœurs à tester successivement lors d'un vol, ne prend pas en compte les spécificités des architectures multi-cœurs. Tout particulièrement, elle ne considère pas la proximité qui peut exister entre les cœurs du fait d'un partage de cache. Pour étayer nos affirmations, nous avons mesuré le nombre de fautes de cache L2 par événement traité sur le serveur Web avec et sans vol de tâches. Nous avons pu constater une augmentation significative de 146 % du nombre de fautes de caches L2 par événement lorsque le vol de tâches est activé. Ce résultat suggère qu'un algorithme de vol efficace doit essayer de favoriser la relocalisation d'événements sur un cœur avec un cache L2 partagé.

La seconde raison pour laquelle le vol de tâches peut s'avérer inefficace dans Libasync-smp est que l'architecture n'a pas été conçue pour réduire les coûts de ce mécanisme. Par exemple, comme décrit dans la section précédente, la fonction `construct_event_set`, retirant les événements de la couleur choisie de la file du cœur, peut avoir besoin de parcourir l'intégralité de la liste pour récupérer tous les événements associés à la couleur volée. Sur notre plate-forme de test, le temps requis pour vérifier si un élément de la liste est une bonne cible est de 190 cycles. Cela explique pourquoi le coût du vol de tâches est si important dès que le nombre d'événements dans la file est grand. Par exemple dans le cas du serveur Web, le cœur le plus chargé peut contenir plus de 1000 éléments dans sa file. Ces résultats montrent très clairement qu'il est nécessaire de réduire le coût du vol de tâches et de le rendre autant que possible indépendant du nombre d'éléments dans la file du cœur victime.

5.2 Un nouvel algorithme de vol de tâches

Dans la section précédente, nous avons vu que le mécanisme de vol de tâches de Libasync-smp peut dans certains cas réduire sensiblement les performances des serveurs de données. Une des causes de cette baisse de performance est la naïveté de l'algorithme de vol de tâches. Dans cette section, nous présentons trois différentes heuristiques dont l'objectif est d'améliorer le comportement du vol de tâches pour augmenter ses performances. Pour cela, les heuristiques que nous allons présenter essaient

de faire prendre les bonnes décisions aux fonctions `construct_core_set`, `can_be_stolen` et `choose_color_to_steal`, qui ont été introduites dans la section 5.1.3. Le premier objectif est d'améliorer l'efficacité des caches en tirant profit du partage potentiel de caches entre deux cœurs d'un même processeur ainsi qu'en prenant des décisions minimisant les déplacements de données entre les caches. Le second objectif est de ne voler des événements que lorsqu'on est sûr que ce vol sera bénéfique.

5.2.1 Vol de tâches sensible à la localité

La première heuristique que nous présentons, appelée *VSAL (Vol de tâches Sensible À la Localité)*, a pour objectif d'améliorer le choix du cœur victime effectué par la fonction `construct_core_set`. Cette heuristique est basée sur le constat fait dans le chapitre 1 qui montre que l'utilisation efficace des caches du processeur a un impact important sur les performances des applications. Dans le chapitre 1, nous avons montré que les architectures multi-cœurs proposent généralement des caches partagés entre les cœurs d'un même processeur. Nous avons notamment vu que sur un processeur Intel Xeon à 4 cœurs, les cœurs sont divisés en groupes de deux cœurs et chaque cœur d'un groupe possède son cache L1 privé et partage un cache L2 avec le second cœur du groupe. Dans la seconde architecture que nous avons étudiée, une architecture AMD composée de 4 processeurs de 4 cœurs chacun, les 4 cœurs possèdent chacun un cache L1 et un cache L2 privés et partagent un cache L3 de taille importante. Enfin, sur ces architectures, le temps d'accès au cache partagé est généralement bien inférieur au temps d'accès à la mémoire. Par exemple, sur une architecture Intel à 8 cœurs, il est 17 fois plus coûteux d'accéder à la mémoire que d'accéder à une donnée dans le cache L2 partagé.

Il devient ainsi extrêmement important de prendre en compte la hiérarchie mémoire de la machine dans l'algorithme de vol de tâches. Le coût d'un vol dépend en effet fortement de la *distance* entre le cœur voleur et le cœur victime. Ces deux cœurs peuvent soit partager un cache, soit être sur le même processeur, soit être sur des processeurs différents (et même éventuellement sur des nœuds différents dans le cas d'une architecture NUMA).

L'heuristique de vol *VSAL* présentée dans cette section a pour objectif de minimiser le nombre de fautes de cache et donc leur impact sur la performance des applications. Pour cela, la fonction construisant la liste ordonnée de cœur à voler (`construct_core_set`) retourne dans ce cas une liste de cœurs ordonnée par leur distance par rapport au cœur voleur. Cette information peut être obtenue soit grâce à des primitives proposées par le système d'exploitation, soit mesurée au démarrage de l'environnement d'exécution.

Prendre en compte la hiérarchie des accès mémoire lors du vol de tâches présente deux avantages. Le premier est de réduire le coût du vol de tâches en manipulant des données déjà en cache. Ces données sont les structures internes de l'environnement d'exécution comme les files d'événements et les informations associées (verrou, taille de la file, etc.). Le second avantage est de réduire le nombre de fautes de caches lors du traitement des événements volés. En effet, déplacer des événements implique que les données manipulées par le traitement ces événements vont également devoir être déplacées d'un cache à l'autre lors du traitement de l'événement.

5.2.2 Vol de tâches avec analyse de la rentabilité

Comme nous l'avons montré dans la section 5.1, déplacer un événement d'un cœur à un autre a un coût important. Ce coût vient majoritairement de la prise de verrou nécessaire à la fois sur la file du cœur victime ainsi que sur celle du cœur voleur. L'heuristique *VAR (Vol de tâches avec Analyse de la Rentabilité)* a pour objectif de déterminer si un cœur est une bonne victime ou non. Pour cela, l'idée

est d'intégrer dans l'algorithme de choix de la victime, le temps de traitement associé aux événements qui peuvent être volés.

Dans un système sans verrou, voler une petite quantité de travail est intéressant car cela n'a pas d'impact sur la performance du cœur sur lequel le vol est effectué. Inversement, dans un système utilisant des verrous pour protéger les files d'événements, le vol d'une couleur peut ralentir le cœur victime à cause de la contention sur les verrous. Par conséquent, il est nécessaire d'éviter de voler trop fréquemment et donc de ne voler que des couleurs dont le temps de traitement est suffisamment important.

Plus précisément, l'heuristique *VAR* consiste à classer les couleurs en deux ensembles : l'ensemble des couleurs qu'il peut être intéressant de voler et celui des couleurs dont le vol est à éviter (car il n'est pas rentable). Nous classons ici les couleurs et non les événements, car un vol concerne toujours l'intégralité des événements d'une couleur. Nous définissons une couleur comme étant intéressante à voler lorsque le temps total de traitement de tous les événements de cette couleur est supérieur au temps nécessaire pour voler l'ensemble des événements de cette couleur.

Pour cette heuristique, la fonction `can_be_stolen` est modifiée de façon à retourner « vrai » uniquement lorsqu'une couleur intéressante à voler est présente dans la file d'événements du cœur victime. Pour mettre en place cette heuristique, il est nécessaire de connaître le temps de traitement moyen de chaque type d'événement. Cette connaissance peut être acquise automatiquement par une profilage dynamique effectué par le support d'exécution. Cependant, une observation systématique peut introduire des surcoûts non désirés. Une autre solution est d'effectuer une première exécution de l'application avec l'observation du temps de traitement des événements puis d'annoter le code des traitants d'événements avec leur temps estimé. L'avantage de cette méthode est de pouvoir donner des temps différents à des événements associés au même traitant. Par exemple, dans un serveur Web, le temps estimé du traitement d'un événement peut être fonction de la taille du fichier accédé.

5.2.3 Vol de tâches avec pénalité

Les deux heuristiques présentées précédemment ont pour objectif d'améliorer le choix du cœur victime pour réduire les fautes de cache et assurer que le vol sera bénéfique. Même dans le cas où un vol est bénéfique, le choix de la couleur à voler peut avoir un impact sur les performances. L'heuristique *VAP (Vol de tâches Avec Pénalité)* a pour objectif d'améliorer ce choix. Concrètement, l'heuristique *VAR* construit une liste des couleurs qu'il est intéressant de voler et l'heuristique *VAP* permet de faire un choix dans cette liste. Ce choix est effectué en prenant en compte le profil mémoire des traitants d'événements associés à chaque couleur, actuellement fourni par le concepteur de l'application.

L'idée principale du vol de tâches avec pénalité peut être décrite comme suit. Les événements dont le traitement provoque un accès à un ensemble de données peu volumineux sont de bons candidats pour le vol de tâches, puisque leur déplacement ne déclenchera que peu le mécanisme de cohérence de cache. Les autres traitants d'événements demandent une analyse plus poussée de leur profil d'accès mémoire. Si les données manipulées ont un temps de vie très faible, par exemple si un traitant alloue un grand volume de mémoire et la libère avant sa terminaison, le vol des événements associés à ce traitant permettra d'améliorer le parallélisme et n'augmentera pas le nombre de fautes de cache causées par ce traitant. Inversement, les traitants manipulant de grandes quantités de données avec une durée de vie importante (par exemple lorsque la donnée est passée par référence entre plusieurs traitants) sont de moins bons candidats pour le vol de tâches. En effet, voler de tels événements risque d'augmenter fortement le nombre de fautes de caches puisqu'il sera nécessaire de faire migrer les données associées.

L'heuristique de vol de tâches *VAP* permet au programmeur d'associer des pénalités aux traitants

d'événements. Les événements associés à des traitants sur lesquels une pénalité importante a été fixée auront moins de chance de se faire voler que les autres. Ce mécanisme permet de réduire l'attractivité d'un événement pour le mécanisme de vol de tâches. Dans l'état actuel de nos travaux, ces annotations sont fixées par le programmeur en utilisant les informations de profilage fournies par l'environnement d'exécution, ainsi que sa connaissance du comportement des traitants. Une hypothèse principale de ce mécanisme est qu'un événement a un temps d'exécution et des profils d'accès mémoire relativement stables. Cette hypothèse est raisonnable pour deux raisons : (i) les tâches considérées sont de petite granularité, et (ii) les effets combinés des deux premières heuristiques présentées (*VSAL* et *VAR*) permettent de limiter les fluctuations dans les profils de fautes de caches des tâches.

5.3 L'environnement d'exécution Mely

Dans cette section, nous présentons l'environnement d'exécution *Mely (Multi-core Event LibrarY)*. Mely est un environnement d'exécution pour la programmation événementielle multi-cœur basée sur la coloration d'événements. Mely a été conçu avec pour objectif de réduire au maximum le coût intrinsèque du vol de tâches et permet de mettre en œuvre les trois heuristiques de vol précédemment décrites. Mely est rétrocompatible avec les applications conçues pour Libasync-smp. Toutefois, les structures utilisées pour le stockage et la gestion des événements, ainsi que la mise en œuvre du mécanisme de vol de tâches sont différentes. Dans la suite de cette section, nous débutons par la description de l'architecture de Mely. Nous décrivons ensuite l'implantation du mécanisme de vol de tâches. Enfin, nous fournissons des détails sur les choix d'implantation effectués.

5.3.1 Architecture de Mely

De façon similaire à Libasync-smp, chaque cœur possède son thread responsable de récupérer les événements dans une file et de les traiter. Cependant, Mely modifie la façon dont sont stockés les événements au sein de l'environnement d'exécution. En effet, le stockage des événements au sein de la même file induit des coûts important lors du vol. Cela nécessite de parcourir la file pour trouver tous les événements d'une couleur donnée. Pour réduire de manière conséquente le coût des différentes fonctions du mécanisme de vol de tâches, notamment `construct_event_set`, Mely regroupe les événements de la même couleur dans une file d'événements. Ainsi, contrairement à Libasync-smp, Mely utilise une file par couleur, appelée `color-queue`.

Chaque cœur est responsable d'un ensemble de couleurs et maintient donc une liste de `color-queue`. La liste de `color-queue` est une liste doublement chaînée appelée `core-queue`. Tout comme Libasync-smp, il est possible de définir des priorités pour les événements. En fait, la priorité est fixée par `color-queue` et correspond à la priorité maximale des événements associés à la couleur. Au sein d'une `core-queue`, les `color-queue` sont triées par priorité. Cette solution fonctionne bien si les événements d'une même couleur ont des priorités similaires. Cette hypothèse est raisonnable dans le cas des serveurs de données. En effet, dans de telles applications, le parallélisme est souvent effectué entre les clients et ainsi les couleurs sont fixées par clients. Les différentes opérations d'un client ont généralement la même priorité. Ces dernières peuvent être utilisées pour différencier un client d'un autre client en fonction de critères de qualité de service ou pour favoriser une opération spécifique, telle que la vérification des descripteurs de fichiers actifs, par rapport aux autres opérations. Par conséquent, le fonctionnement que nous avons adopté pour les priorités est adapté pour les serveurs de données.

La figure 5.7 illustre l'architecture sur un cœur de l'environnement d'exécution Mely. La notion de `stealing-queue` est décrite dans la section 5.3.2.

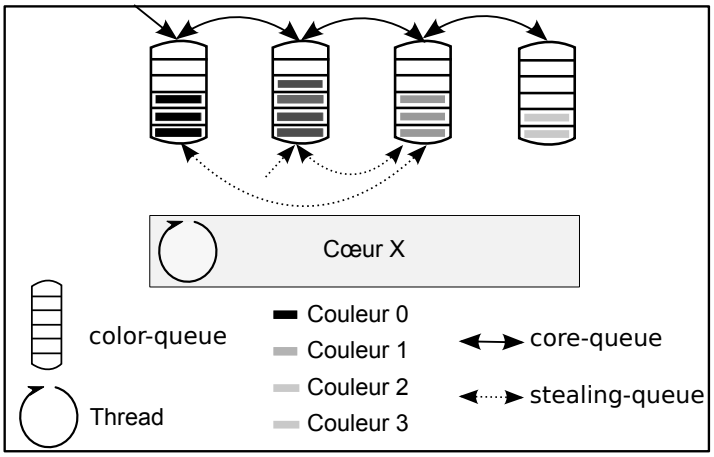


Figure 5.7 – Architecture de Mely sur un cœur.

Avec cette nouvelle organisation en files d'événements par couleur, lorsqu'un cœur veut choisir le prochain événement à traiter, il va simplement prendre l'événement en tête de la première `color-queue` de sa `core-queue`. Cela rend relativement simple l'ordonnancement. Pour éviter la famine, un cœur n'est pas autorisé à traiter indéfiniment des événements d'une seule couleur. Pour assurer cela, nous fixons une limite sur le nombre d'événements pouvant être traités successivement pour une couleur donnée. Lorsque la limite est atteinte, la file de cette couleur est placée en queue de la liste du cœur et la prochaine couleur dans la file devient la couleur active. Cette optimisation nous permet de traiter les couleurs par lot et ainsi d'améliorer la localité des caches. Dans nos expérimentations, la limite a été fixée empiriquement à 10 événements. Lorsqu'une couleur n'a plus d'événements, et donc que sa file est vide, la `color-queue` associée est retirée de la file du cœur concerné.

Lorsqu'un nouvel événement est produit, le cœur producteur doit tout d'abord récupérer la `color-queue` adéquate dans laquelle l'événement doit être inséré. Pour cela, Mely utilise un tableau effectuant le lien entre les couleurs et leur `color-queue` associée. Notons que ce tableau est également présent dans Libasync-smp pour savoir combien d'événements sont liés à une couleur donnée. Une `color-queue` est une liste chaînée d'événements ainsi que des informations notamment sur le nombre d'événements présents dans la file. Cette structure est allouée uniquement lors de la production du premier événement associé à la couleur. Enfin, lorsqu'un événement d'une couleur est produit, soit la couleur est déjà présente dans la `core-queue` concernée, soit elle ne l'est pas. Dans le deuxième cas, elle est insérée. Dans tous les cas, la priorité de la couleur est remise à jour et la file est déplacée si nécessaire pour garder une `core-queue` triée.

Les accès aux différentes files doivent être faits en exclusion mutuelle. Pour cela, comme Libasync-smp, Mely utilise un verrou par cœur. Ces verrous, qui nécessitent une attente active, sont utilisés par les différents cœurs lors de la production, de la consommation, ou encore du vol d'un événement. Il n'est pas possible d'utiliser un verrou par couleur pour réduire la contention. En effet, il est également nécessaire de garantir l'exclusion mutuelle lors de l'insertion ou de la suppression d'une `color-queue` dans une `core-queue`, ainsi que pour la mise à jour du nombre d'événements présents sur le cœur. Il est important de rappeler qu'un environnement d'exécution pour la programmation événementielle colorée ne peut pas se baser sur des structures sans verrous pour la gestion des événements, comme les structures DEqueue [30]. En effet, ces structures se basent sur le fait qu'un thread ne peut produire des tâches que dans sa propre file. Dans l'approche événementielle colorée, plusieurs threads peuvent produire des événements simultanément sur un cœur donné.

L'avantage de cette nouvelle architecture est de fournir des opérations de vol de tâches en temps constant. En effet, pour effectuer un vol, il faut (i) choisir une couleur à voler, ce qui revient à choisir la seconde couleur la file du cœur puisque la première est la couleur en cours d'exécution, (ii) supprimer les liens entre la file de la couleur et la `core-queue`, ce qui consiste à changer 4 pointeurs, et (iii) réinsérer la file dans la liste du cœur voleur.

5.3.2 Mise en œuvre du vol de tâches

L'implantation du mécanisme de vol de tâches dans Mely est basée sur celle effectuée dans Libasync-smp, étendue avec les heuristiques *VAR*, *VSAL* et *VAP*. Dans cette section, nous décrivons la manière dont nous avons mis en œuvre ces trois heuristiques.

5.3.2.1 Vol de tâches sensible à la localité

L'implantation de cette heuristique est très simple : la fonction `construct_core_set` construit une liste de cœurs à voler triée en fonction de la hiérarchie mémoire. Pour cela, nous utilisons la réification de l'architecture proposée par le noyau Linux et accessible depuis le pseudo-système de fichiers `/sys`. Plus précisément, Mely construit sa propre cartographie mémoire au démarrage de l'environnement d'exécution, ce qui permet à chaque cœur de découvrir ses voisins. Cette cartographie est stockée dans une `HashMap` qui fournit un accès rapide en *O(1)* à la liste des cœurs voisins d'un cœur donné. Le cœur voleur essaie donc d'abord de voler ses cœurs voisins et ensuite, si aucun cœur voisin est une bonne cible, les cœurs distants. Notons que la version actuelle de notre prototype ne gère qu'un seul niveau de cache partagé entre les cœurs.

5.3.2.2 Vol de tâches avec analyse de la rentabilité

L'implantation de cette stratégie consiste à introduire un nouveau chaînage entre les files de couleurs. Ce chaînage, appelé `stealing-queue`, est utilisé par cœur pour référencer les couleurs dont le vol est considéré comme rentable. À l'intérieur d'une `stealing-queue`, les couleurs sont ordonnées en fonction du temps de traitement total estimé de leurs événements. En fait, pour réduire les coûts d'insertion, la liste est seulement partiellement ordonnée. La file est découpée en trois tronçons : les couleurs très rentables, celles moyennement rentables, et celles tout juste rentables. Au sein d'un tronçon, les éléments ne sont pas ordonnés. Cette organisation partiellement ordonnée permet d'obtenir une insertion en *O(1)*. Ceci est particulièrement important, car les insertions et les retraits peuvent être fréquents. Notons que cette liste est indépendante de la `core-queue` et peut ne pas référencer toutes les couleurs d'un cœur. La figure 5.7 donne un exemple de `stealing-queue`. Dans cet exemple, seules trois couleurs sur les quatre sont référencées et l'ordre des couleurs dans cette liste est différent de l'ordre d'exécution.

Cette nouvelle liste est utilisée lors du vol pour voler une couleur la plus rentable possible. Trouver une couleur à voler a ainsi un coût constant puisqu'il s'agit de regarder si une couleur est présente dans l'un des trois tronçons et de prendre la première du tronçon le plus rentable.

Lorsqu'un nouvel événement est inséré dans la `color-queue` correspondante, le temps de traitement cumulé de tous les événements de la couleur est augmenté du temps de traitement estimé de l'événement. De manière symétrique, lorsqu'un événement est retiré de la `color-queue` correspondante, le temps de traitement cumulé de tous les événements de la couleur est décrémenté. Lorsqu'une couleur devient rentable à voler, la *color-queue* correspondante est insérée dans la file du cœur. L'opération inverse est effectuée lorsqu'une couleur ne devient plus rentable. Enfin, lorsque sa rentabilité évolue, une couleur peut être déplacée d'un tronçon à un autre.

Comme nous l'avons expliqué dans la section 5.2, dans l'état actuel de nos travaux, le temps de traitement estimé de chaque traitant est fourni par le programmeur après une première phase d'analyse de l'application. Cette analyse est obtenue grâce à des outils internes de l'environnement accessibles au programmeur pour surveiller l'état de l'application. Le tableau 5.4 présente les modifications apportées à l'API pour prendre en compte ce besoin. Nous pouvons noter que les temps sont associés aux événements plutôt qu'aux traitants. Cette caractéristique permet éventuellement de différencier deux événements liés à un même traitant. Par exemple, sur un serveur Web, cela permet de fixer le temps estimé en fonction de la page demandée. Dans l'état actuel, nous supposons que tous les événements associés au même traitant nécessitent le même temps de traitement, ce qui est une supposition raisonnable comme nous l'avons expliqué dans la section 5.2.

5.3.2.3 Vol de tâches avec pénalité

La mise en œuvre du vol de tâches avec pénalité nécessite de fournir au programmeur une API pour spécifier une pénalité sur les traitants d'événements. Le tableau 5.4 présente l'introduction d'une nouvelle fonction. Il s'agit ici d'affecter une pénalité à une couleur. Cette fonctionnalité est utile par exemple lorsque des traitants d'événements d'une couleur font des modifications sur une donnée statiquement allouée d'une taille importante. Cette pénalité est utilisée lors du calcul du temps de traitement cumulé d'une couleur. Lorsqu'un événement est inséré, le temps de traitement total de la couleur associé est augmenté de la valeur suivante : $\frac{durée_estimée_de_traitement}{pénalité_de_vol}$. Ainsi, une couleur avec une pénalité de vol très forte sera perçue comme nécessitant un temps de traitement inférieur à la réalité et, par conséquent, comme une moins bonne candidate pour l'algorithme de vol de tâches.

`événement =` **cwrap_timeleft** `(traitant, paramètres du traitant, couleur,` *temps estimé de traitement*`)`	Création d'un événement associé à une couleur en indiquant le temps estimé du traitement
`événement =` **cpwrap_timeleft** `(traitant, paramètres du traitant, couleur, priorité,` *temps estimé de traitement*`)`	Création d'un événement associé à une couleur et une priorité en indiquant le temps estimé du traitement
`set_penalty (couleur, pénalité)`	Associe une pénalité à une couleur

TABLE 5.4 – Évolution de l'API pour fournir au programmeur la possibilité de fixer le temps estimé d'un événement ainsi que la pénalité sur un traitant.

5.3.3 Détails supplémentaires de mise en œuvre

Dans cette partie, nous donnons des détails sur la mise en œuvre de Mely. Ces améliorations ont également été apportées dans Libasync-smp pour pouvoir comparer de manière équitable les deux environnements d'exécution.

Environnement logiciel utilisé. La version initiale de Libasync-smp a été évaluée par Zeldovich *et al.* en 2003 en utilisant un noyau Linux 2.4, la version 2.95 du compilateur GCC, ainsi que la version 2.3 de la bibliothèque Glibc. Les logiciels cités précédemment ont depuis largement évolué tant en termes de performances que de fonctionnalités. Nous avons donc modifié le code de Libasync-smp pour qu'il soit compatible avec les versions plus récentes de ces outils.

Mely est maintenant basé sur la version 4.3 du compilateur GCC. De nombreux changements ont été effectués dans la gestion des *templates* C++ pour rendre leur usage plus strict. Ce compilateur

amène également depuis sa version 4 de nouvelles optimisations de code. L'environnement d'exécution, ainsi que les applications sont compilés en utilisant le niveau *O2* d'optimisation. Mely est également compatible avec la version 2.7 de la bibliothèque *Glibc*. Cette compatibilité permet de bénéficier de l'implantation `NPTL` des threads `POSIX` ainsi que de la possibilité de fixer un thread sur un cœur donné en utilisant la fonction `pthread_setaffinity_np`. Cette dernière possibilité assure que le noyau ne sera pas tenté de déplacer un thread d'un cœur à un autre, anéantissant ainsi tous les efforts d'optimisation de l'environnement d'exécution. Cette fonctionnalité est disponible depuis la version 2.5.8 du noyau Linux, ce qui explique pourquoi elle n'était pas exploitée originellement dans Libasync-smp. Nous utilisons dans nos expériences un noyau Linux 2.6.24.

Gestion efficace des E/S non bloquantes. Pour améliorer la performance et le passage à l'échelle de la gestion des E/S, nous avons également remplacé le mécanisme de surveillance de l'activité sur une socket. Le mécanisme original est basé sur l'appel système `select` que nous avons remplacé par l'appel système `epoll` plus efficace. Nous configurons `epoll` pour le déclenchement de notifications d'activité réseau en mode *level*. Dans cette configuration, `epoll` permet de savoir que la socket est active tant que l'E/S n'a pas été traitée. Nous avons également un support expérimental du mode *edge*, dans lequel un événement d'E/S n'est reporté qu'une seule fois par `epoll`. Ce second mécanisme donne sensiblement les mêmes performances. Les gains de performances dus à l'utilisation de l'appel système `epoll` ont été observés dans le contexte des serveurs de données [56]. Pour conserver la compatibilité avec les applications existantes, nous avons conservé la même API que Libasync-smp utilisant `fdcb`.

Amélioration de la gestion de la mémoire. Tout d'abord, nous avons supprimé le mécanisme de ramasse-miettes de Libasync-smp. En effet, ce mécanisme est basé sur du comptage de références, c'est-à-dire qu'un compteur est incrémenté lorsqu'une fonction possède un lien vers un objet et décrémenté lorsqu'elle relâche ce lien. Comme les objets peuvent être liés à plusieurs cœurs, il est nécessaire que la manipulation du compteur soit atomique. Le mécanisme de comptage de références réduit donc ainsi sensiblement les performances du fait de ses nombreux appels à ces fonctions atomiques (qui peuvent de plus verrouiller le bus mémoire et réduire le passage à l'échelle même en l'absence de contention sur un même objet). Comme le mécanisme de ramasse-miettes est un frein important à la performance de Libasync-smp sur les architectures multi-cœurs, nous avons simplement choisi de gérer manuellement les libérations de zones mémoire. La mise en œuvre d'un ramasse-miettes optimisé pour les architectures multi-cœurs nous semble être un défi important, que nous laissons pour des travaux futurs.

Nous avons vu dans la section 3.2.3 que la gestion de la mémoire a un impact important sur la performance ainsi que sur le passage à l'échelle des applications. Le premier problème considéré est celui du faux partage. Ce problème, qui survient lorsque deux cœurs manipulent deux données disjointes situées sur la même ligne de cache, limite les performances de l'application sur les architectures multi-cœurs. Nous avons donc considéré le placement des données statiques en utilisant la méthode de remplissage artificiel de lignes de cache. Pour réduire la consommation mémoire, nous groupons dans une même ligne de cache les données d'un cœur. Libasync-smp utilise également des stratégies de remplissage de ligne de caches. Cependant, de façon étonnante, le remplissage est fait de telle façon que du faux partage est toujours possible et dix lignes de caches par coeur sont utilisées pour toutes les données statiques. En remplissant de façon systématique les données des cœurs et en les regroupant, nous éliminons le faux partage et n'utilisons qu'une ligne de cache par cœur.

Nous avons également choisi d'utiliser un allocateur mémoire spécialisé pour les architectures multi-cœurs : l'allocateur utilisé dans l'environnement d'exécution TBB. Comme décrit dans la section 3.2.3, l'objectif de cet allocateur est de fournir une allocation mémoire passant à l'échelle en

réduisant la contention sur les primitives d'allocation et en évitant le faux partage. Pour cela, un allocateur mémoire pour architecture multi-cœurs utilise des zones d'allocation privées pour chaque thread.

Nous avons évalué le passage à l'échelle de Mely lorsque l'application est constituée de flots indépendants, sans changement de processeur. Pour cela, nous avons utilisé un micro-test simple dans lequel un traitant d'événement produit toujours un nouvel événement sur le même cœur. Nous regardons le nombre d'événements traités par seconde par cœur en fonction du nombre de cœurs. Idéalement, ce nombre doit rester stable. La figure 5.8 illustre les résultats obtenus avec un environnement d'exécution sans aucune optimisation, avec remplissage artificiel de lignes de cache et avec remplissage de lignes de caches et utilisation de TBB pour l'allocation mémoire. Elle affiche également le nombre d'accès (succès ou échecs) au cache L2 à 8 cœurs dans les différentes configurations. Les résultats présentés sont une moyenne sur plusieurs exécutions et l'écart type constaté est très faible.

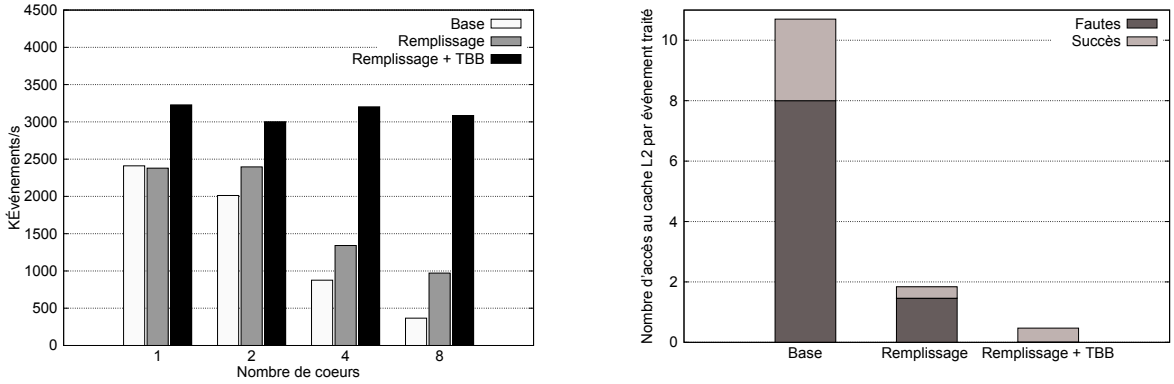


Figure 5.8 – Passage à l'échelle de Mely avec des flots d'événements indépendants et observation du nombre d'accès au cache L2 par événement à 8 cœurs.

Cette figure montre que remplir artificiellement les lignes de caches améliore les performances de l'environnement d'exécution en évitant le faux partage. En effet, le nombre d'accès au cache L2 est grandement réduit. Le faux partage peut être présent entre deux cœurs partageant un cache ou non. Dans le premier cas, le faux partage provoque une invalidation du cache L1 qui va par conséquent déclencher un accès en L2 pour récupérer de nouveau la donnée. Cet accès sera probablement un succès dans le cas de notre processeur puisque les caches sont inclusifs comme décrit dans le chapitre 1). Dans le second cas, le faux partage provoque une invalidation du cache L2 et éventuellement du cache L1, ainsi qu'un accès à la mémoire centrale pour récupérer de nouveau la donnée. Nous pouvons voir sur la figure que le passage à l'échelle n'est pas idéal, car il reste du faux partage. De plus, les verrous par *arenas* (voir section 3.2.3) empêchent le passage à l'échelle. Nous avons mesuré que la fonction de `malloc` prend trois fois plus de temps avec 8 cœurs actifs qu'avec un seul. L'utilisation de TBB résout le problème. Dans cette configuration, l'environnement est capable d'afficher un passage à l'échelle idéal sur des flots indépendants grâce à deux raisons principales : (i) le temps d'un `malloc` est constant quelque soit le nombre de cœurs actifs, et (ii) il n'y a presque plus de fautes de cache L2, car le faux partage est quasiment éliminé. Notons que pour assurer l'équité de nos comparaisons, nous avons également vérifié que ces améliorations produisent les mêmes effets dans Libasync-smp.

L'utilisation conjuguée d'une allocation mémoire efficace pour les architectures multi-cœurs, ainsi que de la technique de remplissage de ligne de cache permet à l'environnement d'exécution d'afficher un passage à l'échelle idéal pour les applications utilisant des flots indépendants. Dans ce cas précis, l'isolation entre les flots est similaire à ce que propose la méthode *N-Copy* tout en res-

tant plus flexible car il est possible de partager simplement des descripteurs de fichiers ou des zones mémoire entre les cœurs.

5.4 Évaluation du mécanisme de vol de tâches de Mely

Dans cette section, nous évaluons l'environnement d'exécution Mely et tout particulièrement le mécanisme de vol de tâches. Nous décrivons tout d'abord notre environnement de test. Nous présentons ensuite les micro-tests utilisés pour évaluer individuellement chacune des heuristiques proposées. Enfin, nous évaluons les performances de Mely et de son vol de tâches sur deux serveurs de données réalistes : un serveur Web et SFS, un serveur de fichiers sécurisé.

5.4.1 Conditions expérimentales

Nos expériences sont menées sur une machine composée de deux processeurs Intel Xeon E5410 *Harpertown* à 4 cœurs. Chaque processeur est composé de 4 cœurs à une fréquence de 2.33 GHz et les cœurs sont groupés par paires. Les cœurs d'un groupe partagent un cache L2 de 6 Mo. Les temps d'accès mémoires sont uniformes entre les cœurs. Plus d'informations sur ce processeur ont été données dans le chapitre 1. La machine est également équipée de 8 Go de mémoire et de 8 cartes réseau à 1 Gb/s.

Pour les expérimentations sur les serveurs de données, nous utilisons entre 8 et 16 machines jouant le rôle de clients. Ces machines sont équipées d'un processeur bi-cœurs Intel T2300. Les machines sont connectées avec le serveur avec des liens Ethernet à 1 Gb/s via un commutateur non bloquant.

La configuration de la partie logicielle correspond à celle décrite en section 5.3.3. Chaque test a été exécuté plusieurs fois et les variations observées sont faibles (moins de 1 %).

5.4.2 Micro-tests

Nous utilisons un ensemble de micro-tests pour évaluer la performance de l'environnement d'exécution Mely. Nous évaluons tout d'abord les performances de l'algorithme de vol de tâches de base de Mely, c'est-à-dire le mécanisme de vol de tâches défini dans Libasync-smp. Nous étudions ensuite l'impact des trois heuristiques de vol que nous avons présentées dans les parties précédentes.

5.4.2.1 Algorithme de base du vol de tâches

Pour évaluer les gains de performances dus à un placement efficace des données et à une nouvelle structure pour les files de messages, nous comparons les performances de Mely à celles obtenues par Libasync-smp en activant puis en désactivant le mécanisme de vol de tâches. Pour cela nous utilisons un micro-test dans lequel les événements sont produits de manière déséquilibrée pour déclencher le mécanisme de vol de tâches. Ce micro-test est construit selon un modèle *fork-join* : à chaque tour, 50000 événements sont produits sur le premier cœur. 98% de ces événements ont un temps de traitement très faible (100 cycles) et les autres ont un temps de traitement important (entre 10 et 50 Kcycles). Les événements sont indépendants, c'est-à-dire qu'ils ont des couleurs différentes et peuvent ainsi être traités de façon concurrente. Lorsque tous les événements ont été traités, un nouveau tour commence. Nous répétons cette opération pendant 5 secondes et mesurons le nombre d'événements traités par seconde.

Les résultats obtenus sont présentés dans le tableau 5.5. Ce micro-test met en évidence les mauvais résultats du mécanisme de vol de tâches de Libasync-smp lorsque la charge n'est pas équilibrée.

Configuration	KÉvénements/s	Temps passé à acquérir les verrous	Coût moyen du vol (cycles)
Libasync-smp sans vol de tâches	1310	0.93 %	-
Libasync-smp avec vol de tâches	122	39.73 %	28329
Mely sans vol de tâches	1265	0.89 %	-
Mely avec vol de tâches	1195	1.42 %	2261

TABLE 5.5 – Impact du mécanisme de vol de tâches de Mely sur les performances.

En particulier, nous pouvons voir que sur un cœur, la durée moyenne nécessaire pour voler les événements d'une couleur est de 28 Kcyles. Cela est à mettre en correspondance avec le temps de traitement moyen volé qui est de 484 cycles. De plus, ces mesures montrent qu'une majeure partie du temps est passée à acquérir un verrou sur la victime. Par conséquent, la mise en œuvre du vol de tâches de Libasync-smp dégrade fortement les performances de 90 %.

Ce micro-test montre également que Mely réduit drastiquement l'impact du vol de tâches sur les performances. L'architecture de Mely permet de diviser le temps nécessaire pour un vol par 12. Cependant, nous pouvons voir que, même avec une architecture améliorée, Mely réduit les performances de 5.5 %. Ce résultat met bien en évidence la nécessité d'avoir des heuristiques de vol plus intelligentes.

5.4.2.2 Vol de tâches avec analyse de la rentabilité

Pour évaluer l'heuristique *VAR*, nous avons utilisé le micro-test décrit dans la section précédente. Ce micro-test est adapté, car certains événements sont rentables à voler (les événements nécessitant beaucoup de temps de traitement) et d'autres ne le sont pas (les événements nécessitant peu de temps de traitement). Pourtant, les petits événements sont majoritaires, ce qui rend leur vol plus probable. Nous mesurons le débit d'événements traités par seconde lorsque différents algorithmes de vol sont utilisés. Nous comparons notamment Mely avec le vol de tâches de base avec la version du vol dans laquelle l'heuristique d'analyse de la rentabilité est activée. Les résultats sont présentés dans le tableau 5.6. Nous pouvons voir que l'heuristique *VAR* permet une amélioration de performance de 70% par rapport à Mely avec le vol de tâches de base. Comme le montre ce tableau, ces résultats s'expliquent par la quantité de travail obtenue (en nombre de cycles) à chaque vol qui est nettement plus importante puisque seuls les événements rentables sont volés.

Configuration	KÉvénements/s	Temps moyen de traitement volé (cycles)
Libasync-smp sans vol de tâches	1310	-
Libasync-smp avec vol de tâches	122	484
Mely avec vol de tâches de base	1195	445
Mely avec vol de tâches *VAR*	2042	49987

TABLE 5.6 – Impact de l'heuristique de vol VAR sur la performance du vol de tâches.

5.4.2.3 Vol de tâches avec pénalité

Nous avons ensuite utilisé un second micro-test pour évaluer l'heuristique de vol VAP. Ce micro-test utilise deux types d'événements *A* et *B* associés à des traitants distincts. Au démarrage de l'application un seul cœur possède des événements de type *A* et les autres démarrent avec une file vide. Le traitement d'un événement de type *A* produit un événement de type *B* de la même couleur. De plus,

le traitement d'un événement de type *A* créé un tableau dont la taille tient dans le cache L1 du cœur. Chaque événement de type *B* créé effectue une opération sur une partie de ce tableau et enregistre un nouvel événement de type *B* pour traiter la suite. Cette opération est répétée tant que tout le tableau n'a pas été traité. Initialement, les événements de type A ont tous une couleur différente. Avec ce micro-test, chaque cœur exécute un ensemble d'événements de la même couleur qui accèdent au même tableau. Lorsque tous les événements ont été traités, un nouveau tour recommence.

L'idée de ce micro-test est la suivante. Les cœurs n'ayant pas de travail vont essayer de voler des tâches. Comme il y a plus d'événements de type *B* dans le système, il est plus probable de voler un événement de ce type. Toutefois, voler ce type d'événements est moins intéressant que de voler des événements de type *A*. En effet, voler un événement de type *B* implique de déplacer sa continuation, engendrant ainsi des fautes de caches et potentiellement du faux partage, du fait de l'allocation par cœur de TBB.

Il est important de considérer que la pénalité de vol ne peut s'appliquer que sur des événements définis comme rentable à voler par l'heuristique *VAR*. Dans ce test, il est rentable de voler des événements de type *B*, car cela augmente le parallélisme, mais il est plus rentable de voler des événements de type *A* car ceux-ci ne provoquent pas de fautes de cache.

Pour ce micro-test, nous mesurons le nombre total d'événements traités par seconde. Les résultats sont présentés dans le tableau 5.7. La pénalité des événements a été fixée à 1000. Nous observons tout d'abord que le vol de tâches de Libasync-smp dégrade les performances. Cependant, nous pouvons constater sur ce test que le vol de tâches peut être bénéfique puisque le vol de tâche de base de Mely améliore les performances. Enfin, l'heuristique *VAP* permet d'augmenter les performances de 53 % par rapport à la version de base du vol de tâches dans Mely. Ces résultats peuvent être expliqués par l'observation du nombre de fautes de cache par événement. En effet, le vol de tâches avec priorité permet de répartir la charge tout en réduisant le nombre de fautes de cache. Le nombre de fautes de cache diminue nettement (95 %) par rapport à la version de base du vol de tâches de Mely.

Configuration	**KÉvénements/s**	**Fautes de cache L2 / Événement**
Libasync-smp sans vol de tâches	1103	29
Libasync-smp avec vol de tâches	190	167K
Mely avec vol de tâches de base	1386	42K
Mely avec vol de tâches *VAP*	2122	2K

TABLE 5.7 – Impact de l'heuristique de vol VAP sur les performances du vol de tâches.

5.4.2.4 Vol de tâches sensible à la localité

Nous avons évalué l'heuristique *VSAL* à l'aide d'un troisième micro-test. Ce micro-test utilise un modèle de production de type *fork-join*, c'est-à-dire que lorsque tous les événements ont été traités, un nouveau tour est relancé. L'objectif de ce micro-test est de montrer l'intérêt que peut avoir la prise en compte de la localité sur les performances des applications. Pour cela, nous utilisons trois types d'événements que nous nommerons *A*, *B* et *C*. À chaque tour, un cœur de chacune des paires de cœurs partageant un cache L2 reçoit une centaine d'événements de type *A*. Le traitement de ces événements consiste en l'allocation d'un tableau tenant dans un cache L1 et enregistre deux événements de type *B* avec des couleurs disjointes (affectées initialement au même cœur). Le premier événement va trier la première partie du tableau tandis que le second événement va trier la seconde partie du tableau (cela simule le début d'un tri par fusion). Lorsque chacun des événements a fini de trier sa partie de tableau, il enregistre un événement de type *C* permettant de faire une synchronisation. Les deux événements

de type *C* produits ont donc la même couleur. Lorsque les deux événements de type *C* ont été reçus, la dernière partie du tri s'opère.

Les résultats obtenus sur ce micro-test sont présentés dans le tableau 5.8. Ces résultats montrent que l'heuristique de vol de tâches prenant en compte la hiérarchie des caches permet de répartir la charge sur les cœurs sur lesquels il n'y a aucun événement initialement, tout en assurant que les tableaux manipulés par ces traitants restent dans le cache partagé. Nous pouvons voir que la stratégie de vol en regardant uniquement le nombre d'événements dans la file des cœurs peut se révéler inefficace pour deux raisons. Premièrement, comme le montre le tableau 5.8, elle augmente grandement le nombre de fautes de caches par événement traité. En effet, initialement toutes les files contiennent le même nombre d'événements. Par conséquent, les cœurs voleurs vont choisir le cœur 0 comme victime de leur vol, puisque c'est le premier cœur parmi ceux qui ont le plus d'événements. Par conséquent, seul le cœur 1 peut bénéficier de la localité de cache lors du vol. Deuxièmement, pour la même raison, la contention est plus forte sur le verrou du cœur 0 puisque c'est le cœur le plus sollicité. Le vol de tâches prenant en compte la hiérarchie mémoire permet de réduire cette contention et d'améliorer la localité des caches, puisque le nombre de fautes de cache est réduit de 83 % par rapport au vol de tâches de base de Mely.

Configuration	**KÉvénements / s**	**Fautes de cache L2 / Événement**
Libasync-smp sans vol de tâches	1156	0
Libasync-smp avec vol de tâches	1497	13
Mely - avec vol de tâches de base	1426	12
Mely - avec vol de tâches *VSAL*	1869	2

TABLE 5.8 – Impact de l'heuristique de vol VSAL sur les performances du vol de tâches.

5.4.3 Serveurs de données

Après avoir évalué individuellement nos propositions sur des micro-tests, nous les évaluons sur deux serveurs de données réalistes. Le premier est un serveur Web, nommé *SWS*, composé principalement de traitants d'événements relativement courts. Notre second cas d'étude est le serveur de fichiers sécurisé SFS [69]. Contrairement au serveur Web, ce serveur de fichiers passe la majeure partie du temps à exécuter des opérations cryptographiques très coûteuses. Dans ces deux cas, nous comparons Mely en activant le vol de tâches (ainsi que toutes les heuristiques) avec Libasync-smp en activant puis en désactivant le vol de tâches. Enfin, si le vol de tâches améliore les performances, nous comparons les performances de Mely avec et sans vol de tâches pour vérifier que les gains ne viennent pas uniquement de l'architecture de Mely.

5.4.3.1 Serveur Web SWS

Architecture. SWS est un serveur Web capable de servir un contenu statique. Il supporte un sous-ensemble de la norme HTTP/1.1, notamment la méthode `GET` ainsi que les connexions persistantes et la gestion des requêtes en pipeline. Pour améliorer les performances du serveur Web et éviter au maximum les E/S vers le disque, SWS construit les réponses lors de son lancement. Cette optimisation est inspirée des travaux effectués sur la performance du serveur Web Flash [75]. Pour chaque page, la réponse est stockée dans une table de hachage indexée avec le nom de la page. Cette méthode réduit sensiblement les appels au système d'exploitation, ainsi que les copies mémoire. Nous fournissons également un support de la méthode d'envoi sans copie (`sendfile`). Nous n'utilisons pas dans notre évaluation cette possibilité. Nous avons cependant vérifié que les performances sont similaires avec

et sans utilisation de `sendfile`. Enfin, pour ressembler autant que possible à un serveur Web de production, SWS prend en charge les requêtes erronées, c'est-à-dire mal formées ou concernant un fichier n'existant pas.

L'architecture du serveur Web SWS est similaire à celle décrite par Zeldovich *et al.* dans leurs travaux sur l'environnement d'exécution Libasync-smp [102]. Contrairement à leur serveur Web qui utilise un cache de fichiers qui se remplit au fur et à mesure de l'accès aux différents fichiers, SWS charge tous les fichiers en mémoire au lancement. Cette hypothèse est raisonnable étant donnée la quantité de mémoire importante (8 Go) équipant notre serveur. Elle nous a permis de simplifier la grandement gestion du cache et d'éviter les changer de processeur lors des différentes phases de traitement d'une requête donnée. L'architecture de SWS est illustrée par la figure 5.9.

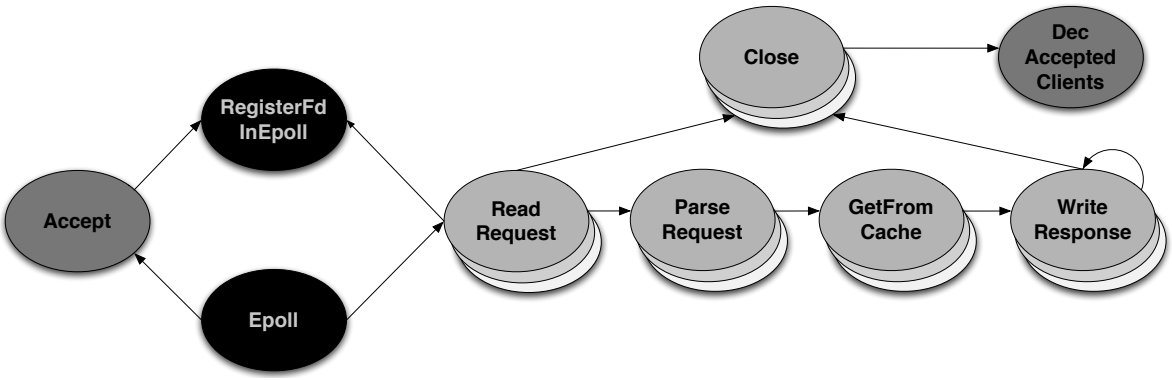


Figure 5.9 – Architecture du serveur Web SWS.

Comme nous pouvons le voir sur cette figure, SWS est structuré en 9 traitants d'événements. Les deux événements en noir, c'est-à-dire les événements liés à la surveillance des activités réseau en utilisant `epoll`, sont des événements générés par l'environnement d'exécution et ne sont pas visibles par le programmeur de l'application.

Les neuf types de traitants de SWS sont les suivants :

- Le traitant d'événement `Epoll` est responsable de la surveillance des activités de lecture ou d'écriture sur un ensemble de sockets enregistrées avec la fonction `fdcb`. Les deux événements gérés par `Epoll` sont les demandes de connexions et les réceptions de requêtes pour une connexion déjà ouverte. Par conséquent, `Epoll` peut enregistrer des événements d'acceptation de la connexion ou de lecture de la requête. L'événement associé à ce traitant est colorié initialement avec la couleur 0 (et par conséquent enregistré initialement sur le premier cœur).

- Le traitement d'événement `RegisterFdInEpoll` permet d'enregistrer un nouveau descripteur à surveiller. Comme le traitement de ces événements manipule la liste utilisée par le traitant `Epoll`, il est nécessaire de les colorier avec la couleur 0.

- Le traitant d'événement `Accept` est chargé d'accepter les nouvelles connexions. Comme pour les serveurs Web de production (de type Apache), il est possible de spécifier un nombre de clients maximum pouvant être servis simultanément. Pour cela, `Accept` garde un compteur de connexions acceptées. Ce compteur doit être manipulé en exclusion mutuelle, ce qui explique pourquoi tous les événements liés à ce traitant sont coloriés avec la couleur 1 et par conséquent placés initialement sur le second cœur. Lorsqu'une nouvelle connexion a été acceptée, le descripteur de fichier résultant est mis à surveiller grâce à un événement de type `RegisterFdInEpoll`. Ce traitant alloue les tampons nécessaires à la lecture de la requête.

- Le traitant `ReadRequest` est utilisé pour lire les requêtes HTTP sur la connexion. Une requête peut être lue en plusieurs fois. Lorsque l'intégralité d'une requête a été lue, un événement de type `ParseRequest` est posté et un nouveau tampon est alloué pour gérer les multiples requêtes successives (HTTP pipelining). Ce traitant est également chargé de détecter les fermetures de connexions. Dans ce cas, un événement de type `Close` est enregistré.

- Le traitant `ParseRequest` est chargé de déterminer les paramètres de la requête et notamment quelle est la page demandée. Cet événement produit un événement de type `GetFromCache`.

- Le traitant `GetFromCache` a pour rôle de récupérer la réponse construite au lancement de l'application en fonction du fichier demandé. Cet événement poste ensuite un événement de type `WriteResponse` avec pour paramètre la réponse à envoyer.

- Le traitant `WriteResponse` envoie la réponse au client. Cet envoi peut éventuellement se faire en plusieurs étapes si les tampons d'émission sont pleins. Pour cela, `WriteResponse` peut poster un événement de même type. Nous aurions également pu utiliser `Epoll` pour nous indiquer quand les tampons redevenaient utilisables. Toutefois, cette nécessité est rare et il est plus simple et moins coûteux de poster un événement du même type (pas de changement potentiel de cœur). Lorsque tout a été envoyé, un événement `Close` peut être généré si le serveur choisit de fermer la connexion.

- Le traitant `Close` est utilisé pour fermer une connexion et libérer les ressources allouées pour cette connexion (notamment les tampons). Notons que pour gérer la cohérence, il est nécessaire d'attendre qu'il n'y ait plus d'événements postés pour cette connexion en attente dans une file, avant de fermer réellement la connexion. En effet, dans le cas contraire, cet événement manipulerait soit un descripteur lié à une connexion fermée, soit au pire un descripteur lié à une nouvelle connexion, engendrant de graves incohérences. Après avoir fermé une connexion, un événement `DecClientAccepted` est enregistré.

- Le traitant `DecClientAccepted` est utilisé pour décrémenter le nombre de clients acceptés et éventuellement autoriser de nouveau les acceptations de connexion. Puisque ce traitant manipule des données partagées avec le traitant `Accept`, sa coloration est la même que ce dernier pour garantir la cohérence des données.

Pour assurer un maximum de parallélisme entre les requêtes, les traitants `ReadRequest`, `ParseRequest`, `WriteResponse` et `Close` sont coloriés en utilisant l'identifiant de la connexion, c'est-à-dire le numéro du descripteur de fichier. Par conséquent, les requêtes issues de clients différents peuvent être traitées en parallèle par différents cœurs.

Injection de charge. Nous avons développé un injecteur de charge événementiel. Cet injecteur est similaire dans son architecture à celui développé par Banga et Druschel [17]. L'injecteur est construit autour de deux threads, un pour l'émission des requêtes et un pour la réception des réponses. Nous effectuons une injection en boucle fermée [82], c'est-à-dire qu'un client envoie une requête puis attend la réponse avant d'en envoyer une nouvelle. La principale qualité de ce modèle d'injection de charge est d'être fidèle au comportement d'un client. Son principal défaut est d'avoir un nombre de clients fixe. Le modèle d'injection de charge en boucle fermée est utilisé dans des injecteurs reconnus tel que SPECWeb 2005 [86]. Pour distribuer l'injection sur plusieurs machines, nous utilisons un mode de communication maître / esclave. Le maître est utilisé pour synchroniser les phases d'injections sur

les machines, ainsi que pour collecter et fusionner les résultats. Chaque nœud esclave est responsable de la gestion d'un certain nombre de clients.

Nous avons évalué l'environnement d'exécution Mely sur le serveur Web SWS avec de petits fichiers de 1 Ko. À titre de comparaison, avec l'injecteur de charge reconnu SPECWeb 2005 [86], 95 % des accès sont faits sur le dossier contenant les images qui contient des fichiers allant de 40 octets à 5 Ko. Pour effectuer l'injection de charge, nous utilisons 8 machines physiques simulant au total entre 200 et 2000 clients. Chaque client virtuel se connecte au serveur et émet 150 demandes de fichiers pendant cette connexion. Cette opération est répétée pendant 30 secondes et nous effectuons 3 itérations. Les variations de performances observées sur les trois itérations sont faibles (toujours inférieures à 2 %).

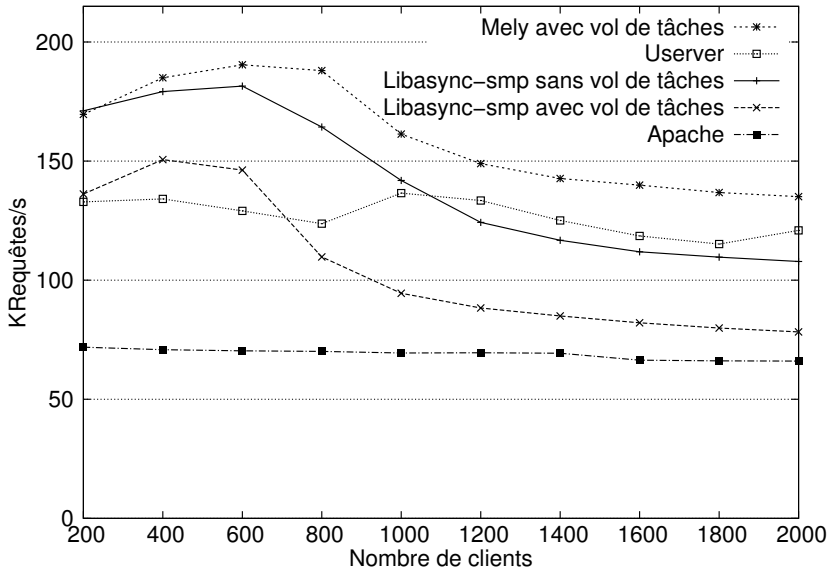


Figure 5.10 – Performance du serveur Web SWS utilisant Mely par rapport à la version Libasync-smp ainsi que d'autres serveurs Web connus.

Résultats. La figure 5.10 présente le débit observé avec l'environnement d'exécution Mely avec le vol de tâches et toutes ses heuristiques activées, ainsi qu'avec l'environnement d'exécution Libasync-smp avec et sans vol de tâches. Pour évaluer la performance absolue de SWS, nous comparons les performances de SWS avec celles de deux autres serveurs Web connus et efficaces : la version *worker* d'Apache [6] (c'est-à-dire la version utilisant un mélange de threads et de processus pour le parallélisme) et une version *N-Copy* du serveur Web événementiel *µserver* [76] [1].

Nous observons premièrement que SWS a de bonnes performances comparées aux autres serveurs Web. Ce résultat permet de confirmer que notre serveur Web est réaliste par rapport aux autres serveurs. Nous pouvons également observer que l'approche *N-Copy* a également de bonnes performances. Cependant, comme nous l'avons expliqué dans le chapitre 2, cette approche ne peut pas toujours être mise en œuvre. Enfin, nous pouvons voir que Mely à des performances constamment supérieures à celle de Libasync-smp avec et sans vol de tâches.

1. Le serveur Web µserver peut utiliser soit une architecture SYMPED (*SYmmetric Multiple Process Event-Driven*), qui consiste en *N* instances du serveur, soit une architecture shared-SYMPED, qui consiste également en *N* instances du serveur mais utilise de la mémoire partagée pour éviter la duplication des fichiers ouverts et du cache des entêtes associés. Dans nos expériences, ces deux architectures ont présenté des performances identiques.

Dans Libasync-smp, activer l'algorithme de vol de tâches fait baisser les performances sous cette charge jusqu'à 33 %. Comme expliqué dans la section 5.1, cette dégradation est due à deux facteurs principaux : (i) des coûts de vol très importants (197 Kcycles) qui sont largement supérieurs au temps moyen de traitement volé (20 Kcycles), et (ii) une augmentation importante du nombre de fautes de cache L2 (+146%) par rapport à la version de Libasync-smp sans vol de tâches.

Mely a des performances jusqu'à 73 % supérieures par rapport à Libasync-smp avec vol de tâches. Cela s'explique par le fait que Mely vole plus de temps moyen de traitement que Libasync-smp et est 32 fois plus rapide pour réaliser un vol (6 Kcycles en moyenne pour un vol). De plus, la prise en compte de la hiérarchie mémoire, ainsi que de la pénalité lors du vol permet de réduire le nombre de fautes de cache de 24% par rapport au vol de tâches de Libasync-smp.

Mely augmente également les performances jusqu'à 25 % par rapport à Libasync-smp sans vol de tâches. Le profil de répartition des événements sur les cœurs indique que le mécanisme de vol permet de délester le cœur chargé du traitant `Epoll` de la gestion des connexions et des requêtes. Par conséquent, le traitant `Epoll` est plus fréquemment exécuté, ce qui permet de fournir plus de travail aux autres cœurs.

Nous avons également mesuré la performance de Mely lorsque le mécanisme de vol de tâches est désactivé. Cette configuration a des performances légèrement plus faibles que Libasync-smp sans vol de tâches (entre -7 % et -20 %). Cette dégradation est principalement due au fait que le serveur Web utilise un grand nombre de couleurs avec une durée de vie faible. Par conséquent, cette utilisation des couleurs provoque fréquemment des opérations coûteuses d'insertion et de suppression des `color-queue` dans les `core-queue`. Le vol de tâches améliore ainsi les performances de l'environnement d'exécution Mely entre 7 % et 50 %, ce qui démontre son efficacité.

5.4.3.2 Serveur de fichiers sécurisé (SFS)

Architecture. SFS est un serveur de fichiers sécurisé réparti (de type NFS [77]). SFS est construit autour d'un serveur et de clients qui accèdent aux fichiers proposés par le serveur. Pour qu'il soit possible de manipuler un fichier réparti comme s'il s'agissait d'un fichier local, la partie cliente de SFS s'intègre avec le système de fichiers virtuel de Linux (*VFS, Virtual File System*). Par conséquent, un client voit un point de montage SFS comme un point de montage local. Pour assurer la sécurité des données, les communications entre un client et le serveur sont chiffrées et authentifiées. Les communications sont effectuées en utilisant une connexion TCP persistante.

SFS est composé de plusieurs centaines de types d'événements différents. Colorier *a posteriori* une application nécessite de connaître précisément les rôles et les implications de chacun des traitants. Ceci est difficile à faire sur SFS étant donné le nombre d'événements de type différents possibles. Par conséquent, nous avons choisi de faire la même coloration que les auteurs de Libasync-smp. Comme eux, nous avons pu constater que plus de 60 % du temps est passé à faire des communications de chiffrement et de déchiffrement. Nous avons donc choisi de ne colorier que les événements liés aux traitants effectuant ces opérations, soit 8 traitants. Comme pour le serveur Web, nous utilisons le descripteur de la connexion pour avoir du parallélisme entre les requêtes. Pour pouvoir effectuer en parallèle le déchiffrement d'une requête et le chiffrement d'une réponse pour une même connexion, la couleur de chiffrement est égale au numéro du descripteur de la connexion plus un décalage. Cette solution garantit encore que deux opérations de chiffrement ne s'exécutent pas simultanément pour la même connexion.

Alors que la publication initiale sur Libasync-smp utilise la version 0.5 de SFS, nous avons modifié la dernière version disponible (0.7.2) pour la rendre compatible avec Libasync-smp et Mely. Nous avons utilisé cette version pour nos expérimentations.

Injection de charge. Nous avons effectué l'injection de charge via 16 nœuds clients connectés au serveur en utilisant un commutateur Ethernet non bloquant. Étant donné que la version de SFS que nous utilisons n'est pas capable de prendre en charge plus d'une interface réseau, nous avons utilisé une technique, appelée *bonding*, permettant de lier plusieurs cartes réseau réelles à une carte réseau virtuelle [92]. Cela nous permet de ne pas être limités par le débit d'une carte réseau.

Chaque machine cliente fait tourner un unique client. Il n'est pas possible d'avoir plusieurs clients par machine, car le pilote SFS est responsable de la communication avec le serveur et plusieurs points de montage utilisent le même pilote. Chaque client envoie des requêtes au serveur en utilisant le protocole SFS. Pour cela, nous utilisons l'injecteur de charge Multio [4] de la façon suivante : chaque client lit en boucle un fichier de 200 Mo. Ce test est identique à celui utilisé pour l'évaluation initiale de Libasync-smp. Dans cette expérience, nous supposons qu'aucun accès disque n'est effectué car le fichier tient dans la mémoire du serveur. Pour garantir que toutes les expériences sont identiques, le fichier demandé est chargé au lancement dans la mémoire cache du serveur. Enfin, pour être sûr que le client effectue bien les requêtes au serveur, le cache du système de fichiers du client est vidé entre deux requêtes. Le client est chargé de calculer le débit observé pour chaque requête de lecture. À la fin du test, un nœud maître est chargé de collecter et de synthétiser les résultats de chaque client.

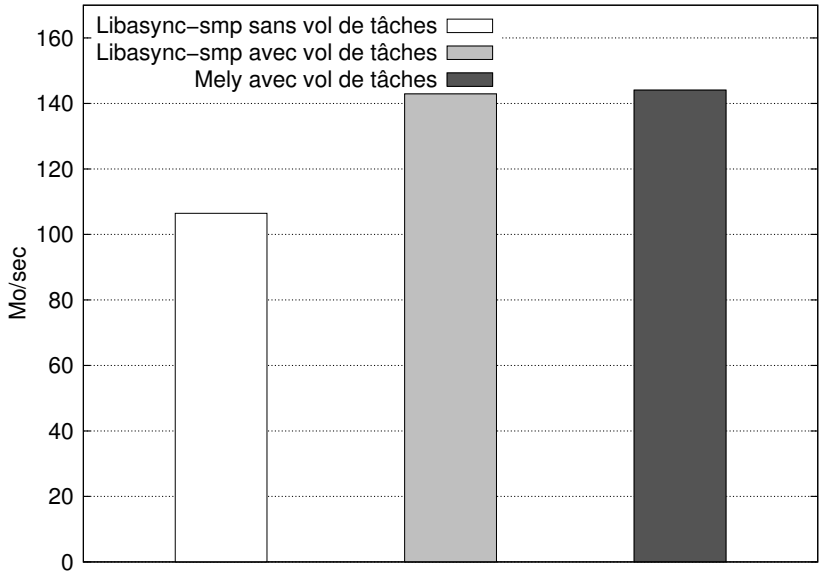


Figure 5.11 – Performance du serveur de fichiers sécurisé utilisant Mely par rapport à la version Libasync-smp avec et sans vol de tâches.

Résultats. Le débit total obtenu par les clients est illustré par la figure 5.11. Nous avons évalué Libasync-smp en activant puis en désactivant le mécanisme de vol de tâches ainsi que Mely avec le vol de tâches et ses heuristiques d'actives. Les écarts types observés sont imperceptibles.

Nous pouvons faire deux observations sur ces résultats. La première est que, comme nous l'avons déjà mentionné dans la section 5.1, le vol de tâches de Libasync-smp permet d'améliorer sensiblement les performances de près de 35 %. En effet, la charge sur les processeurs est déséquilibrée, car une majorité des événements ont la couleur par défaut 0 et sont ainsi initialement associés au premier cœur. Par conséquent, le vol de tâches permet de mieux équilibrer la charge. De plus, les événements volés ont un temps de traitement relativement long ce qui rend leur vol rentable. La seconde observation que nous pouvons faire est que les heuristiques que nous avons introduites ne dégradent pas les

performances. Dans ce cas, comme le vol est toujours très rentable, il n'est pas facile d'en améliorer les performances avec nos propositions, car elle n'ont d'impact sur les performances que si le vol doit être fait de façon à minimiser ses coûts et à maximiser la justesse de ses choix.

5.5 Conclusion

Le modèle de programmation événementiel est adapté pour la construction de serveurs de données qui réagissent à des requêtes clientes. Ce modèle d'exécution est réputé pour ses bonnes performances. La coloration d'événements est une extension au modèle de programmation événementiel classique qui permet d'utiliser simplement le parallélisme offert par les nouvelles architectures multi-cœurs.

Libasync-smp est un support d'exécution permettant la programmation événementielle colorée. Libasync-smp utilise un mécanisme de vol de tâches pour assurer l'équilibrage de charge entre les cœurs. Nous avons étudié, dans le contexte des serveurs de données, l'impact du mécanisme de vol de tâches sur la performance des applications. Nous avons notamment observé que ce mécanisme peut, dans certains cas comme celui du serveur Web, dégrader sensiblement les performances.

Pour améliorer les performances de la programmation événementielle colorée, nous avons introduit un nouvel environnement d'exécution, nommé Mely, qui est compatible avec les applications développées pour Libasync-smp. L'architecture de Mely a été pensée pour réduire les coûts liés au vol de tâches. Nous avons également proposé des heuristiques pour améliorer les performances de ce mécanisme en prenant de meilleures décisions. Ces optimisations peuvent être pour la plupart transparentes pour le programmeur et apportent des améliorations de performances sensibles. Nous avons montré des gains dans le cas du serveur Web jusqu'à 73 % par rapport à Libasync-smp avec vol de tâches et 25 % par rapport à Libasync-smp sans vol de tâches. Nous avons également montré que, dans le cas où le vol de tâches est efficace, Mely et ses heuristiques ne dégradent pas les performances.

Notre environnement d'exécution et nos heuristiques ont été évalués dans le contexte de la programmation événementielle colorée telle que définie par Libasync-smp. Nous pensons cependant que nos propositions sont plus générales et peuvent être aisément mises en œuvre dans d'autres environnements d'exécution multi-cœurs. Toutes les heuristiques proposées ont pour objectif d'améliorer la rentabilité du vol. Elles ont un impact sur les performances uniquement si le vol est coûteux et si certaines tâches ne sont pas rentables à voler. Par conséquent, ces heuristiques peuvent être mises en œuvre pour un autre support d'exécution dans lequel une partie des tâches est de courte durée. Ces tâches peuvent être peu rentables à voler à cause du temps passé dans le vol ou des fautes de cache induites. L'architecture de l'environnement d'exécution Mely peut également être mise en place pour tout environnement d'exécution dont le vol est soumis à des contraintes sur l'exécution des tâches telles que les couleurs.

Dans l'état actuel de nos travaux, le temps moyen d'exécution d'un traitant ainsi que les pénalités sur le vol des événements sont fixés par le programmeur en utilisant les possibilités de profilage offertes par Mely. Nous pensons, dans des travaux futurs, automatiser ces opérations en mesurant à la volée le temps d'exécution d'un traitant ainsi que sa consommation mémoire moyenne.

Chapitre 6

Étude et amélioration des performances du serveur Web Apache sur une architecture multi-cœurs NUMA

Sommaire

6.1 État de l'art de l'évaluation des serveurs Web 94

6.2 Mécanismes d'injection de charge 95

6.2.1 Caractéristiques des différents mécanismes d'injection de charge 96

6.2.2 Charge et environnement d'injection employés 98

6.3 Présentation du serveur Web Apache 100

6.3.1 Modèle d'exécution 100

6.3.2 Interaction/Intégration avec PHP 102

6.3.3 Détails de configuration 102

6.4 Configuration des expériences et métriques utilisées 104

6.4.1 Configuration matérielle 104

6.4.2 Configuration logicielle 105

6.4.3 Méthodologie et métriques utilisées 107

6.5 Étude de la performance du serveur Web Apache sur les architectures multi-cœurs 109

6.5.1 Mesure de performances initiale 109

6.5.2 Analyse de l'utilisation des cartes réseau 110

6.5.3 Analyse de l'utilisation des différents cœurs 111

6.5.4 Analyse du temps passé dans les fonctions 111

6.5.5 Analyse de l'efficacité des accès mémoire 112

6.5.6 Bilan 114

6.6 Première étape : Réduction des accès distants en colocalisant les processus Apache et PHP 115

6.6.1 Mesure de la performance de la solution 116

6.6.2 Analyse de l'utilisation des cartes réseau 117

6.6.3 Analyse de l'utilisation des différents cœurs 117

6.6.4 Analyse de l'efficacité des accès mémoire 119

6.6.5 Bilan 119

6.7 Deuxième étape : Équilibrage de la charge entre les nœuds en N-Copy 120
6.7.1 Mesure de la performance de la solution 121
6.7.2 Analyse de l'utilisation des cartes réseau 121
6.7.3 Analyse de l'utilisation des différents cœurs 122
6.7.4 Analyse du temps passé dans les fonctions 122
6.7.5 Analyse de l'efficacité des accès mémoire 124
6.7.6 Bilan 126
6.8 Discussion 126
6.9 Conclusion 127

Dans ce chapitre, nous avons choisi d'étudier le passage à l'échelle du serveur Web Apache. Ce serveur est le plus déployé à l'heure actuelle, c'est pourquoi il est intéressant d'étudier sa capacité à utiliser les architectures multi-cœurs NUMA dans des conditions de charges réalistes.

Dans la suite de ce chapitre, nous présentons tout d'abord l'état de l'art de l'évaluation des serveurs Web. Nous donnons ensuite dans la section 6.2 des détails sur les différents mécanismes d'injection de charge disponibles et tout particulièrement celui choisi. La section 6.3 fournit des informations sur l'architecture des serveurs Web en général et du serveur Apache en particulier. Nous donnons également des détails sur la configuration utilisée pour les expériences. La section 6.4 présente les conditions d'expérimentation à la fois logicielles et matérielles. Dans la section 6.5, le passage à l'échelle du serveur Apache est étudié et nous déterminons, à l'aide d'un ensemble de métriques, les raisons de son passage à l'échelle limité. Enfin, les sections 6.6 et 6.7 présentent deux améliorations introduites pour ce serveur, permettant d'améliorer de 20 % ses performances à 16 cœurs. Nous terminons ce chapitre par une discussion sur les problèmes restant et sur les solutions qui peuvent être envisagées dans des travaux futurs.

6.1 État de l'art de l'évaluation des serveurs Web

Une évaluation des performances des différentes architectures de serveurs Web a été menée par Pariag *et al.* [76]. Ils comparent trois différentes architectures : une architecture événementielle (μserver), une architecture à base d'étages (watpipe) et une architecture utilisant des threads (knot). L'objectif de ces travaux est de savoir quel modèle d'exécution est le plus adapté pour construire des serveurs Web efficaces. Les auteurs sont arrivés à deux conclusions : (i) le modèle événementiel ainsi que le modèle à base d'étages sont généralement plus performants que le modèle utilisant un thread par connexion, et (ii) une mauvaise configuration des serveurs Web peut grandement impacter les performances de celui-ci.

Cette évaluation est toutefois incomplète pour plusieurs raisons. Premièrement, les serveurs Web n'ont été évalués que sur des architectures mono-processeur. La validité de leurs conclusions sur les architectures multi-cœurs reste donc à démontrer. Deuxièmement, les auteurs ont évalué une charge en boucle semi-ouverte. La distribution de fichiers utilisée ne concerne que la partie statique du test SPECWeb99 [87]. L'impact de la partie dynamique de SPECWeb, et notamment l'interaction entre les serveurs Web et PHP, n'a pas été prise en compte. Enfin, pour le modèle de programmation à base de threads, les auteurs ont choisi Knot, un serveur Web expérimental basé sur Capriccio. Toutefois, Capriccio est une bibliothèque de threads de niveau utilisateur qui ne s'appuie que sur un seul thread noyau, ce qui ne permet pas d'exploiter de multiples cœurs au niveau applicatif. De plus, Knot est un serveur Web qui n'a pas été largement déployé, contrairement à d'autres tels qu'Apache.

Dans la suite de ces travaux [51], Harji a évalué la performance des serveurs précédemment cités sur les architectures multi-cœurs (excepté Knot). Ces travaux ont les mêmes lacunes que les

précédents : la distribution évaluée est uniquement composée de fichiers statiques et les serveurs choisis ne sont pas des serveurs utilisés en production. De plus, comme Knot ne sait pas tirer parti des multi-cœurs, il n'a pas été évalué. Par conséquent, aucun serveur Web utilisant le modèle de threads n'a été évalué dans ces travaux.

Veal et Foong ont mené en 2007 une campagne de tests visant à évaluer le passage à l'échelle du serveur Web Apache sur une architecture Intel à 8 cœurs [95]. La charge mise en œuvre pour cette évaluation est la charge SPECWeb2005 Support que nous décrivons dans la partie 6.2. La distribution de fichiers utilisée a été réduite pour pouvoir tenir intégralement en mémoire. Ces évaluations ont montré que le serveur Web Apache n'a pas un passage à l'échelle idéal sur les architectures multi-cœurs. Une analyse détaillée du comportement d'Apache a permis de déterminer que le problème ne vient pas d'un mauvais passage à l'échelle du système d'exploitation ou de l'application. Dans leur cas, le problème est matériel avec la saturation du bus d'adresses.

Tout d'abord, notons que cette évaluation a été effectuée sur une machine avec des temps d'accès à la mémoire uniformes. Comme nous l'avons décrit dans le chapitre 1, dans une telle architecture toutes les communications avec la mémoire passent par le même bus qui devient par conséquent le goulot d'étranglement. Les nouvelles architectures NUMA ont justement pour objectif de lever cette restriction en distribuant la mémoire sur les processeurs. Par conséquent, les conclusions formulées par Veal et Foong ne sont plus valides dans notre contexte. Par ailleurs, leur évaluation n'utilise que 4 cartes réseau. D'après leurs résultats, l'utilisation de seulement 4 cartes présente des risques de limitation du passage à l'échelle dus à une saturation de ces cartes. Il est étonnant qu'ils n'aient pas employé, avec Apache, l'intégralité des 8 cartes réseau présentes sur leur machine de test (et utilisées par ailleurs dans leurs micro-tests).

Les systèmes d'exploitation Corey [100] et Barrelfish [18] ont été en partie évalués sur des architectures multi-cœurs NUMA en utilisant des serveurs Web. Cependant, il existe plusieurs problèmes à ces évaluations. Premièrement, dans les deux cas, les serveurs Web utilisés sont des serveurs écrits spécialement pour l'expérience et n'ont jamais été testés dans des conditions réelles de production. Deuxièmement, les charges utilisées dans ces tests sont relativement éloignées des conditions de charge réelles [1]. Enfin, comme ces évaluations font parties d'un ensemble plus global de tests, peu de détails sont donnés sur les indicateurs de performances.

Toyotaro *et al.* [90] ont choisi d'améliorer les interactions entre le serveur Web et l'interpréteur de scripts PHP. Ils ont proposé une solution pour limiter les copies entre ces deux entités lors du traitement d'une requête. Leur solution traite le cas dans lequel le script PHP inclut un fichier dans la réponse via la fonction `file_get_content`. Dans le cas standard, le fichier est lu par l'interpréteur PHP et est renvoyé au serveur Apache avec le reste des données générées par le script. Ce fonctionnement empêche l'utilisation du mécanisme d'envoi sans copie `sendfile`. Leur solution est donc de modifier le protocole FastCGI pour qu'il n'inclue plus directement le fichier dans la réponse, mais qu'il insère à la place une directive à l'attention du serveur Apache pour que ce dernier puisse utiliser `sendfile` pour l'envoi du fichier. Notons que cette solution a été évaluée uniquement dans le cadre d'une machine uniprocesseur.

6.2 Mécanismes d'injection de charge

Dans cette section, nous décrivons tout d'abord le fonctionnement des injecteurs de charge pour les serveurs Web. Nous nous intéressons ensuite aux trois modèles d'injection de charge disponibles ainsi qu'aux ensembles de fichiers pouvant être utilisés pour simuler un véritable site Internet en

1. Barrelfish utilise toutefois une charge moins fictive que celle de Corey.

fonction de ses caractéristiques. Enfin, nous présentons l'injecteur de charge que nous avons retenu et fournissons des informations sur le protocole expérimental utilisé dans la suite de ce chapitre.

6.2.1 Caractéristiques des différents mécanismes d'injection de charge

Un système d'injection de charge est caractérisé par deux aspects principaux :

- *Le modèle d'injection de charge.* Ce modèle spécifie le comportement des clients. Ce comportement indique notamment la loi d'arrivée des nouvelles connexions, le nombre de requêtes par connexion ou encore le nombre de requêtes concurrentes sur le serveur.
- *La distribution de fichiers utilisée.* Cette distribution, ainsi que les fréquences d'accès à chacun des fichiers, permet de simuler un type de site bien défini. Par exemple, le nombre et la fréquence d'accès à des données générées dynamiquement sont différents si le site offre un service de vente en ligne ou de consultation de comptes bancaires.

Dans la suite de cette section, nous allons détailler, pour chacun de ces aspects, les différents choix possibles avec leurs avantages et leurs inconvénients respectifs.

6.2.1.1 Modèles d'injection de charge

Les deux principaux modèles d'injection de charge sont la boucle fermée et la boucle ouverte. Ces modèles diffèrent essentiellement par leur gestion du nombre de requêtes effectuées simultanément sur le serveur. Il existe également un troisième modèle, appelé boucle semi-ouverte, qui a pour objectif de combiner les points forts des deux modèles précédents. L'analyse des résultats se concentre généralement à la fois sur la latence de chargement des pages perçue par le client ainsi que sur le débit en nombre de requêtes traitées par seconde par le serveur.

Boucle fermée. Un modèle d'injection en boucle fermée fonctionne de la manière suivante : au début de l'expérience, un nombre défini de clients se connectent au serveur. Par la suite, plus aucun nouveau client ne demandera de connexion. Un client émet des requêtes et attend la réception de la réponse avant de réémettre une nouvelle requête. Pour modéliser plus finement le comportement d'un client, un délai aléatoire peut être introduit entre la réception de la réponse et l'émission d'une nouvelle requête. Ce délai permet de simuler un temps de réflexion de l'utilisateur qui reçoit une page Web et l'analyse avant de sélectionner un nouveau lien. Dans le cas de l'injection en boucle fermée, le débit de requêtes en entrée du serveur est par conséquent fortement lié à son débit en sortie. Un autre paramètre de ce modèle d'injection est le nombre de requêtes qu'un client va effectuer au sein d'une connexion avant de la fermer puis d'en ouvrir une nouvelle. Avec ce modèle d'injection, la charge sur le serveur est exprimée en nombre de clients.

Ce modèle de charge est utilisé par deux des injecteurs de charge pour serveur Web les plus connus : TPC-W [45] et les différents profils de SPECWeb 2005 [86].

Boucle ouverte. Le second modèle d'injection pouvant être utilisé est celui en boucle ouverte. Dans ce modèle, le nombre de clients n'est pas fixé à l'avance, mais de nouveaux clients se connectent régulièrement au serveur pour effectuer une ou plusieurs requêtes sur le serveur en parallèle. Le nombre de clients effectuant des requêtes peut varier grandement. Il est possible avec ce modèle de mettre plus facilement le serveur dans une position de surcharge en lui demandant d'accepter de nouvelles connexions à un rythme plus important que son débit maximum. Avec ce modèle d'injection, la charge sur le serveur est exprimée en nombre de requêtes par seconde.

Le modèle de charge en boucle ouverte est utilisé par plusieurs injecteurs de charge pour serveurs Web tels que Httperf [72].

Boucle semi-ouverte. Schroeder *et al.* ont démontré que selon le modèle d'injection de charge, les conclusions peuvent être différentes [82]. Par exemple, un modèle d'injection de charge en boucle ouverte permet de mettre en évidence plus aisément les effets de différentes politiques d'ordonnancement. Dans d'autres cas, notamment lorsque le nombre de clients simulés est important, le modèle fermé est identique en terme de comportement au modèle en boucle ouverte. Aucun des deux modèles n'est représentatif d'une véritable charge. Schroeder *et al.* proposent donc de mélanger les concepts des deux modèles précédents en un nouveau basé sur une boucle semi-ouverte. Dans cette solution, après avoir effectué une requête, les clients peuvent soit en faire une nouvelle avec une probabilité λ, soit arrêter les demandes au serveur avec une probabilité $1 - \lambda$. Comme avec les deux autres modèles d'injection de charge présentés précédemment, un temps d'attente peut être inséré entre deux requêtes d'un même client. De nouveaux clients peuvent également se connecter au serveur durant le test pour remplacer les clients partis. Dans ce type d'injection, le nombre de clients n'est donc pas constant à tout instant, mais une partie des entrées est quand même asservie à la sortie du serveur.

6.2.1.2 Distribution de fichiers utilisée

En plus de la modélisation de l'arrivée des clients et de leur fréquence de requêtes, un injecteur de charge doit définir un ensemble de ressources pouvant être manipulées ainsi que le modèle d'accès à ces ressources. Ces ressources peuvent être soit des pages Web statiques, soit des pages Web dynamiques, soit d'autres fichiers tels que des images ou des fichiers à télécharger (document au format *PDF*, fichiers à exécuter sur la machine cliente, etc.). Les pages Web dynamiques nécessitent un traitement du serveur pour construire la réponse en fonction des paramètres de la requête, ainsi que de l'état de ressources tierces comme une base de données.

Le premier modèle d'accès est très simple et est utilisé fréquemment. L'ensemble de fichiers demandés est composé de fichiers statiques dont les probabilités d'accès sont égales. C'est le modèle que nous avons utilisé dans le chapitre 5 pour évaluer nos propositions.

Le second modèle utilisable est basé sur la reproduction d'une exécution à partir d'observations préliminaires. Lors de cette observation, la trace correspondant aux fichiers requis et aux ouvertures et fermetures de connexions est récupérée. De telles traces sont généralement collectées lors d'événements particuliers, comme la coupe du monde de football 98 [14]. La charge sur le serveur est donc plus réaliste, car elle est basée sur une observation de faits réels.

Enfin, la dernière possibilité est celle utilisée dans les injecteurs TPC-W [45] et SPECWeb [86]. Le principe est d'avoir un modèle probabiliste qui donne la fréquence d'accès à chacun des fichiers (ou à chacune des classes de fichiers) ainsi que le temps d'attente entre deux requêtes. Ce modèle prend également en compte le comportement des clients. En effet, après avoir consulté une page, un client peut préférentiellement accéder à une ressource donnée. Le comportement des clients est ainsi différent en fonction du type de service fourni par le serveur. Par exemple, SPECWeb05 permet de simuler un site marchand, un site de banque en ligne ou encore un site de support avec des documentations en ligne (plus de détails sont donnés sur cette charge dans la section 6.2.2).

6.2.1.3 Architecture interne des injecteurs de charge

Lors de la conception d'un injecteur de charge, les points fondamentaux à considérer sont le nombre de clients ainsi que le nombre de requêtes par seconde qui vont être simulés. Pour gérer efficacement un grand nombre de clients, il est nécessaire pour un injecteur de charge d'utiliser une

architecture *maître-esclave* dans laquelle le maître est utilisé pour synchroniser les injections des esclaves ainsi que pour collecter et synthétiser les résultats. Il est fondamental d'utiliser une telle architecture, car une machine n'est pas capable de simuler seule un nombre de clients suffisant pour s'approcher d'une situation réaliste. En effet, cette machine est rapidement limitée soit par le débit maximum de ses interfaces réseau, soit par sa capacité de traitement des réponses et des requêtes.

Par ailleurs, un injecteur de charge doit être capable de simuler le plus grand nombre possible de clients par machine. En effet, cela permet de restreindre grandement le nombre de machines nécessaires pour réaliser l'injection d'une charge donnée. Cette caractéristique est importante puisque l'ensemble de machines clientes à disposition est généralement contraint. Une solution simple pour la mise en œuvre est d'affecter un thread par client. Cependant, la coûteuse gestion des threads limite le nombre de clients pouvant être simulés. En réponse à ce problème, Banga et Druschel [17] ont proposé et démontré l'efficacité d'une architecture événementielle basée sur deux threads : un pour l'envoi de requêtes et un pour la réception des réponses.

6.2.2 Charge et environnement d'injection employés

À partir des considérations effectuées dans la section précédente, nous avons choisi d'utiliser la charge `Support` de SPECWeb 2005 pour nos évaluations. En effet, l'avantage de cette charge est d'être représentative de charges réellement obtenues sur des serveurs en production. L'inconvénient du modèle de charge en boucle fermée est compensé par le nombre important de clients simulés qui permettent de saturer les ressources du serveur.

SPECWeb est basé sur une injection de charge en boucle fermée. Pour qu'une exécution soit considérée comme valide, il est nécessaire que 95 % des requêtes soient servies dans un temps jugé « bon » et 99 % des requêtes soient servies dans un temps jugé « raisonnable ». L'injecteur de charge de SPECWeb est un programme Java utilisant un thread par client simulé. Par conséquent, il est nécessaire d'utiliser une grande quantité de machines pour simuler un nombre important de clients.

Une injection SPECWeb est composée de cinq phases distinctes :

- *La phase de* **ramp-up**. Cette période sert à créer tous les threads sur toutes les machines employées pour l'injection de charge. Cette phase a une durée de 180 secondes.

- *La phase de* **warm-up**. Cette période est utile pour mettre le serveur dans les conditions expérimentales. Cette période est notamment nécessaire pour assurer que, sur un serveur Web utilisant le modèle à base de threads, tous les threads aient été créés avant l'expérience. De même, cette phase est importante si le serveur utilise un cache de fichiers applicatif, car les fichiers les plus réclamés seront préchargés dans ce cache. Cette phase à une durée de 1200 secondes.

- *La phase de test.* C'est durant cette période que sont récupérées les statistiques sur le débit et la latence perçus par les clients. Cette phase a une durée de 1800 secondes.

- *La phase de* **ramp-down**. Cette période correspond à une décharge progressive du serveur. Cette phase à une durée de 180 secondes.

- *La phase de collecte des résultats.* Durant cette période, les résultats sont envoyés depuis les machines clientes vers la machine maître qui synchronise les injections. Cette phase n'a pas de durée prédéterminée.

La charge `Support` de SPECWeb 2005 simule un serveur de support en ligne. Ce site permet aux utilisateurs de lister les produits en utilisant éventuellement des filtres, ainsi que de télécharger des

documents techniques ou des applications. Nous avons choisi d'utiliser SPECWeb, car cet injecteur permet d'obtenir des charges réalistes basées sur une modélisation du comportement des clients. Parmi les 3 charges SPECWeb disponibles, deux sont extrêmement intensives dans l'utilisation des scripts dynamiques. Ce sont les charges `E-Commerce` et `Banking`. Nous avons choisi d'utiliser la charge `Support` car elle est moins intensive en terme de génération de pages dynamiques que les deux précédentes, ce qui concentre les problèmes sur le serveur Web plutôt que sur le moteur de script.

Dans la charge SPECWeb Support, différents types de fichiers peuvent être sollicités. Le premier type concerne les fichiers générés dynamiquement. Le second concerne les images. Le troisième représente les fichiers disponibles au téléchargement. Ces derniers sont découpés en classes en fonction de leur taille et une pondération est affectée à chaque classe. Le fichier le plus petit à une taille de 105 Ko alors que le fichier le plus gros à une taille de 37.8 Mo. Chaque dossier contient un nombre fixe de fichiers de chaque classe. Le nombre de dossiers générés est égal au nombre requêtes simultanées sur le serveur (`SIMULTANEOUS_SESSIONS`) multiplié par un coefficient (`DIRSCALING`). Le tableau 6.1 illustre l'accès aux différents types de fichiers. Nous pouvons observer que les fichiers les plus populaires sont les images. Cette répartition a été observée en utilisant les traces d'une injection sur le serveur Web Apache.

Type de fichier	**Ratio d'accès observé**
Fichiers disponibles en téléchargement	0.3 %
Images	95 %
Scripts PHP	4.7 %

TABLE 6.1 – Répartition observée de l'accès aux ressources avec l'injection de charge SPECWeb 2005 Support.

Dans un système réel, les scripts de génération dynamique de pages peuvent faire des accès à une base de données pour construire leur réponse. SPECWeb prend en compte cette spécificité en introduisant un simulateur de base de données, appelé BeSim *(Back-end Simulator)*.

Pour adapter le test SPECWeb 2005 Support à notre architecture, nous avons effectué quelques modifications sur les paramètres de l'injection SPECWeb 2005 Support :

- Nous avons supprimé la simulation de la base de données (*BeSim*). En effet, un tel simulateur nécessite d'avoir une autre machine capable d'absorber la charge pour qu'il ne devienne pas le goulot d'étranglement.

- Nous avons réduit la taille de la partie téléchargement en réduisant l'option `DIRSCALING` de 0.25 à 0.00625. Avec le coefficient initial, la taille totale de la distribution de fichiers aurait été de plus de 770 Go. Avec nos modifications, cette taille est ramenée à un peu plus de 19 Go. L'objectif de ce changement est d'avoir une charge capable de tenir en mémoire afin d'éviter les accès disque. En effet, dans ce cas, les accès disques sont le principal goulot d'étranglement. Des architectures récentes [85] utilisent jusqu'à 56 disques durs en parallèle pour supprimer ce goulot d'étranglement. Comme nous n'avons pas accès à ce type de système de stockage, nous supprimons le goulot d'étranglement en faisant tenir les données en mémoire. Il est important de noter que nous avons utilisé la même valeur que d'autres études antérieures (notamment celle de Veal et Foong décrite dans la section 6.1) pour le paramètre `DIRSCALING`.

- Pour raccourcir la durée des tests, nous avons modifié la durée de chacune des phases de warm-up et de test. Nous avons réduit la phase de warm-up de 1200 à 300 secondes et la phase de test de 1800 à 180 secondes. Nous avons dimensionné la phase de warm-up de manière à ce

que tous les threads d'Apache aient été créés au moment de la phase de test. Ces modifications ont permis de réduire la durée totale d'une injection de charge de 56 minutes à 14 minutes sans compter le temps nécessaire à la collecte des résultats. Nous avons vérifié sur plusieurs expériences que ces paramètres ne changent pas les comportements constatés ni la reproductibilité des résultats malgré des temps plus courts.

- Tout comme Veal et Foong avant nous, nous avons rencontré un problème avec le paramètre `MAX_OVERTHINK_TIME`. Ce paramètre fixe la déviation maximum entre le temps de réflexion voulu par un client et celui mesuré. Avec la valeur par défaut, il est arrivé que certains clients soient bloqués. En augmentant ce paramètre à la valeur préconisé par Veal et Foong, ce problème a disparu.

Pour effectuer notre injection, nous utilisons un groupe de 22 machines connectées au serveur par des liens gigabit. Le commutateur reliant les machines dispose de 48 ports non bloquants.

6.3 Présentation du serveur Web Apache

Cette section présente le serveur Web Apache dont l'évaluation des performances sur les architectures multi-cœurs est présentée dans ce chapitre. Nous avons choisi d'évaluer Apache, car il s'agit du serveur le plus déployé à l'heure actuelle. Le tableau 6.2 présente la répartition des parts de marché à l'échelle mondiale entre les 5 principaux serveurs Web utilisés en mai 2010 d'après la société Netcraft [7]. Nous pouvons voir dans ce tableau que cinq serveurs se partagent 94 % des parts de marché et qu'Apache fournit l'hébergement de plus de la moitié des sites Web.

Serveur Web	Nombre de sites hébergés	Pourcentage de part de marché
Apache	113 M	55 %
Microsoft IIS	51 M	25 %
Nginx	13 M	7 %
Google	12 M	6 %
lighttpd	2 M	1 %

TABLE 6.2 – Répartition des parts de marché entre les 5 principaux serveurs Web.

Nous allons tout d'abord nous intéresser au modèle d'exécution du serveur Web Apache. Ce choix de conception conditionne les performances ainsi que le nombre de clients pouvant être traités en parallèle par le serveur. Nous allons ensuite décrire ses interactions avec l'interpréteur de scripts PHP utilisé pour générer des pages Web dynamiques. Nous donnons également des détails sur la configuration d'Apache utilisée pour son évaluation. Enfin, nous terminons par l'état de l'art sur l'évaluation de la performance des serveurs Web.

Il est important de remarquer que le terme *passage à l'échelle* peut designer soit l'augmentation des performances lorsque le nombre de clients augmente, soit l'augmentation des performances lorsque le nombre de cœurs augmente. Dans la suite de ce document, ce terme est utilisé dans son deuxième sens.

6.3.1 Modèle d'exécution

Le modèle d'exécution est un des choix qui a un impact important sur la performance du serveur. Nous avons vu dans le chapitre 2 que deux modèles principaux existent pour développer des applications : le modèle à base de processus ou de threads et le modèle événementiel. D'autres modèles, comme celui basé sur les étages, permettent de développer des serveurs Web efficaces.

Apache se situe dans la première catégorie. Il utilise des processus ou des threads pour gérer les requêtes entrantes. Plus précisément, il est possible d'utiliser trois modèles d'exécution différents (appelés *MPM, Multi-Processing Modules*).

- *Le modèle* `Prefork` *correspond à un modèle d'exécution utilisant des processus.* Dans ce modèle, chaque processus est chargé de traiter une connexion. Les processus sont créés à la demande, c'est-à-dire que lorsqu'une nouvelle connexion est effectuée, le processus ayant accepté la connexion va tout d'abord regarder si un processus libre existe. Si un tel processus n'existe pas, un nouveau processus est démarré. Lorsqu'un client ferme sa connexion vers le serveur, le processus correspondant est inséré dans la liste des processus libres. Si cette liste est pleine, le processus est arrêté.

- *Le modèle* `Worker` *est un modèle d'exécution hybride utilisant à la fois des processus et des threads pour gérer les connexions des clients.* Dans ce modèle, chaque connexion est gérée par un thread. Un processus père est responsable du démarrage des processus fils. Chaque processus fils est composé d'un nombre de threads fixé. Tout comme le modèle `prefork`, un ensemble de threads inactifs est gardé pour réduire les coûts de création et de suppression de threads. L'objectif de ce modèle d'exécution par rapport au précédent est de permettre la gestion d'un grand nombre de clients en parallèle. En effet, les threads ont à la fois une empreinte mémoire et des coûts de gestion moins importants que les processus. Le modèle est hybride pour conserver la robustesse apportée par l'isolation inter-processus. En effet, si une requête engendre une faute du serveur, seuls les autres threads du processus peuvent être impactés et non pas toutes les connexions du système. Le nombre de threads par processus est également limité, sur les machines 32 bits, par la consommation de mémoire virtuelle liée à la pile des threads.

- *Le modèle* `Event` *est un modèle très proche de la version* `worker`. Contrairement à ce dernier, les threads associés aux connexions inactives sont utilisés pour traiter d'autres demandes. En effet, dans le protocole HTTP/1.1 il est possible de garder une connexion ouverte pour effectuer plusieurs requêtes. Le temps de réflexion du client entre deux requêtes peut être long, utilisant ainsi inutilement une partie des ressources de la machine pour stocker notamment les piles de threads. Dans le modèle d'exécution `event`, les connexions inactives sont surveillées par le processus maître en utilisant un mécanisme de type `epoll` précédemment décrit. Contrairement à ce que son nom peut suggérer, le modèle `event` se rapproche du modèle événementiel uniquement par sa gestion des connexions inactives. `Event` est un modèle d'exécution plus proche d'un modèle utilisant un pool de thread. Pour le moment, ce modèle d'exécution est considéré comme expérimental.

Dans toutes les configurations, pour éviter de créer trop de processus et donc d'avoir des temps de réponse (latences) inacceptables, il est possible de spécifier le nombre maximum de processus utilisables et ainsi de borner le nombre de clients à un instant donné sur le serveur. Cette limite est appelée *MPL (Multi-Programming Level)*. La vitesse de création des threads est bornée pour éviter des créations et des suppressions de threads trop fréquentes. Par conséquent, il est important, lors de l'injection de charge, de considérer cette caractéristique en ayant une phase de montée en charge suffisamment longue pour que le serveur arrive à un régime stationnaire.

Dans les expériences menées par la suite, le modèle d'exécution `worker` a été choisi à cause de sa meilleure résistance à la charge par rapport au modèle `prefork`. C'est également le modèle généralement choisi pour les serveurs en production soumis à un grand nombre de clients. Nous n'avons pas choisi la version `event` car elle est encore considérée comme expérimentale et, par conséquent, peu déployée.

6.3.2 Interaction/Intégration avec PHP

Le serveur Web Apache n'est pas le seul acteur impliqué dans les réponses aux clients. Les pages Web dynamiques font généralement appel à un autre programme qui, à partir des paramètres de la requête, construit une réponse personnalisée. Ce second acteur peut être par exemple une machine virtuelle PHP, une machine virtuelle Java ou encore d'autres programmes CGI *(Common Gateway Interface)*. L'interaction avec ce processus tierce est donc une composante importante de la performance du serveur. Dans notre évaluation, nous utilisons le langage PHP pour la génération des pages dynamiques, car ce langage de script est très populaire et très fréquemment utilisé pour le développement de pages Web dynamiques. Plusieurs choix sont possibles pour l'exécution de la machine virtuelle :

- *Interpréter directement les scripts PHP dans le contexte de l'appelant.* Cette solution peut être mise en place lorsque le modèle d'exécution `prefork` est utilisé. Il n'est pas possible d'utiliser ce type d'exécution avec les autres modèles, car ils utilisent des threads et la machine virtuelle PHP n'est pas réentrante.

- *Utiliser un ou plusieurs serveurs PHP.* La communication avec ces serveurs est effectuée en utilisant le protocole FastCGI. Dans ce cas, Apache envoie la demande de page à construire en passant également les paramètres de la requête puis attend la réponse qui lui sera retournée par le serveur PHP. La communication entre les deux est effectuée au moyen d'un tube (*pipe*) ou d'une socket Unix. Le passage de paramètres est effectué par copie. Ce modèle est utilisé lorsque la version `worker` d'Apache est choisie, car l'isolation entre les requêtes PHP est assurée par les multiples instances de la machine virtuelle puisqu'un processus PHP ne dispose que d'un unique thread. Pour cette raison, nous utilisons cette solution dans nos expériences.

Lorsque la solution choisie consiste à utiliser un ou plusieurs serveurs PHP, il est nécessaire de calibrer finement le nombre de processus PHP utilisés, ainsi que leur durée de vie. En particulier, pour éviter les fuites mémoire, les processus PHP sont redémarrés périodiquement. Le fait de fixer cette période a un impact sur les performances : un redémarrage trop fréquent implique des surcoûts alors qu'une période trop importante entre deux redémarrages risque d'entraîner des fuites de mémoire importantes. De même, le nombre de processus alloués à PHP a un impact important sur les performances. Le tableau 6.3 montre le nombre de requêtes traitées dans un temps considéré comme bon à 16 cœurs avec 30000 clients [2]. Ce tableau montre bien que si le nombre de serveurs PHP est trop faible, peu de requêtes peuvent être satisfaites dans un temps considéré comme bon, car PHP devient le point d'étranglement. Inversement, si un nombre trop important de processus PHP sont utilisés, le même phénomène apparaît et est dû à l'empiètement trop important de PHP sur le temps imparti aux processus Apache. Il y a donc un intervalle dans lequel les performances sont bonnes. Dans nos expériences, le nombre de processus PHP est fixé manuellement à sa meilleure valeur après observation (soit 240 processus dans l'expérience présentée précédemment).

6.3.3 Détails de configuration

Dans cette section, nous donnons des détails additionnels sur la configuration de notre architecture serveur. Tout d'abord, nous avons activé uniquement les traces concernant les erreurs du serveur et avons désactivé toutes les autres. L'objectif de cette manipulation est de limiter le plus possible les accès au disque dur. Toujours dans l'optique d'éviter que le disque dur soit le goulot d'étranglement

2. Cette valeur correspond au débit maximal du serveur Web Apache à 16 cœurs, comme nous le montrons dans la section 6.5

Nombre de processus PHP	Ratio de requêtes « bonnes »	Ratio de requêtes « tolérables »
30	8 %	8 %
60	7 %	7 %
120	11 %	11 %
180	83 %	96 %
240	98 %	100 %
300	74 %	87 %
360	28 %	40 %

TABLE 6.3 – Impact du nombre de processus PHP sur le débit du serveur Apache à 16 cœurs et 30000 clients.

de l'application, avant chaque expérience, tous les fichiers pouvant être demandés sont préchargés dans la mémoire centrale. Comme nous disposons d'une architecture NUMA composée de 4 nœuds mémoire de 8 Go chacun, nous préchargeons donc les données équitablement sur chacun des nœuds mémoire.

Le tableau 6.4 indique les valeurs que nous avons affectées aux principaux paramètres d'Apache. Nous avons principalement modifié les limites concernant le nombre de clients pouvant être acceptés simultanément de manière à ce qu'aucune connexion ne soit refusée. Nous avons également augmenté le délai de garde avant la fermeture de la connexion, de manière à ce que tous les clients reçoivent une réponse. Nous avons également augmenté le nombre de threads par processus au seuil maximal pour réduire le nombre de processus dans le système.

Comme la charge que nous avons choisie comporte une importante partie de fichiers statiques, nous avons choisi d'activer l'option `sendfile`. `Sendfile` est un appel système qui permet d'effectuer un envoi de fichier en évitant les copies entre le noyau et l'espace utilisateur. Dans un système n'utilisant pas `sendfile`, l'application va faire une première copie du fichier depuis le noyau vers un tampon en espace utilisateur lors de la lecture, puis une seconde copie depuis l'espace utilisateur vers le noyau lors de l'envoi des données dans la socket. `Sendfile` permet de s'affranchir de ces copies, et ainsi de diminuer la charge processeur et mémoire, en déléguant au noyau la tâche d'envoi du fichier dans la socket.

Paramètre	Valeur
ServerLimit	2500
StartServers	10
MaxClients	160000
MinSpareThreads	250
MaxSpareThreads	250
ThreadsPerChild	64
MaxRequestsPerChild	0
MaxKeepAliveRequests	6000
KeepAliveTimeout	3600
EnableSendfile	on

TABLE 6.4 – Principaux paramètres de configuration du serveur Web Apache.

Comme nous l'avons vu dans la partie précédente la configuration de PHP à un impact important sur les performances du système. En plus de chercher manuellement les bonnes valeurs en ce qui concerne le nombre de processus PHP, nous avons utilisé eAccelerator [1], un accélérateur pour la

compilation des scripts PHP. Sans ce cache, à chaque accès à la page, le code de PHP est lu et compilé dans un langage intermédiaire (*bytecode*). La machine virtuelle exécute ensuite le script à partir de ce *bytecode*. Le principe d'eAccelerator est de conserver dans un cache le *bytecode* des pages les plus populaires dans le but d'accélérer les phases de lecture du fichier et de compilation. Le nombre de pages pouvant être gardées en cache est dépendant de la taille du cache (cette taille est fixée à 16 Mo dans nos expériences, ce qui correspond à la valeur par défaut). L'utilisation d'un cache pour le *bytecode* PHP est une optimisation fréquemment utilisée en production. Nous avons mesuré un gain de performance de 34 % en activant eAccelerator sur la configuration d'Apache à 16 cœurs.

6.4 Configuration des expériences et métriques utilisées

Dans cette section, nous donnons des détails sur l'architecture matérielle de la machine de test, ainsi que sur la configuration logicielle utilisée.

6.4.1 Configuration matérielle

Pour les expériences présentées dans ce chapitre, nous utilisons une machine NUMA équipée de 16 cœurs, de 20 cartes réseau 1Gb/s, ainsi que de 32 Go de mémoire centrale. Les caractéristiques (temps d'accès mémoire, débit des liens HyperTransport, etc.) des processeurs ont été présentées dans le chapitre 1 (section 1.4). La figure 6.1 illustre les interconnexions entre les processeurs, ainsi que les liens avec les périphériques d'E/S. Cette topologie a été vérifiée empiriquement en utilisant les micro-tests d'accès à la mémoire décrits dans le chapitre 1 et en combinant le résultat de chacun de ces tests avec l'observation des compteurs de performances fournis par le processeur. Il est important de considérer plusieurs caractéristiques :

- *Le débit maximum des liens HyperTransport est de 24 Gb/s.* Par conséquent, ces liens sont théoriquement capables d'absorber la charge des 20 cartes réseau qui peuvent gérer jusqu'à 20 Gb/s en émission comme en réception.

- *Toutes les cartes réseau sont connectées sur le lien d'E/S n°2.* Le lien d'E/S n°1 est réservé aux autres périphériques tels que les disques durs ou encore les périphériques USB. Le débit maximum théorique du lien d'E/S n°2 est de 24 Gb/s, ce qui permet d'absorber la charge des 20 cartes réseau.

- *Les 20 cartes réseau utilisées sont des Intel Pro/1000 à 1 Gb/s.* Lorsqu'un paquet doit être envoyé à destination d'un client, la carte va directement chercher le contenu en mémoire via le mécanisme de DMA (*Direct Memory Access*).

- *Il n'y a pas de connexion HyperTransport entre les nœuds 0 et 3.* Par conséquent, lorsque des données transitent du nœud 0 vers le nœud 3 ces données sont routées par le nœud 1. Inversement, pour les données transitant du nœud 3 vers le nœud 0, ces données sont routées par le nœud 2. Notons que la politique de routage est statique et ne dépend pas de la charge sur les liens HyperTransport. Ce routage est valable également pour les accès à la mémoire effectués par les cartes réseau grâce au mécanisme de DMA. Les paquets émis par la carte réseau se situant sur le nœud 0 vont ainsi transiter par le nœud 1 puis par le nœud 3 avant d'être émis.

Comme nous l'avons décrit dans la section 6.2, l'injection est effectuée par 24 machines connectées au serveur par un commutateur Ethernet 1 Gb/s non bloquant.

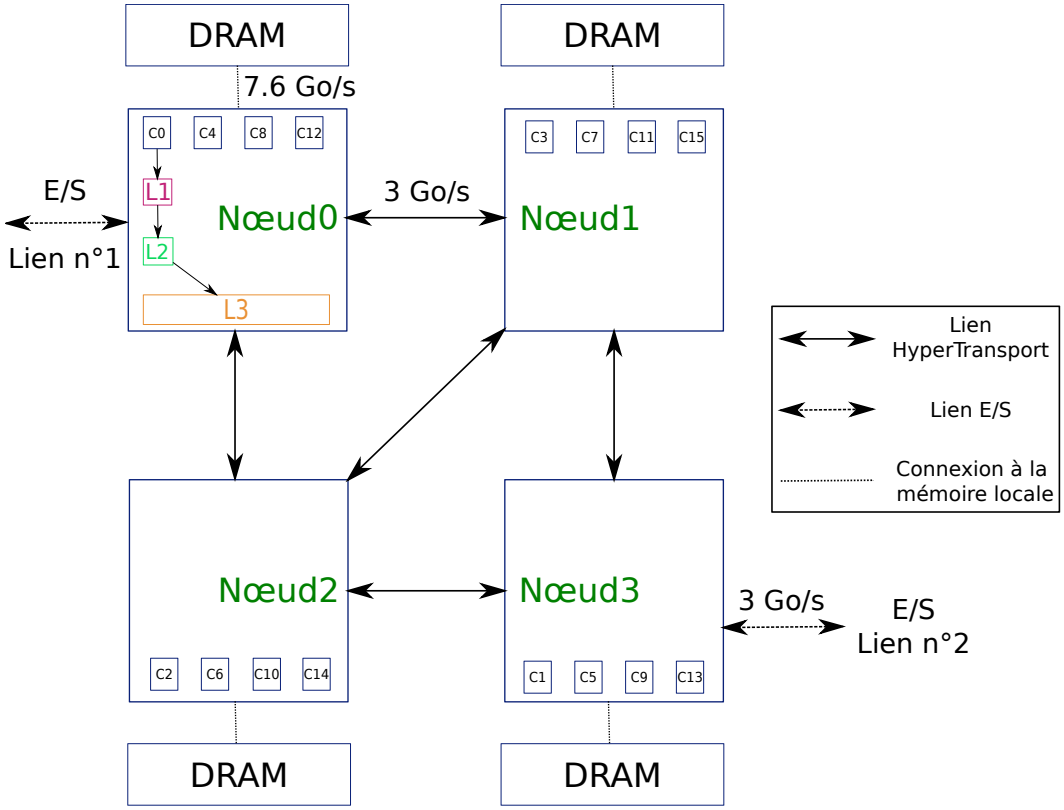


Figure 6.1 – Illustration des liens entre les 4 processeurs AMD Opteron 8380 à quatre cœurs, les nœuds mémoire et les périphériques d'E/S.

6.4.2 Configuration logicielle

Nous utilisons comme système d'exploitation une distribution Linux Debian Lenny 5.0. Nous avons remplacé le noyau par défaut de cette distribution par un noyau Linux 2.6.32 plus récent. La bibliothèque standard utilisée est la *Glibc* version 2.7. Pour obtenir de meilleures performances, nous avons également modifié la configuration par défaut du système d'exploitation en ce qui concerne le nombre de fichiers maximum pouvant être ouverts, ainsi que la taille des tampons de réception et d'émission IP et TCP. Cette configuration a été établie empiriquement pour obtenir le maximum de débit de la part du serveur Web. Les valeurs des différents paramètres sont données dans le tableau 6.5.

Nous avons également changé l'affectation des requêtes d'interruption (IRQ) des cartes réseau aux différents cœurs. Par défaut, lorsqu'une carte déclenche une interruption, cette interruption est affectée à un cœur en fonction d'heuristiques d'équilibrage de charge gérées par le matériel. Nous avons constaté des baisses de performances dues à ce mécanisme et seul le premier cœur recevait les interruptions. Une autre solution est d'affecter statiquement les interruptions aux cœurs disponibles. Nous avons pu vérifier que cette solution donne de meilleures performances. Par conséquent, nous avons affecté une carte à chaque cœur et une carte supplémentaire à chaque nœud.

Nous utilisons pour ces expériences la version 2.2.14 du serveur Web Apache, ainsi que la version 5.2.12 de la machine virtuelle PHP.

Enfin, la répartition homogène des clients sur les différentes cartes réseau est assurée par un mécanisme de DNS. Pour cela, nous utilisons le serveur `Bind9` et nous le configurons de manière à retourner une nouvelle adresse IP à chaque nouvelle connexion grâce à une politique de tourniquet. Enfin, nous désactivons le cache DNS que maintient la machine virtuelle Java, utilisée par SPECWeb,

Paramètre	**Valeur**
fs.file-max	786762
net.core.netdev_max_backlog	400000
net.core.optmem_max	10000000
net.core.rmem_default	10000000
net.core.rmem_max	10000000
net.core.wmem_default	10000000
net.core.wmem_max	10000000
net.core.somaxconn	10000
net.ipv4.conf.all.rp_filter	1
net.ipv4.conf.default.rp_filter	1
net.ipv4.tcp_congestion_control	cubic
net.ipv4.tcp_ecn	0
net.ipv4.tcp_max_syn_backlog	10000
net.ipv4.tcp_max_tw_buckets	1800000
net.ipv4.tcp_mem	10000000 10000000 10000000
net.ipv4.tcp_rmem	10000000 10000000 10000000
net.ipv4.tcp_wmem	10000000 10000000 10000000
net.ipv4.tcp_sack	1
net.ipv4.tcp_syncookies	0
net.ipv4.tcp_timestamps	1
net.ipv4.tcp_tw_recycle	1
net.ipv4.tcp_fin_timeout	1

TABLE 6.5 – Principaux paramètres de configuration du système d'exploitation.

pour garantir une répartition homogène constante.

6.4.3 Méthodologie et métriques utilisées

Pour déterminer les sources de problème de passage à l'échelle d'Apache, nous allons utiliser dans la suite de ces travaux un ensemble de métriques. Plusieurs hypothèses peuvent expliquer une dégradation de performances. La première hypothèse est que la charge injectée par les clients n'est pas suffisante pour utiliser complètement les 16 cœurs du serveur. La seconde hypothèse est que le système d'exploitation ou l'interconnexion des cartes réseau n'est pas capable de gérer 20 Gb/s. Cela peut venir d'un problème de performance de la pile TCP/IP du noyau ou d'un problème matériel comme la contention sur les bus PCI-Express. La troisième hypothèse est que l'application ou le système d'exploitation ne passe pas à l'échelle à cause de l'utilisation de verrous. La quatrième hypothèse est qu'à cause de l'architecture NUMA, les accès mémoire à 16 cœurs soient principalement des accès distants, réduisant ainsi leur efficacité. Enfin, la dernière hypothèse possible est que l'architecture matérielle ne permet pas d'atteindre le débit de 20 Gb/s. En effet, nous avons vu précédemment que les liens HyperTransport sont capables d'absorber cette charge théoriquement. Toutefois, sur ces liens transitent, en plus des paquets réseau, des données applicatives (par exemple la communication entre un processus Apache et un processus PHP situés sur deux processeurs différents) ainsi que des messages dus au protocole de cohérence de cache. Notons que dans le cas de Veal et Foong [95], la saturation du bus d'adresses est la cause principale de la limitation du passage à l'échelle. Nous allons détailler dans cette section chacune des métriques que nous allons utiliser par la suite pour confirmer ou infirmer chacune de ces hypothèses, en expliquant la procédure employée pour les obtenir ainsi que leur intérêt.

Utilisation des cartes réseau. Le premier paramètre que nous considérons dans notre analyse est le taux d'utilisation des cartes réseau durant l'expérience. Pour cela, nous utilisons l'outil de profilage `sysstat` [49] qui permet d'obtenir périodiquement un ensemble d'informations sur l'état de la machine (nombre d'octets transmis et reçus sur chacune des cartes, taux d'utilisation des processeurs, etc.). L'objectif de cette mesure est de vérifier si les cartes réseau sont le goulot d'étranglement.

Nous devons tout d'abord connaître le débit maximum atteignable avec nos 20 cartes réseau. Pour cela, nous avons écrit un micro-test simulant le comportement de SPECWeb, basé sur l'observation de ce dernier. Dans ce micro-test, le débit en réception est comparable au débit observé avec SPECWeb et le débit en émission a pour objectif de saturer au maximum les cartes réseau. Les résultats de ce micro-test sont présentés dans le tableau 6.6. Ce tableau montre très clairement que (i) le débit maximum atteignable par carte est 940 Mb/s et le débit total atteignable est 18804 Mb/s, (ii) les 16 cœurs sont largement capables de gérer le débit sortant des 20 cartes réseau, et (iii) la pile TCP/IP de Linux ne semble pas avoir de problème de passage à l'échelle.

Nombre de cœurs	Débit total observé
1	6070 Mb / s
4	15140 Mb / s
8	18816 Mb / s
16	18804 Mb / s

TABLE 6.6 – Débit total observé en fonction du nombre de cœurs sur un micro-test simulant une injection SPECWeb.

Dans la suite de ces travaux, nous nous intéressons au taux d'utilisation des cartes réseau. Ce taux est calculé en fonction des observations effectuées à l'aide de `sysstat` et en comparant ces

résultats avec le débit maximum présenté dans le tableau 6.6. Nous présentons par la suite le taux moyen d'utilisation des interfaces réseau durant l'expérience ainsi que le taux moyen de la carte la plus chargée pour déterminer si la charge entre les cartes est correctement répartie.

Utilisation des processeurs. La seconde métrique que nous étudions est le taux d'utilisation des processeurs durant l'expérience. Tout comme précédemment, nous utilisons dans ce but l'outil de profilage `sysstat`. L'objectif de cette mesure est de vérifier que tous les cœurs du serveur sont complètement chargés. En effet, si tous les cœurs ne sont pas chargés alors que le réseau n'est pas limitant, cela peut induire que la charge est mal répartie entre les cœurs ou que les clients ne sont pas capables d'injecter suffisamment de charge sur le serveur.

Répartition du temps passé dans les fonctions. La troisième métrique que nous utilisons est le nombre de cycles passés dans chaque fonction du système. L'obtention de cette valeur est effectuée à l'aide d'Oprofile [3], un outil d'échantillonage capable d'exploiter les compteurs de performance fournis par le processeur. Pour chaque compteur, un seuil est défini et lorsque le compteur atteint ce seuil, une interruption est déclenchée. Oprofile regarde simplement quelle fonction est en cours d'exécution au moment de l'interruption. Il est possible avec cet outil d'obtenir des statistiques par cœur.

En étudiant cette métrique, il est possible de vérifier si le temps passé dans les fonctions normalisé par le nombre d'octets transmis augmente lorsque le nombre de cœurs utilisés augmente. Cela peut être dû à plusieurs raisons : une diminution de l'efficacité des accès mémoire, l'utilisation de verrous ou encore l'utilisation de fonction dont le temps dépend du nombre de cœurs (par exemple des fonctions en $O(n)$, n étant le nombre de cœurs).

Nombre d'instructions par cycle. Le nombre d'instructions exécutées par cycle (appelé également IPC, *Instructions per cycle*) est un bon indicateur de la performance des accès mémoire. Si le nombre d'instructions exécutées par cycle baisse, cela implique que le temps dans lequel le processeur attend des données depuis la mémoire augmente et donc que les accès mémoire sont moins performants. Sur une application idéale, capable d'utiliser parfaitement les registres ainsi que les pipelines, chaque cœur des processeurs AMD que nous utilisons est capable de traiter jusqu'à 3 instructions par cycle.

Pour obtenir cette métrique, nous avons développé un outil de mesure capable de récupérer périodiquement la valeur des compteurs fournis par chaque processeur. Cette obtention est effectuée en utilisant l'API *Linux Performance Events* fournie par le noyau Linux depuis sa version 2.6.31. L'avantage de notre outil par rapport à Oprofile est qu'il permet de regarder l'évolution temporelle des compteurs surveillés. Il permet également d'observer plusieurs compteurs simultanément (jusqu'à 4 sur notre architecture). Cette dernière possibilité est également fournie par Oprofile mais, en pratique, nous avons eu des problèmes pour observer plusieurs compteurs simultanément lorsque la séparation par CPU est activée. Contrairement à Oprofile, le principal inconvénient de notre outil est qu'il ne permet pas de séparer les mesures des compteurs par fonction, puisqu'il s'exécute périodiquement et non pas en fonction de la valeur des compteurs. Par conséquent, il n'est pas possible de détecter avec cet outil un problème ponctuel d'une fonction, mais uniquement de regarder l'état global du système. Notons que nous pouvons cependant séparer l'IPC de l'application de celui du noyau.

Pour obtenir le nombre d'instructions par cycle, nous avons besoin de combiner deux compteurs de performance fournis par le processeur : le compteur de cycle (`CPU_CLK_UNHALTED`) et le compteur du nombre total d'instructions exécutées (`RETIRED_INSTRUCTIONS`).

Accès mémoire distants. Le nombre d'accès mémoire distants est également une métrique intéressante et complémentaire de l'IPC. En effet, si lors du passage de 4 à 16 cœurs la répartition entre les accès locaux et les accès distants change et que le nombre d'accès distants augmente sensiblement, cela va provoquer une réduction de l'IPC à 16 cœurs. Pour récupérer cette valeur, nous utilisons cette fois encore notre outil de surveillance des compteurs du processeur. Le compteur `CPU_DRAM_REQUEST_TO_NODE` permet de connaître le nombre de requêtes mémoire générées par un cœur vers un nœud donné.

Utilisation des liens HyperTransport. La dernière métrique que nous utilisons est le taux d'utilisation des liens HyperTransport à 16 cœurs. Cette métrique est également complémentaire à l'IPC et au nombre de requêtes distantes. En effet, une baisse de l'IPC peut s'expliquer par une saturation d'un ou de plusieurs liens HyperTransport. Pour obtenir le taux d'utilisation, nous avons combiné plusieurs compteurs de performance en utilisant notre outil de mesures des compteurs de performances. Grâce au compteur `HYPERTRANSPORT_LINKX_TRANSMIT_BANDWIDTH`, où X est le lien à surveiller, il est possible de savoir si le lien est actif ou non à chaque coup d'horloge. Nous pouvons donc obtenir la fréquence maximum du lien en regardant le nombre de coups d'horloge pendant lesquels celui-ci est complètement inactif. En comparant le nombre de fois où le lien est actif à sa fréquence maximale, nous pouvons déduire le taux courant d'utilisation du lien HyperTransport.

Méthodologie. Dans la suite de ce chapitre, pour chacune des expériences, nous allons suivre la même méthodologie d'expérience. La première étape de l'évaluation consiste à mesurer le passage à l'échelle du serveur Web dans la configuration donnée. Ensuite, pour expliquer l'évolution des performances, nous allons présenter l'état de chacun des indicateurs présentés ci-dessus. Nous allons commencer tout d'abord par nous assurer que le réseau n'est pas limitant et que tous les cœurs sont pleinement utilisés. À l'aide des quatre autres métriques, nous essayons de déterminer si le problème est dû aux accès à la mémoire ou à des problèmes logiciels (e.g. prise de verrous). L'objectif de ces mesures systématiques est de pouvoir évaluer l'impact de chacune de nos propositions sur tous les indicateurs que nous avons choisis.

6.5 Étude de la performance du serveur Web Apache sur les architectures multi-cœurs

Après avoir décrit dans les sections précédentes le contexte matériel et logiciel de ces tests, nous étudions dans cette section les performances du serveur Web Apache sur l'architecture NUMA à 16 cœurs considérée.

6.5.1 Mesure de performances initiale

La figure 6.2 présente le nombre maximum de clients pouvant être traités simultanément tout en conservant 95 % des clients servis dans un temps considéré comme bon et 99 % dans un temps considéré comme raisonnable. Les mesures ont été effectuées plusieurs fois et l'écart type est très faible. Cette figure montre également les débits obtenus avec un passage à l'échelle idéal [3].

3. Notons que nous calculons le passage à l'échelle idéal en fonction du débit obtenu à 4 cœurs et non pas à 1 cœur. En effet, comme nous l'expliquons dans la section 6.5.5.1, calculer le passage à l'échelle à partir du débit obtenu à un cœur est utopique, car, dans cette configuration, celui-ci dispose de l'intégralité du cache L3 alors qu'à 16 cœurs le cache L3 est partagé entre les 4 cœurs du processeur.

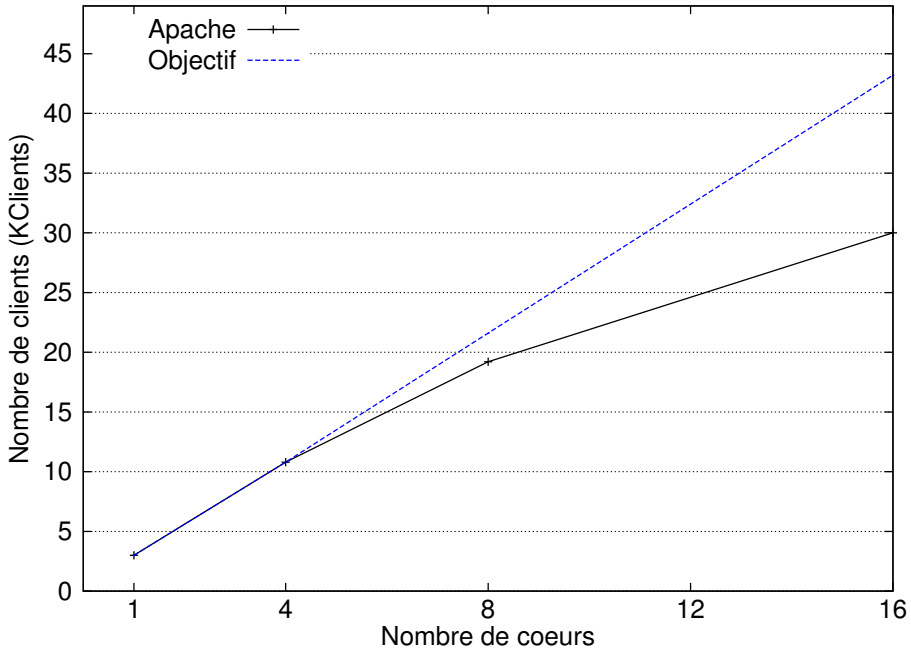


Figure 6.2 – Nombre maximum de clients pouvant être traités pour la charge SPECWeb 2005 Support.

Nous pouvons constater sur cette figure qu'Apache améliore ses performances avec l'augmentation du nombre de cœurs, mais n'a pas un passage à l'échelle idéal sur la charge SPECWeb Support. En effet, nous pouvons observer une baisse des performances du serveur de 32 % par rapport au passage à l'échelle idéal. Nous avons également mesuré que la dégradation du débit par cœur est linéaire et donc qu'il n'y a pas une limite à partir de laquelle la dégradation s'amplifie.

6.5.2 Analyse de l'utilisation des cartes réseau

Nous analysons tout d'abord le taux d'utilisation des cartes réseau pour vérifier si une ou plusieurs interfaces peuvent être le goulot d'étranglement du système. Le tableau 6.7 présente les taux d'utilisation moyen et maximal des cartes réseau observés durant une injection de charge SPECWeb sur le serveur Web Apache. Nous pouvons effectuer plusieurs observations à partir de ce tableau. La principale observation que nous pouvons faire est que le débit soutenu par le serveur Web dans cette configuration est largement inférieur au débit maximum atteignable avec nos 20 cartes réseau. De même, aucune carte n'est en état de saturation, le taux maximum d'utilisation d'une carte réseau étant de 63 %. Enfin, nous pouvons remarquer que la charge est bien équilibrée entre les interfaces réseau.

Nombre de cœurs	Taux moyen d'utilisation	Taux maximal d'utilisation
1	6 %	7 %
4	20 %	22 %
8	35 %	38 %
16	59 %	63 %

TABLE 6.7 – Utilisation moyenne et maximale des cartes réseau pour la charge SPECWeb Support.

6.5.3 Analyse de l'utilisation des différents cœurs

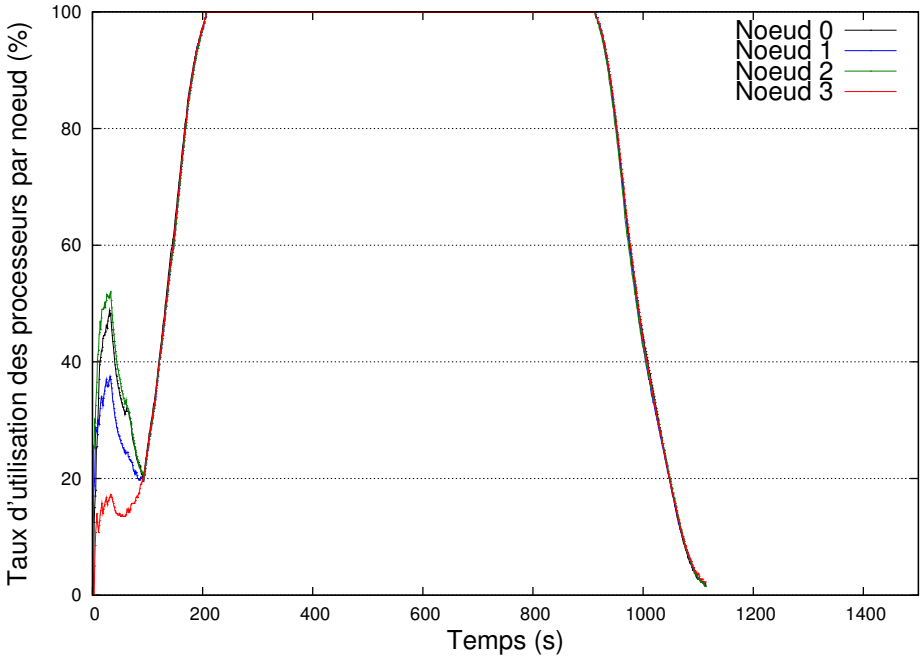


Figure 6.3 – Utilisation des cœurs regroupés par nœuds sur le test SPECWeb Support à 16 cœurs.

Nous avons ensuite évalué l'utilisation des cœurs durant l'expérience. L'objectif de cette évaluation est de vérifier si les cœurs sont complètement chargés. La figure 6.3 présente la charge moyenne par nœud observée au cours de l'expérience SPECWeb Support à 16 cœurs. Nous rappelons qu'un nœud comporte 4 cœurs. Nous présentons la charge agrégée par nœud pour conserver une figure lisible.

Nous pouvons voir sur cette figure que, après la phase de montée en charge du serveur, tous les cœurs sont utilisés à 100 %. Cela élimine donc l'hypothèse du manque d'injection de charge ou du déséquilibre de charge entre les cœurs. Cette situation, qui présente une saturation des cœurs sans passage à l'échelle, peut être expliquée par deux hypothèses. La première est que les fonctions utilisent des verrous à attente active qui augmentent ainsi sensiblement le temps passé dans les fonctions. La seconde hypothèse est que les accès mémoire sont plus lents et accentuent ainsi les temps d'obtention des données par le processeur.

6.5.4 Analyse du temps passé dans les fonctions

Pour déterminer si certaines fonctions ne passent pas à l'échelle, nous comparons l'augmentation du nombre de cycles et d'instructions par octet transmis entre 4 et 16 cœurs. Le tableau 6.17 présente à 16 cœurs, pour les dix principales fonctions, la proportion de temps moyen passé dans chacune de ces fonctions ainsi que l'augmentation de ce temps entre 1 et 16 cœurs. Les fonctions sont triées en fonction du pourcentage de temps total passé dans la fonction.

Ce tableau révèle que trois fonctions ont un coût qui augmente sensiblement avec l'augmentation du nombre de cœurs. La première de ces fonctions est `__d_lookup`. L'augmentation sensible du temps passé dans cette fonction est liée à son utilisation de verrous. Cette fonction fait partie du système de fichiers virtuel de Linux (`VFS`) et est appelée à chaque fois qu'un nouveau fichier est ouvert, ce qui est très fréquent dans le serveur Web. Ainsi, pour chaque fichier ouvert, le noyau prend

Fonction	Temps passé dans la fonction	Cycles / octet transmis	Augmentation constatée
`__d_lookup`	5.15 %	0.166	398 %
`/e1000e`	3.39 %	0.109	18 %
`memcpy`	3.35 %	0.108	17 %
`_atomic_dec_and_lock`	2.57 %	0.083	523 %
`_zend_mm_alloc_int`	1.92 %	0.062	30 %
`ap_merge_per_dir_configs`	1.84 %	0.059	27 %
`copy_user_generic_string`	1.68 %	0.054	5 %
`lookup_mnt`	1.41 %	0.045	907 %
`apr_palloc`	1.18 %	0.038	31 %
`memcmp`	1.09 %	0.035	43 %

TABLE 6.8 – Nombre de cycles par octet transmis dans les dix principales fonctions ainsi que leur augmentation par rapport à l'exécution sur quatre cœurs.

un verrou sur le dossier dans lequel est placé le fichier. La fonction `_atomic_dec_and_lock` permet, comme son nom l'indique, une prise de verrou et ne peut pas par définition passer à l'échelle. Enfin, la fonction `lookup_mnt` est également une fonction du `VFS` qui permet de savoir quels sont les périphériques associés à un point de montage. Cette fonction protège également les structures de données manipulées des accès concurrents en utilisant un verrou à attente active.

Toutefois, malgré ces goulots d'étranglement potentiels, seule une petite fraction du temps total est passée dans ces fonctions. Cela nous permet de dire que, si le problème de passage à l'échelle de certaines fonctions existe, il ne s'agit pas du problème principal.

Enfin, nous pouvons constater que le coût moyen de toutes les fonctions augmente légèrement. Cette constatation semble indiquer que les accès mémoire sont moins efficaces à 16 cœurs qu'à 1 cœur.

6.5.5 Analyse de l'efficacité des accès mémoire

Pour analyser la performance des accès mémoire, nous disposons de deux indicateurs : le nombre d'instructions exécutées par cycle ainsi que la charge observée sur les liens HyperTransport qui connectent les processeurs. Dans cette section, nous allons étudier ces deux paramètres.

6.5.5.1 Étude du nombre d'instructions exécutées par cycle

Le tableau 6.9 illustre l'IPC observé avec différentes configurations. L'IPC par nœud est la moyenne des IPC de ce nœud. Les cases grisées correspondent aux nœuds pour lesquels aucun cœur n'est actif.

Cette évaluation du nombre d'instructions par cycle processeurs nous permet de faire plusieurs observations. Premièrement, Apache a une efficacité mémoire relativement faible, car une instruction est exécutée au mieux tous les deux cycles ce qui est loin de l'optimal. Deuxièmement, le nombre d'instructions exécutées par cycle baisse lorsque le nombre de cœurs augmente. Cela implique que les accès à la mémoire sont moins efficaces à 16 cœurs qu'à un cœur. Troisièmement, on peut également distinguer une asymétrie dans les performances d'accès mémoire. En effet, nous pouvons voir que l'IPC des nœuds 0 et 3 est plus faible que celui des nœuds 1 et 2. Cette asymétrie est due à l'interconnexion entre les processeurs. En effet, comme nous l'avons expliqué précédemment, les nœuds 0 et 3

Nombre de cœurs	IPC			
	Nœud 0	Nœud 1	Nœud 2	Nœud 3
1	0.47			
4 sur le même nœud	0.36			
4 sur quatre nœuds différents	0.39	0.41	0.41	0.39
8 sur deux nœuds	0.31			0.31
16	0.3	0.34	0.32	0.29

TABLE 6.9 – Évolution du nombre d'instructions par cycle sur le serveur Web Apache en fonction du nombre de cœurs.

n'ont pas de lien HyperTransport les connectant directement. Par conséquent, la communication entre les nœuds 0 et 3 doit nécessairement passer par un nœud intermédiaire réduisant ainsi leur efficacité.

Pour expliquer le phénomène de baisse de l'IPC, nous pouvons formuler plusieurs hypothèses. Premièrement, avec un seul cœur, le cache L3 est totalement disponible pour ce cœur alors que lorsque plusieurs cœurs sont utilisés sur un même nœud, le cache L3 de ces nœuds est partagé entre les différents cœurs. Par conséquent, l'efficacité du cache L3 diminue lorsque le nombre de cœurs actifs sur le processeur augmente. Cela est confirmé par les différences observées dans l'IPC des configurations « 4 cœurs sur 4 processeurs différents » et « 4 cœurs sur le même processeur ». Nous avons également évalué le nombre de fautes de cache L3 par requête. Les résultats sont fournis dans le tableau 6.10 et montrent clairement que le partage du cache à un fort impact sur les fautes de cache L3 alors que l'augmentation du nombre de cœurs à un impact moindre. Il est important de noter que cette limitation matérielle ne peut pas être résolue par l'application, c'est pourquoi nous calculons le passage à l'échelle idéal à partir des résultats à 4 cœurs [4].

Nombre de cœurs	Fautes L3 / Requête	Augmentation / Valeur précédente	Ratio fautes / accès
1	43838		8 %
2 sur le même nœud	68597	56 %	12 %
3 sur le même nœud	79843	16 %	14 %
4 sur le même nœud	84819	6 %	15 %
8 sur deux nœuds	88528	4 %	15 %
16 sur quatre nœuds	92506	4 %	15 %

TABLE 6.10 – Nombre de fautes de cache par requête traitée en fonction du nombre de cœurs.

Deuxièmement, la communication sur les liens HyperTransport est une raison de la baisse de l'IPC. Comme nous l'avons montré dans le chapitre 1.4, l'accès à une donnée distante demande un temps plus important que l'accès à une donnée locale. Par conséquent, la réduction du nombre d'accès locaux au détriment du nombre d'accès distants implique donc une baisse de la performance globale des accès mémoire. Le tableau 6.11 présente le pourcentage d'accès mémoire effectué localement par rapport à la totalité des accès mémoire. Ces résultats montrent très clairement que, dès que plusieurs nœuds sont utilisés (configuration « 4 cœurs sur 4 processeurs différents »), le taux d'accès à la mémoire locale diminue fortement, ce qui a pour résultat une baisse significative de la performance des accès mémoire. Enfin, nous pouvons remarquer que plus le nombre de cœurs utilisés augmente,

4. La seule solution à ce problème consiste à ne plus utiliser le cache L3 et à faire tenir les données dans les caches L1 et L2, ce qui est difficile à obtenir avec un serveur Web à cause de la complexité du code et de la taille des données manipulées.

plus le taux d'accès mémoire locaux diminue.

Pour bien comprendre ces résultats, il convient de distinguer deux types d'accès mémoire véhiculés par les liens HyperTransport. Le premier type d'accès mémoire est effectué par le mécanisme de DMA des interfaces réseau lors de l'envoi du contenu des fichiers. Comme les fichiers sont toujours répartis de façon identique, le profil des accès mémoire effectué par les DMA ne change pas entre 4 et 16 cœurs. Les résultats présentés dans le tableau 6.11 ne prennent donc pas en compte les accès mémoires directement effectués par les cartes réseau. Le second type d'accès mémoire est effectué par les applications ou par le noyau du système d'exploitation. La localité de ces accès mémoire est dépendante du placement des processus (ou des threads) sur les cœurs disponibles, notamment si les processus communiquent entre eux comme c'est le cas de processus Apache et PHP.

Nombre de cœurs	**Pourcentage d'accès mémoire locaux**			
	Nœud 0	**Nœud 1**	**Nœud 2**	**Nœud 3**
1	81 %			
4 sur le même nœud	83 %			
4 sur quatre nœuds différents	54 %	48 %	38 %	40 %
8	44 %			51 %
16	40 %	37 %	30 %	39 %

TABLE 6.11 – Taux de requêtes vers un banc de mémoire locale par rapport à la totalité des accès mémoire.

6.5.5.2 Utilisation des liens HyperTransport

Nous avons également voulu déterminer si les liens HyperTransport sont un goulot d'étranglement pour l'application. La figure 6.4 présente le taux d'utilisation à 16 cœurs des différents liens HyperTransport. Notons qu'un lien HyperTransport est bidirectionnel. Nous pouvons observer sur cette figure que quatre liens HyperTransport sont chargés à plus de 50 % et un lien est chargé à plus de 80 %. L'utilisation moyenne des liens est de 48 %. Il est important de considérer que, dans leurs travaux, Veal et Foong ont affirmé qu'un bus chargé à plus de 2/3 peut être considéré comme saturé.

La charge importante de certains liens est due à l'interconnexion entre les processeurs. En effet, lorsque les cœurs du nœud 0 souhaitent communiquer avec les périphériques réseau ou avec les cœurs du nœud 3, les données vont transiter par le nœud 1. Inversement, lorsque les cœurs du nœud 3 vont communiquer avec les cœurs du nœud 0, les données vont transiter par le nœud 2 (voir section 6.4). C'est ce fonctionnement qui explique « l'anneau » de charge observé.

Cette caractéristique de notre architecture suscite la surcharge de certains des liens HyperTransport et peut donner lieu à une baisse de performance de 4 à 16 cœurs.

6.5.6 Bilan

Dans cette section, nous avons montré tout d'abord que le serveur Web Apache ne permet pas un passage à l'échelle idéal sur les architectures multi-cœurs. En analysant les traces d'exécution, nous avons pu déterminer trois problèmes. Le premier problème est l'augmentation du temps pris par certaines fonctions à cause de l'utilisation de verrous. Cependant, bien que ce problème limite le passage à l'échelle du serveur Web, il ne s'agit pas de la cause principale. Nous avons également mis en évidence que l'efficacité des accès mémoire diminue lorsque le nombre de cœurs augmente. Ce phénomène est dû à l'augmentation du nombre d'accès aux bancs de mémoire distants. Il augmente également fortement l'utilisation des liens HyperTransport, ce qui peut éventuellement saturer le lien

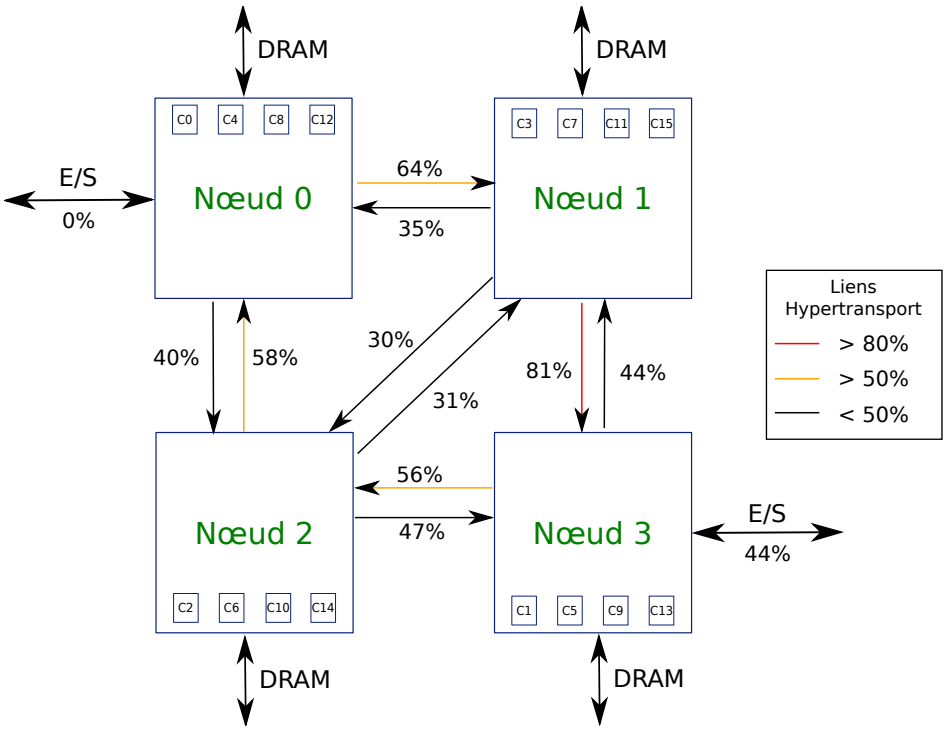


Figure 6.4 – Utilisation des liens HyperTransport à 16 cœurs.

entre les nœuds 1 et 3. À partir de ces observations, nous allons faire, dans la suite de ce chapitre, un ensemble de propositions visant à améliorer le passage à l'échelle du serveur Web Apache sur les architectures multi-cœurs.

6.6 Première étape : Réduction des accès distants en colocalisant les processus Apache et PHP

Notre premier objectif est d'améliorer l'efficacité des accès mémoire (IPC) pour réduire la contention sur les liens HyperTransport. Pour cela, il est nécessaire d'augmenter le taux de requêtes à la mémoire locale. Une hypothèse peut expliquer la charge importante sur les liens HyperTransport. Lors de la réception d'une requête pour une page dynamique, le processus Apache qui a reçu la requête doit communiquer avec l'un des processus PHP en utilisant le protocole FastCGI. Cependant, rien ne permet de garantir que le processus Apache va dialoguer avec l'un des processus PHP situé sur le même cœur ou sur le même nœud. Ce dialogue va, par conséquent, augmenter la pression déjà exercée par le trafic réseau sur les liens HyperTransport.

La première idée que nous allons évaluer consiste à placer sur le même cœur les processus Apache et les serveurs PHP qui leur sont liés. L'idée de cette solution est qu'un processus Apache ne doit dialoguer qu'avec un processus PHP local, c'est-à-dire s'exécutant sur le même nœud. L'objectif est de réduire la charge sur les liens HyperTransport en limitant la communication inter-nœuds.

Pour réaliser cette solution, nous allons utiliser une instance du serveur Web Apache par nœud NUMA. Chacune de ces instances est responsable de gérer les requêtes de cinq cartes réseau. Pour affecter un ensemble de cœur à une instance d'Apache, nous utilisons les possibilités fournies par Linux avec l'outil `taskset`. La répartition équitable des requêtes entre les interfaces, et donc entre les nœuds, est toujours assurée par le DNS, comme nous l'avons décrit dans la partie 6.4. Pour une

instance donnée d'Apache, les IRQ associées aux cartes sont réparties sur les 4 cœurs du nœud associé à l'instance.

L'approche que nous avons choisie ressemble à l'approche *N-Copy* que nous avons décrite dans le chapitre 5. En effet, nous avons une instance du serveur Apache par nœud NUMA. Dans la suite de ce document, nous utilisons ce terme pour désigner cette première solution.

Un autre avantage de l'approche *N-Copy* est de permettre la duplication d'une partie des fichiers, ce qui n'est pas possible avec le serveur Web Apache d'origine (puisque toutes les processus et threads Apache servent le même ensemble de fichiers). La duplication de ces fichiers permet d'améliorer notamment la localité d'accès, ainsi que de réduire la contention sur les verrous pris par le système de fichiers virtuel. Comme il n'est pas possible de dupliquer l'intégralité des fichiers, puisque les fichiers disponibles au téléchargement ont une taille totale de plus de 11 Go, nous avons choisi de dupliquer les images, car ce sont les fichiers les plus fréquemment demandés, ainsi que les scripts, pour maximiser la localité d'exécution de l'interpréteur PHP.

6.6.1 Mesure de la performance de la solution

La figure 6.5 présente les résultats obtenus avec cette solution sur la charge SPECWeb 2005 Support. La version *N-Copy* correspond à la version dans laquelle les processus Apache et PHP sont colocalisés. Sur cette figure, nous pouvons observer que le fait d'obliger un processus Apache à dialoguer avec un processus PHP situé sur le même nœud permet d'augmenter légèrement le débit à 16 cœurs par rapport à la version normale d'Apache. En effet, nous pouvons voir une augmentation de performances de 8 %. Toutefois, le gain apporté par cette solution n'est que très superficiel. Par conséquent, les résultats sont encore éloignés du passage à l'échelle idéal.

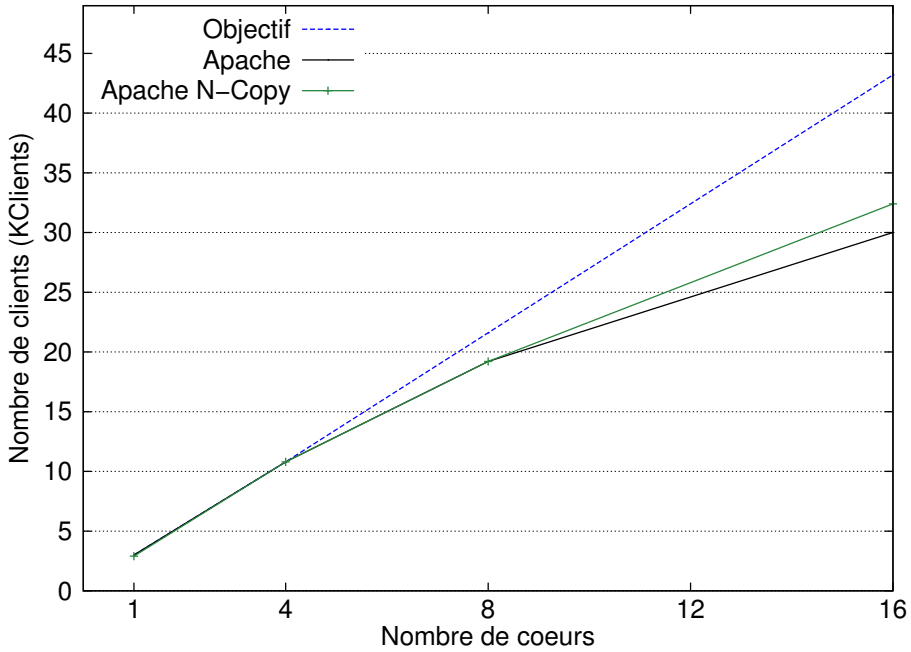


Figure 6.5 – Nombre maximum de clients pouvant être traités pour la charge SPECWeb 2005 Support avec la configuration N-Copy.

6.6.2 Analyse de l'utilisation des cartes réseau

Nous avons ensuite mesuré le débit sortant total ainsi que celui de chacune des cartes réseau. Nous mesurons uniquement les débits réseau des configurations à 8 et 16 cœurs, car la colocalisation des processus PHP et Apache ne change pas la configuration à 1 et 4 cœurs (pour ces 2 configurations, les mesures ont été données dans le tableau 6.7). Les résultats sont présentés dans le tableau 6.12. Ces mesures montrent très clairement que le débit moyen observé est augmenté de 7 % par rapport à la version classique du serveur Web Apache. Toutefois, le débit atteint est encore 33 % moins important que le débit pratique maximal. On peut remarquer que l'augmentation du débit moyen est légèrement supérieure à l'augmentation du nombre de clients pouvant être traités par seconde. Cela est dû au fait qu'avec la version *N-Copy*, le taux de satisfaction est de 99 % de requêtes bonnes et de 100 % de requêtes tolérables alors qu'avec la version classique d'Apache, ces taux sont de 96 % de requêtes bonnes et de 100 % de requêtes tolérables.

Nombre de cœurs	Taux moyen d'utilisation	Taux maximal d'utilisation
8	41 %	42 %
16	71 %	74 %

TABLE 6.12 – Taux d'utilisation moyen et maximal des cartes réseau pour la charge SPECWeb Support en colocalisant les processus Apache et PHP avec la configuration N-Copy.

6.6.3 Analyse de l'utilisation des différents cœurs

Nous avons également mesuré le taux d'utilisation des différents cœurs. La figure 6.6 présente l'utilisation moyenne des cœurs sur chacun des nœuds. Le taux moyen d'utilisation d'un nœud est égal à la moyenne du taux d'utilisation des cœurs de ce nœud.

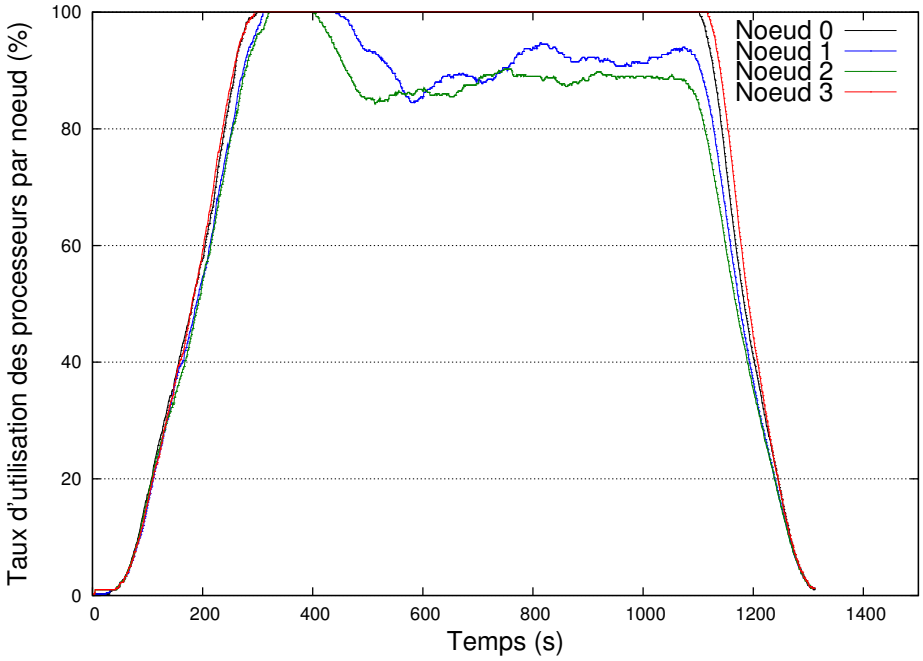


Figure 6.6 – Utilisation des cœurs regroupés par nœuds avec la configuration N-Copy.

Cette figure montre très clairement qu'il existe un déséquilibre de charge entre les nœuds. Certains nœuds deviennent inactifs alors que les autres sont toujours complètement chargés. Pourtant, la répartition homogène des requêtes sur les nœuds est assurée par le mécanisme de DNS. L'hypothèse vérifiée est donc que certains nœuds sont plus lents que d'autres et, comme SPECWeb est une injection de charge en boucle fermée, le taux d'utilisation maximum de ces nœuds est asservi au débit des nœuds les plus lents.

Cette figure montre que les nœuds 1 et 2 deviennent inactifs une fois la période de préchauffage passée. Par conséquent, avec notre hypothèse, ce sont les noeuds 0 et 3 qui sont les plus lents. Cela concorde avec les observations déjà effectuées. Les nœuds 0 et 3 sont plus lents, car il n'y a pas de lien HyperTransport entre ces nœuds. Par conséquent, l'accès à une zone mémoire distante depuis ces nœuds peut se faire en un ou deux sauts alors que depuis les nœuds 1 et 2, cet accès ne se fait qu'en un seul saut. De même, le protocole de cohérence de cache est plus coûteux pour les nœuds 0 et 3 (obligatoirement deux sauts) que pour les nœuds 1 et 2 (obligatoirement 1 saut). Le tableau 6.13 donne le débit mémoire observé en fonction du cœur accédant à cette donnée et du nœud sur lequel la donnée est stockée. Ce tableau montre sans ambiguïté que les cœurs appartenant aux nœuds 0 et 3 sont pénalisés pour les accès locaux (-24 %) à cause des deux sauts nécessaires au mécanisme de cohérence de cache. Nous pouvons également constater que les données placées sur les nœuds 0 et 3 sont plus lentes à manipuler depuis des cœurs distants. Cela est dû au mécanisme de cohérence de cache. En effet, c'est le contrôleur mémoire du nœud qui possède la donnée qui va déclencher le protocole de cohérence de cache. Enfin, lorsqu'un cœur du nœud 0 veut accéder à une zone mémoire placée sur le nœud 3, et réciproquement, le débit est pénalisé par les deux sauts nécessaires ainsi que par le mécanisme de cohérence de cache.

Nœud demandeur	**Nœud destination**			
	Nœud 0	**Nœud 1**	**Nœud 2**	**Nœud 3**
Nœud 0	3305 Mo/s	2830 Mo/s	2858 Mo/s	2052 Mo/s
Nœud 1	2587 Mo/s	4335 Mo/s	2848 Mo/s	2552 Mo/s
Nœud 2	2532 Mo/s	2886 Mo/s	4365 Mo/s	2607 Mo/s
Nœud 3	2051 Mo/s	2849 Mo/s	2846 Mo/s	3306 Mo/s

TABLE 6.13 – Débit mémoire en fonction du placement de la zone mémoire écrite. Un seul cœur par noeud accède à la mémoire.

Notre hypothèse est donc la suivante : les cœurs des nœuds 0 et 3 sont plus lents pour accéder à la mémoire que les autres cœurs. Par conséquent, le débit du serveur est limité au débit de ces cœurs à cause du modèle d'injection de charge. En effet, pour un client, la probabilité de tirer un cœur lent est égale à celle de tirer un cœur rapide. Par conséquent, le débit des cœurs rapides est asservi au débit des cœurs les plus lents, car les cœurs les plus lents ralentissent la production de nouvelles requêtes (modèle d'injection en boucle fermée). Toutefois, comme les cœurs les plus lents ne sont pas en surcharge, le nombre de requêtes sur chaque cœur n'est pas déséquilibré. Les cœurs les plus rapides les traitent simplement plus vite.

Il est important de noter que cette différence entre les nœuds rapides et les nœuds lents est également présente avec le serveur Web Apache de référence. Cependant, l'équilibrage de charge entre les nœuds est assuré par les mécanismes de placement de threads du noyau Linux ainsi que par le partage de la socket d'écoute des connexions entrantes entre tous les threads. L'utilisation de plusieurs instances introduit ces problèmes d'équilibrage de charge, car (i) les instances et l'ensemble de leurs threads sont fixées sur un nœud donné, ce qui empêche l'ordonnanceur du noyau d'équilibrer la charge efficacement, et (ii) chaque instance ne peut accepter que sur un sous-ensemble restreint d'interfaces réseau.

6.6.4 Analyse de l'efficacité des accès mémoire

L'objectif de la colocalisation d'Apache et de PHP est de réduire les accès distants et ainsi d'améliorer la performance des accès mémoire, ainsi que de réduire l'utilisation des liens HyperTransport. Par conséquent, dans cette section, nous avons étudié l'impact de cette nouvelle architecture du serveur Web Apache sur les différents indicateurs mémoire.

6.6.4.1 Étude du nombre d'instructions exécutées par cycle

Nous avons tout d'abord mesuré l'impact de cette nouvelle architecture sur le nombre d'instructions pouvant être exécutées par cycle processeur. Les résultats de cette analyse sont donnés dans le tableau 6.14.

Nombre de cœurs	**IPC**			
	Nœud 0	**Nœud 1**	**Nœud 2**	**Nœud 3**
8 sur deux nœuds	0.34			0.34
16	0.34	0.38	0.37	0.34

TABLE 6.14 – Évolution de l'IPC en fonction du nombre de cœurs avec la configuration N-Copy.

Ces résultats permettent de constater que notre proposition d'architecture pour le serveur Web Apache permet d'améliorer sensiblement les performances de ses accès mémoire (+14 % à 16 cœurs). Nous pouvons voir également que la différence de performance dans les accès mémoire est toujours visible entre les nœuds 0 et 3 et les nœuds 1 et 2, pour les raisons que nous avons expliquées dans la section 6.5.5.1. Enfin, nous pouvons observer que l'IPC moyen à 16 cœurs est proche de l'IPC à 4 cœurs, ce qui est dû une diminution de l'utilisation des liens HyperTransport, comme le montre le tableau 6.19. En effet, à 16 cœurs, le taux de requêtes mémoire locales est passé de 37 % à 74 % en moyenne, ce qui indique bien une localité accrue.

Nombre de cœurs	**Pourcentage d'accès mémoire locaux**			
	Nœud 0	**Nœud 1**	**Nœud 2**	**Nœud 3**
8	59 %			76 %
16	77 %	79 %	68 %	72 %

TABLE 6.15 – Taux de requête à une mémoire locale par rapport à la totalité des accès mémoire avec la configuration N-Copy.

6.6.4.2 Analyse de l'utilisation des liens HyperTransport

Nous avons également mesuré l'utilisation des liens HyperTransport avec notre nouvelle architecture. La figure 6.7 présente le taux d'utilisation des liens HyperTransport. Nous pouvons voir que l'utilisation de 20 instances du serveur Apache et le dialogue local avec PHP permettent de décharger sensiblement les liens HyperTransport. Le taux d'utilisation moyen des liens HyperTransport passe de 48 % à 36 % tandis que l'utilisation maximum des liens descend de 81 % à 55 %. Grâce à cette solution, le seul lien chargé à plus de 50 % est le lien HyperTransport reliant les nœuds 1 et 3.

6.6.5 Bilan

Dans cette section, nous avons évalué une solution consistant à colocaliser les processus Apache et PHP. L'objectif de cette colocalisation est d'améliorer la performance des accès mémoire à 16

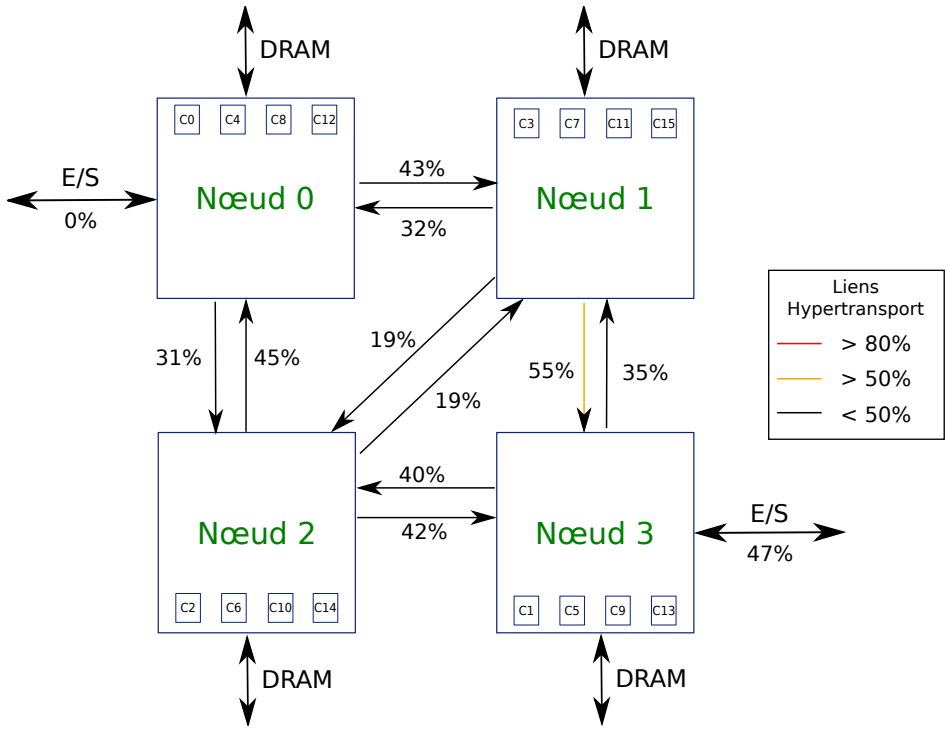


Figure 6.7 – Utilisation des liens HyperTransport à 16 cœurs avec la configuration N-Copy.

cœurs en diminuant le nombre d'accès vers des bancs de mémoire distants. En effet, le coût d'un accès distant est supérieur à celui d'un accès local. De plus, cette solution a tendance à décharger les liens HyperTransport. Cependant, nous avons montré que l'asymétrie de l'interconnexion entre les processeurs de notre architecture ne permet pas une utilisation efficace des processeurs. En effet, certains cœurs sont plus lents que d'autres et limitent le débit atteignable par les cœurs rapides. Par conséquent, la prochaine étape de nos travaux est de permettre l'équilibrage de la charge en tenant compte de cette asymétrie.

6.7 Deuxième étape : Équilibrage de la charge entre les nœuds en N-Copy

Cette étape est complémentaire à la première étape. Avec cette approche, nous avons choisi d'éviter le déséquilibre entre les cœurs en considérant la machine comme une boîte noire sur laquelle la seule façon d'augmenter la charge sur les cœurs est d'augmenter la charge sur leurs interfaces réseau. Pour l'instant, la charge sur les interfaces est distribuée équitablement entre les cœurs grâce à un DNS retournant une adresse IP utilisant un mécanisme de tourniquet. Une solution pour charger de manière plus importante les cœurs les plus rapides est d'affecter des pondérations plus importantes aux adresses IP liées à ces cœurs. Cependant, cette solution a deux défauts principaux. Le premier est que les pondérations sont fixées statiquement sans informations sur la charge courante sur le serveur. Par conséquent, si le type de charge évolue durant le cycle de vie du serveur, il est impossible de réévaluer ces pondérations. Deuxièmement, il est très compliqué de décider comment affecter les pondérations aux différents cœurs. En effet, nous avons classé les cœurs en deux groupes : rapides et lents. En fait, dans ces groupes de cœurs des différences peuvent être présentes. Par exemple, les cœurs du nœud 3 sont légèrement plus efficaces que les cœurs du nœud 0. Il est donc complexe de

trouver des pondérations efficaces, car elles sont très dépendantes de l'architecture matérielle ainsi que du type de charge sur le serveur.

Une autre solution est d'utiliser un mécanisme de proxy. Avec l'utilisation d'un tel proxy, il est possible de charger plus fortement les interfaces liées aux cœurs les plus rapides. Un proxy peut prendre des décisions plus finement qu'un DNS, car il est conscient du nombre de requêtes en attente sur chaque interface et peut donc équilibrer la charge sur toutes les interfaces. Le problème avec ce type de proxy est qu'il doit être capable de gérer en émission et en réception le débit maximal atteignable par le serveur. Dans le cas contraire, il est nécessaire de dupliquer le proxy.

L'utilisation d'un proxy pour équilibrer la charge entre les cœurs est donc une solution meilleure que l'emploi d'un DNS, mais également bien plus coûteuse. Comme nous ne disposons pas d'une autre machine suffisamment puissante pour jouer le rôle de proxy, nous avons choisi de modifier le client SPECWeb 2005 pour qu'il soit responsable de l'équilibrage de charge entre les cœurs. Cette solution est donc conceptuellement identique à un proxy. Au lieu d'effectuer une requête DNS, le client choisit dans une liste d'adresse IP connue celle qui convient en comparant le nombre de requêtes en attente et en retournant la connexion avec le plus petit nombre de requêtes en attente. Notons que cet équilibrage n'est actif qu'après la phase de création de tous les clients et de montée en charge du serveur (*ramp-up*) car ces phases sont utilisées pour initialiser les variables internes du proxy.

Il est important de considérer que cette solution peut être intégrée dans une infrastructure déjà existante, dans laquelle des proxys équilibrent les requêtes sur les nœuds. L'originalité de cette proposition consiste dans la prise en compte de la topologie d'interconnexion entre les processeurs dans l'équilibrage de charge. La modification des clients, pour qu'ils fassent le choix de l'interface à contacter, est effectuée uniquement pour des raisons de simplicité.

6.7.1 Mesure de la performance de la solution

La figure 6.8 présente les résultats obtenus lorsque le client équilibre la charge sur le serveur en fonction de la charge présumée sur chacune des interfaces réseau. Les résultats obtenus montrent très clairement un gain significatif de performances à 16 cœurs. En permettant d'utiliser efficacement les cycles d'inactivités des processeurs les plus rapides, l'équilibrage de charge améliore les performances à 16 cœurs de 11 % par rapport à la version *N-Copy* d'Apache et de 20 % par rapport à la version de référence d'Apache.

6.7.2 Analyse de l'utilisation des cartes réseau

Les gains observés à 16 cœurs sont logiquement visibles sur le taux d'utilisation des interfaces réseau. L'utilisation moyenne observée des cartes réseau est présentée dans le tableau 6.16. Nous pouvons voir une légère augmentation du débit moyen par carte réseau (passage de 71 % à 73 %) ainsi qu'une augmentation du débit maximum (passage de 74 % à 78 %). Comme la carte la plus chargée n'est pas utilisée à son taux maximum, il n'est pas nécessaire d'affecter d'avantage de cartes réseau aux nœuds les plus rapides et moins aux nœuds les plus lents.

Nombre de cœurs	Taux moyen d'utilisation	Taux maximal d'utilisation
8	41 %	42 %
16	73 %	78 %

TABLE 6.16 – Utilisation moyenne et maximal des cartes réseau pour la configuration avec équilibrage de charge.

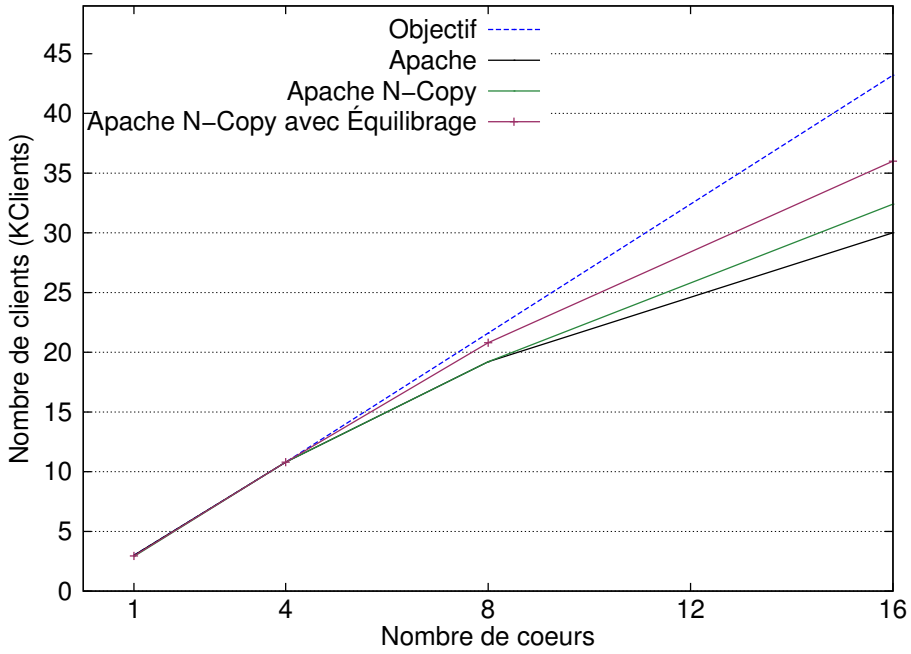


Figure 6.8 – Nombre maximum de clients pouvant être traités pour la charge SPECWeb 2005 Support avec et sans mécanisme d'équilibrage de charge.

6.7.3 Analyse de l'utilisation des différents cœurs

Nous avons ensuite analysé le taux d'utilisation des processeurs lorsque le mécanisme d'équilibrage de charge est présent. Les résultats que nous avons obtenus sont présentés dans la figure 6.9. Nous pouvons constater sur cette figure que le mécanisme d'équilibrage de charge permet de charger complètement les cœurs les plus rapides, les cœurs lents étant toujours saturés. Cette mesure est la confirmation que ce mécanisme est efficace puisque nous avons vu dans la section précédente que cette amélioration du taux d'utilisation des processeurs les plus rapides est suivie d'une amélioration des performances du serveur Web apache.

6.7.4 Analyse du temps passé dans les fonctions

Pour déterminer si certaines fonctions ne passent pas à l'échelle, nous comparons l'augmentation du nombre de cycles par octet transmis entre 4 et 16 cœurs. Le tableau 6.17 présente à 16 cœurs, pour les dix principales fonctions, le temps moyen passé dans chacune de ces fonctions ainsi que l'augmentation de ce temps entre 4 et 16 cœurs. Les fonctions sont triées en fonction du pourcentage de temps total passé dans cette fonction.

Nous pouvons voir avec ces résultats que parmi les dix principales fonctions du serveur Web, trois sont concernées par un problème de passage à l'échelle important. Comme nous l'avons décrit dans la section 6.5.4, la cause du problème de passage à l'échelle de ces fonctions est l'utilisation de verrous pour protéger les structures de données. Nous pouvons noter que l'utilisation de plusieurs instances a permis de réduire le temps passé dans la fonction `__d_lookup`. En effet, comme nous avons dupliqué une partie des fichiers (images, scripts PHP et fichier de configuration et de traces d'Apache), les verrous des dossiers contenant ces fichiers sont moins sollicités.

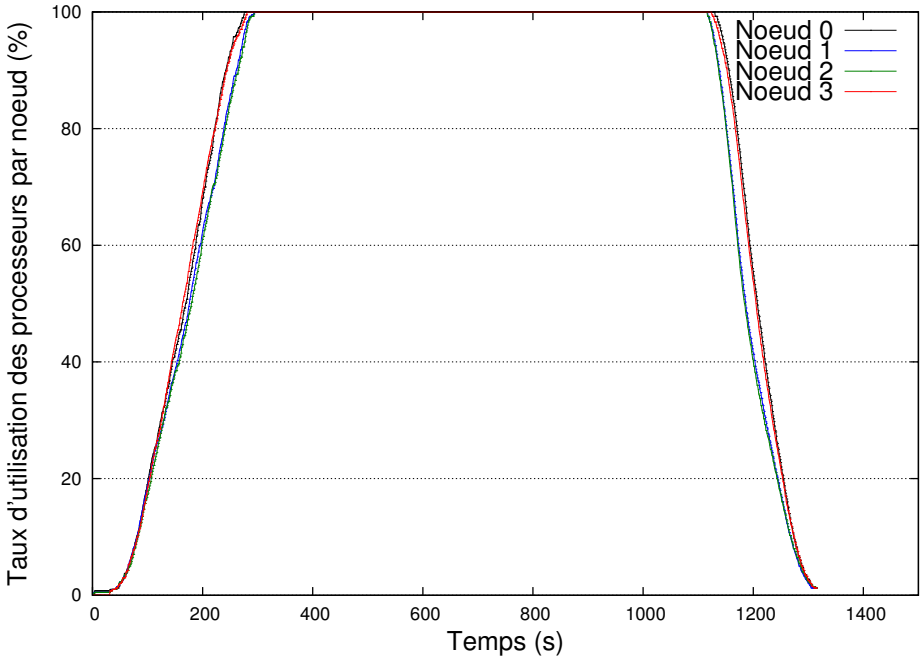


Figure 6.9 – Utilisation des cœurs regroupés par nœuds pour la configuration avec équilibrage de charge.

Fonction	**Temps passé dans la fonction**	**Cycles / octet transmis**	**Augmentation constatée**
`__d_lookup`	4.16 %	0.121	263 %
`memcpy`	3.92 %	0.114	24 %
`/e1000e`	3.33 %	0.097	5 %
`_atomic_dec_and_lock`	2.95 %	0.085	545 %
`copy_user_generic_string`	2.60 %	0.075	47 %
`_zend_mm_alloc_int`	2.04 %	0.059	25 %
`lookup_mnt`	1.57 %	0.046	910 %
`ap_merge_per_dir_configs`	1.48 %	0.043	-8 %
`strlen`	1.43 %	0.041	27 %
`apr_palloc`	1.08 %	0.031	8 %

TABLE 6.17 – Temps passé par octet transmis dans les dix principales fonctions ainsi que leur augmentation par rapport à l'exécution à quatre cœurs pour la configuration avec équilibrage de charge.

6.7.5 Analyse de l'efficacité des accès mémoire

Nous avons enfin étudié l'impact du mécanisme sur les performances des accès mémoire ainsi que sur l'utilisation des liens HyperTransport. L'objectif de cette étude est de vérifier que l'équilibrage de charge ne réduit pas les performances des accès mémoire en augmentant le nombre d'accès aux bancs mémoire distants ainsi que de vérifier que les liens HyperTransport ne sont pas en état de saturation.

6.7.5.1 Étude du nombre d'instructions exécutées par cycle

Nombre de cœurs	IPC			
	Nœud 0	Nœud 1	Nœud 2	Nœud 3
8 sur deux nœuds	0.34			0.34
16	0.34	0.38	0.37	0.34

TABLE 6.18 – Évolution de l'IPC en fonction du nombre de cœurs pour la configuration avec équilibrage de charge.

Nous avons tout d'abord mesuré la performance des accès mémoire. Ces résultats sont présentés dans le tableau 6.18. Nous pouvons voir, grâce à ces mesures, que, sans surprise, le fait d'équilibrer efficacement la charge entre les cœurs n'a pas d'impact négatif sur la performance des accès mémoire. La baisse de la performance des accès mémoire est relativement contenue grâce à la colocalisation des processus Apache et PHP. Nous pouvons voir dans le tableau 6.19 que le taux d'accès à la mémoire locale n'est que très faiblement diminué par l'équilibrage de charge (passage de 74 % à 71 % d'accès mémoire locaux en moyenne) et est probablement dû à l'incertitude des mesures.

Nombre de cœurs	Pourcentage d'accès mémoire locaux			
	Nœud 0	Nœud 1	Nœud 2	Nœud 3
8	64 %			78 %
16	82 %	73 %	65 %	65 %

TABLE 6.19 – Taux de requêtes à une mémoire locale par rapport à la totalité des accès mémoire avec la configuration N-Copy.

Cependant, il est nécessaire de tempérer ces résultats par deux observations. Premièrement, sur le cœur 0, l'IPC baisse toujours de 8 % à 16 cœurs par rapport à l'exécution sur 4 cœurs. Deuxièmement, le nombre d'instructions par cycle présenté est une moyenne sur chacun des nœuds. Le tableau 6.20 détaille l'IPC observé de chacune des fonctions et indique son évolution par rapport à l'IPC observé à 4 cœurs. Nous pouvons constater que les fonctions dont nous avons pointé les problèmes de passage à l'échelle dans la section 6.7.4 ont toutes un IPC en augmentation. Cela est dû à l'utilisation de verrous à attente active dans ces fonctions. En effet, lors de la prise du verrou, la fonction va passer un nombre important de cycles à vérifier l'état d'une valeur présente dans son cache, jusqu'à ce que cette valeur soit invalidée par la libération du verrou. Ce comportement est particulièrement visible sur la fonction `__d_lookup`. Cette augmentation artificielle du nombre d'instructions par cycle de certaines fonctions provoque par conséquent une augmentation de l'IPC global et masque une augmentation globale de la latence des accès mémoire.

6.7.5.2 Analyse de l'utilisation des liens HyperTransport

Pour comprendre l'impact du mécanisme d'équilibrage de charge sur la performance des accès mémoire, nous avons mesuré le taux d'utilisation des liens HyperTransport. Les résultats de cette

Fonction	IPC observé	Différence
`__d_lookup`	0.61	+91 %
`memcpy`	0.35	-16 %
`/e1000e`	0.14	-7 %
`_atomic_dec_and_lock`	0.43	+2 %
`copy_user_generic_string`	0.06	-12 %
`ap_merge_per_dir_configs`	0.11	-40 %
`_zend_mm_alloc_int`	0.78	+5 %
`strlen`	0.33	-37 %
`lookup_mnt`	1.09	+7 %
`memcmp`	0.13	-14 %

TABLE 6.20 – IPC observé des 10 principales fonctions pour la configuration avec équilibrage de charge.

observation sont présentés dans la figure 6.10. Nous pouvons voir sur cette figure que l'augmentation de la charge sur le serveur entraîne une augmentation de la charge sur les liens HyperTransport. Cette augmentation est due à la fois aux accès mémoire distants, ainsi qu'a l'augmentation du nombre d'accès locaux qui déclenchent plus fréquemment le protocole de cohérence de cache.

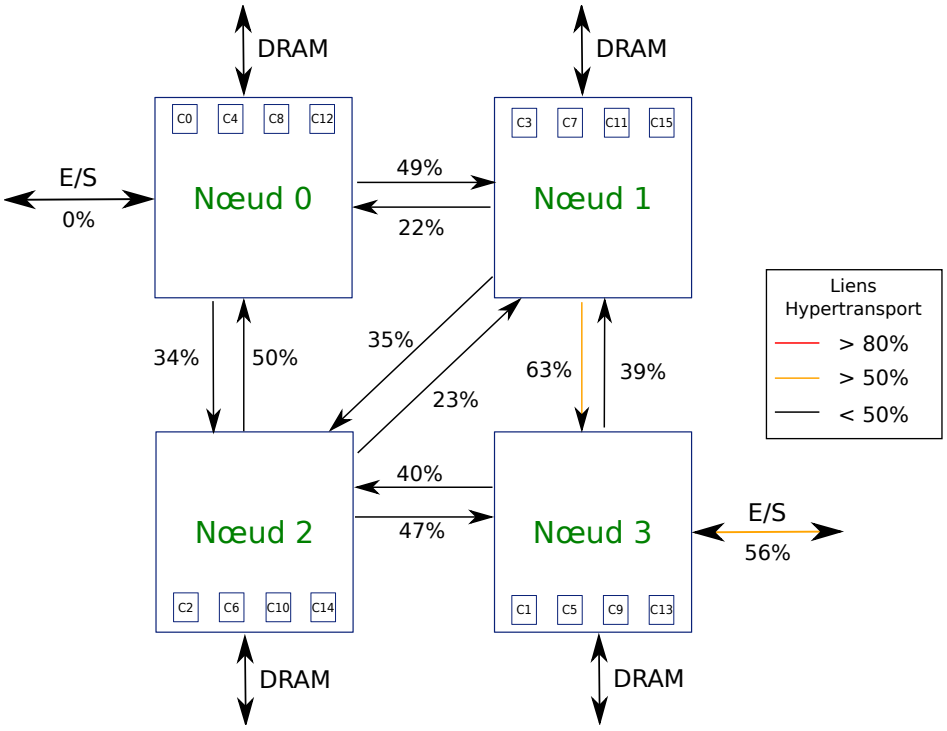


Figure 6.10 – Utilisation des liens HyperTransport à 16 cœurs pour la configuration avec équilibrage de charge.

6.7.6 Bilan

Dans cette section, nous avons étudié l'impact du mécanisme d'équilibrage de charge sur la performance du serveur Web. Ce mécanisme a pour rôle d'augmenter la charge sur les cœurs rapides afin de les exploiter complètement. Ce mécanisme améliore sensiblement les performances du serveur Web mais ne permet toujours par d'atteindre les performances de l'objectif. Nous avons ensuite étudié le comportement du serveur Web pour déterminer les causes de cette limitation. Nous avons conclu que deux problèmes principaux limitent le passage à l'échelle. Premièrement, bien que la colocalisation des processus Apache et PHP améliore sensiblement l'efficacité des accès mémoire, nous observons toujours une baisse considérable de celle-ci. Deuxièmement, certaines fonctions du noyau, notamment les fonctions liées au système de fichiers virtuel de Linux, utilisent des verrous pour se protéger des accès concurrents. Le temps passé dans ces verrous augmente considérablement lorsque le nombre de cœurs augmente, ce qui limite le passage à l'échelle.

6.8 Discussion

Les problèmes soulevés dans la section précédente sont difficiles à résoudre. Dans cette section, nous allons discuter des pistes potentielles pour proposer une solution efficace au problème de passage à l'échelle du serveur Web Apache.

Premièrement, bien que la colocalisation des processus Apache et PHP sur un même nœud ait réduit significativement, le nombre d'accès aux bancs mémoire distants représente en moyenne 30 % de la totalité des accès mémoire. Ces accès distants sont majoritairement effectués par le noyau et non pas par l'application, car chaque instance du serveur est fixée sur un nœud donné et ne dialogue qu'avec des serveurs PHP locaux [5]. Il n'y a donc pas de façon simple de les réduire.

La baisse de l'efficacité mémoire est également due à une augmentation de la latence des accès mémoire. Le tableau 6.21 présente l'évolution de la latence d'accès depuis le nœud 0 vers les différents nœuds par rapport à la version 4 cœurs. Nous pouvons constater à l'aide de ce tableau que le temps nécessaire pour récupérer une donnée augmente en moyenne de 37 % pour des données situées localement et de 18 % pour des données placées dans des zones mémoire distantes. La raison de cette croissance est l'augmentation de l'utilisation des liens HyperTransport due à la fois aux accès mémoire distants et à la cohérence de cache. Pour bien comprendre ces résultats, il est important de se rappeler que les accès mémoire locaux sont également impactés par la performance des liens HyperTransport puisqu'ils nécessitent l'envoi de plusieurs messages de cohérence de cache sur ces liens avant de pouvoir retourner la valeur. Diminuer le nombre d'accès distant permettra donc de réduire également l'augmentation de la latence pour les zones de mémoire locales. Cependant, il est probable que le mécanisme de cohérence de cache provoque toujours une augmentation de la latence mémoire entre 4 et 16 cœurs et ne permette ainsi pas d'atteindre le passage à l'échelle idéal.

Pour réduire les accès mémoire distants, une solution peut être de placer les fichiers dans la mémoire associée à un nœud donné et de n'autoriser que les processus Apache associés à ce nœud à servir ces fichiers. Dans ce cas, la manipulation de ces fichiers par le noyau (notamment l'examen des métadonnées) ne se fera que localement et ne nécessitera pas d'accès mémoire distant. Le principal problème de cette solution est la complexité de la répartition des fichiers sur les différents cœurs pour garder une charge équilibrée, car la popularité de ces fichiers est inégale et peut évoluer dans le temps. Une autre solution consiste à dupliquer l'intégralité des fichiers sur chacun des noeuds. Cependant, cette solution ne peut être mise en œuvre, car l'ensemble de fichiers obtenu ne tient pas en mémoire.

Deuxièmement, le problème de la prise de verrous est extrêmement complexe à résoudre. En effet, pour pouvoir enlever cette prise de verrous ou avoir des verrous à grain plus fins, il est nécessaire de

5. Notons que les instructions de ces deux programme peuvent être placées dans un banc de mémoire distant.

Nombre de cœurs	Évolution de la latence d'accès			
	Nœud 0	Nœud 1	Nœud 2	Nœud 3
8	12 %	-2 %	6 %	5 %
16	37 %	17 %	18 %	18 %

TABLE 6.21 – Augmentation de la latence observée pour des accès depuis le nœud 0 vers les différents nœuds par rapport à la version 4 cœurs.

connaître précisément l'implication de chacun de ces verrous, ce qui peut être difficile à déterminer. De plus, avoir cette connaissance ne permet pas forcément de trouver une autre solution. Dans le cas de `__d_lookup`, le verrou est utilisé pour protéger le déplacement de fichier. Une alternative à la prise de ce verrou pourrait être de mettre les fichiers statiques sur une partition en lecture seule, ce qui évite de devoir prendre un verrou pour ces fichiers. Cependant, cette méthode ne concerne que les fichiers n'évoluant que peu fréquemment et ne peut concerner les fichiers de traces ou les fichiers spéciaux utilisés pour la communication avec les processus FastCGI. Cette solution n'améliorera donc les performances de l'application que marginalement. Notons que pour éviter la contention sur les structures de données du noyau, de nombreuses recherches ont été menées sur l'organisation des systèmes d'exploitation et sur les interfaces fournies aux applications (voir chapitre 3).

Une solution potentielle à ces deux problèmes est l'utilisation de machines virtuelles. Le principe de cette solution est simple et consiste en l'utilisation d'une machine virtuelle par cœur ou par nœud. Chaque machine virtuelle fait tourner son propre système d'exploitation, ce qui limite la contention sur les structures de données du noyau. De plus, comme chaque machine virtuelle ne tourne que sur un seul nœud et possède comme mémoire disponible la mémoire de ce nœud, il n'y a pas d'accès à des zones mémoire distantes. Cependant, les problèmes de cette solution sont également non négligeables. Premièrement, la virtualisation de la machine a un coût. Il est donc nécessaire que le coût de l'hyperviseur ne soit pas plus important que les bénéfices en matière de passage à l'échelle. Deuxièmement, la duplication des fichiers sur chacune des machines virtuelles rend la mise en place de cette solution compliquée pour les mêmes raisons que celles citées précédemment. La mutualisation des fichiers entre les machines virtuelles nécessite de réintroduire des verrous partagés entre les machines virtuelles ou un mécanisme de consensus tel que celui de Barrelfish [18].

Enfin, à plus long terme, nous souhaitons être capables d'utiliser les observations effectuées sur le passage à l'échelle du serveur Apache pour pouvoir proposer des solutions aux problèmes de performances des systèmes d'exploitation sur les architectures multi-cœurs NUMA. Nous avons durant nos expériences utilisé un ensemble de métriques, dérivées des compteurs de performances des processeurs. Ces compteurs sont actuellement uniquement utilisés par les programmeurs et les outils d'analyse de performance pour comprendre le comportement d'une application. Toutefois, ils pourraient être exploités par les noyaux de système d'exploitation pour, par exemple, détecter le partage de mémoire entre deux processus. Ainsi, il serait possible pour l'ordonnanceur de colocaliser des processus partageant des zones de mémoire sur un même nœud.

6.9 Conclusion

Dans ce chapitre, nous avons étudié le comportement sur une machine multi-cœurs du serveur Web le plus répandu actuellement. Pour cela, nous avons utilisé la version Support de SPECWeb 2005, un banc d'essai reconnu pour serveurs Web. Nous avons montré dans nos expériences que, de manière contre-intuitive, le serveur Web n'est pas capable d'exploiter au mieux les architectures multi-cœurs. Nous avons montré au moyen d'un ensemble de mesures de performance que les prin-

cipaux problèmes lors du passage à 16 cœurs sont l'augmentation du coût des verrous, ainsi que la baisse de l'efficacité des accès mémoire. Nous avons proposé, par la suite, de colocaliser les processus Apache et PHP pour améliorer l'efficacité mémoire et avons mis en évidence la nécessité de prendre en compte l'asymétrie dans l'interconnexion entre les processeurs. Grâce à nos propositions, nous avons pu améliorer les performances du serveur Web de 20 % par rapport à la version standard. Toutefois, nous déplorons toujours une baisse de 17 % par rapport à notre objectif de passage à l'échelle.

Nous avons enfin détaillé un certain nombre de solutions potentielles avec leurs avantages et leurs inconvénients. Dans des travaux futurs, nous allons essayer de chercher une solution pour limiter au maximum les accès distants ainsi que la contention sur les verrous. Dans cette optique, une des étapes sera d'évaluer le passage à l'échelle et l'impact sur les performances de la virtualisation. Une autre étape sera de limiter au maximum l'emploi de multiples instances du serveur, car celui-ci amène de nombreux problèmes (pas de partage des descripteurs de fichier, duplication des fichiers, etc.).

Conclusion & Perspectives

L'évolution des processeurs a changé radicalement de direction ces dernières années. Comme nous l'avons montré dans le chapitre 1, les processeurs commencent à devenir de plus en plus parallèles, en intégrant un nombre croissant de cœurs d'exécution. L'architecture sous-jacente exposée au programmeur devient de plus en plus complexe. Par exemple, nous sommes passés de multi-cœurs avec lesquels le temps d'accès à la mémoire est constant à des architectures NUMA dans lesquelles ce coût n'est pas uniforme en fonction du cœur qui fait l'accès et de la zone de mémoire manipulée. Nous avons montré que la tendance pour les années à venir est d'avoir de plus en plus de cœurs sur une même puce.

Cette évolution change radicalement la manière dont un programmeur doit appréhender les problèmes. Auparavant, l'évolution de la performance des logiciels était aisée puisque liée à l'augmentation de la fréquence des processeurs. Cependant, avec ces nouvelles architectures, la fréquence stagne, voire régresse, et il est nécessaire de tirer au maximum parti du parallélisme proposé pour avoir de bonnes performances. Cela passe par une refonte des modèles de programmation, comme nous l'avons montré dans le chapitre 2. Ces modèles se doivent de permettre une exploitation simple, sûre et efficace des processeurs multi-cœurs. Dans ce cadre, nous pensons que la programmation événementielle a d'importants atouts pour s'imposer dans le contexte des serveurs de données. Nous pensons notamment que la préemption maîtrisée est d'une grande aide dans le développement des applications multi-cœurs et évite des prises de verrous complexes. De même, l'utilisation des couleurs fournit un moyen simple pour le programmeur d'exprimer une exclusion mutuelle entre deux tâches. Enfin, ce modèle de programmation pourrait bénéficier à l'avenir des extensions d'envoi de messages qui seront probablement fournies par les futurs processeurs.

Un modèle de programmation est fortement couplé à un modèle d'exécution donné. Le modèle d'exécution est responsable entre autres de l'ordonnancement des tâches sur les cœurs disponibles. L'ordonnancement des tâches est une partie critique dans l'utilisation des multi-cœurs. Nous avons expliqué dans le chapitre 3 que de nombreux travaux se sont intéressés à cette dimension de l'environnement d'exécution. D'autres travaux ont été menés sur la structuration et les interactions au sein des systèmes d'exploitation ainsi que sur les mécanismes d'allocation mémoire.

Nous avons considéré dans ces travaux une classe d'applications que sont les serveurs de données. Ce type d'application est très répandu et doit faire face à une charge importante. Par conséquent, il est essentiel de maximiser l'utilisation de la machine sur laquelle elle tourne et notamment être capable d'utiliser efficacement l'ensemble des cœurs disponibles. Nos travaux comportent deux volets : l'étude et l'amélioration d'un support d'exécution pour la programmation événementielle multi-cœurs ainsi que l'étude et l'amélioration du passage à l'échelle du très répandu serveur Web Apache.

Bilan de nos travaux

Chacun des deux axes de travail a permis d'effectuer un certain nombre d'observations et de propositions concernant l'amélioration de la performance des serveurs de données sur les architectures

multi-cœurs. Dans chacun de ces contextes, plusieurs de points restent cependant encore à explorer.

Vol de tâches efficace pour les systèmes événementiels multi-cœurs

La première étape de nos travaux a été d'étudier la performance d'un environnement d'exécution événementiel sur les nouvelles architectures multi-cœurs. Le mécanisme de vol de tâches est au cœur de la plupart des environnements d'exécution tels que TBB, Cilk, ou encore le noyau Linux lui-même. La performance de ce mécanisme est donc essentielle pour obtenir de bonnes performances à 16 cœurs. Dans ces travaux, nous avons mis en évidence que le mécanisme de vol de tâches pouvait, dans certains cas, impacter négativement les performances des applications, notamment d'un serveur Web, tout en augmentant significativement la performance d'autres applications, comme un serveur de fichiers sécurisé. Nous avons donc choisi de proposer un ensemble d'améliorations de l'algorithme de vol de tâches de Libasync-smp, portant notamment sur la pertinence du choix du cœur victime ainsi que des tâches à voler, et sur l'architecture du support d'exécution. Ces optimisations ont permis de sensiblement améliorer les performances du serveur Web tout en conservant les performances du serveur de fichiers.

Les travaux effectués sur ce sujet peuvent être étendus dans des travaux futurs par plusieurs études. Premièrement, le vol de tâches a été évalué sur le serveur Web avec une charge statique constituée de petits fichiers (simulant par exemple des images). Connaître son comportement sur des charges plus variées, notamment SPECWeb 2005 avec des fichiers statiques et dynamiques, est une étape intéressante pour voir l'impact du mécanisme de vol de tâches et de nos optimisations dans ce contexte. Deuxièmement, deux des trois métriques proposées nécessitent pour le moment des informations fournies par le programmeur. Ces informations peuvent être plus ou moins complexes à déterminer en fonction de la taille de l'application et de sa sensibilité aux variations de charge. Une extension de nos travaux est donc d'alléger le travail d'un programmeur d'application en couplant les informations de profilage fournies par notre environnement d'exécution aux décisions du mécanisme de vol de tâches. Troisièmement, ces travaux ont été effectués et évalués sur une architecture à 8 cœurs avec des coûts d'accès mémoire uniformes. Il peut être très intéressant d'évaluer les modifications à apporter à Mely pour qu'il soit capable d'utiliser efficacement une architecture à 16 cœurs NUMA. Enfin, évaluer l'utilité de nos optimisations sur d'autres environnements d'exécution semble être un point intéressant. Notamment, dans certains types d'environnement d'exécution il est toujours rentable de voler, car le vol ne ralentit pas la victime. Toutefois, classer les tâches par rentabilité peut quand même améliorer l'efficacité du vol.

Étude et amélioration des performances du serveur Web Apache sur une architecture multi-cœurs NUMA

La seconde étape de nos travaux a été d'évaluer et d'améliorer le passage à l'échelle du serveur Web Apache sur une architecture multi-cœurs NUMA. L'étude de ce serveur Web est particulièrement intéressante, car il s'agit du serveur le plus déployé actuellement. Sa capacité à utiliser efficacement les multi-cœurs est donc essentielle pour les années à venir. La configuration du serveur Web Apache que nous avons mis en place utilise un thread par connexion. Comme les connexions sont indépendantes, ce serveur doit être en mesure d'utiliser efficacement les processeurs multi-cœurs. Toutefois, nous avons mis en évidence dans ces travaux que le passage à l'échelle d'Apache n'est pas idéal. Nous avons donc proposé deux améliorations complémentaires visant à augmenter l'efficacité des accès mémoire et à équilibrer la charge entre les nœuds, en prenant en compte la topologie d'interconnexion entre les processeurs. Ces propositions ont permis d'améliorer les performances du serveur Web sans toutefois atteindre le passage à l'échelle que nous avons défini comme raisonnable.

La suite logique de ces travaux est de trouver une solution pour améliorer encore le passage à l'échelle du serveur. Dans cette optique, il sera intéressant de comparer la solution proposée précédemment aux performances obtenues avec la virtualisation. Dans ce cas, la machine est considérée comme N machines, chacune ayant un ou plusieurs cœurs à sa disposition. L'avantage de cette solution est qu'elle ne devrait pas souffrir de problèmes de contention logiciels (comme sur la fonction `_d__lookup` par exemple), sauf si l'hyperviseur est lui-même sujet à ces problèmes. Elle permet également de réduire fortement les accès mémoire distants. Cependant, elle ne permet pas de résoudre des goulots d'étranglement matériels comme ceux liés au mécanisme de cohérence de cache. Il sera intéressant d'évaluer le surcoût de cette solution. Une autre possibilité est de regarder comment adapter le serveur Web aux nouveaux systèmes d'exploitation multi-cœurs. Cependant, ces systèmes ne sont généralement pas très mûrs et n'implantent pas toutes les fonctionnalités d'un vrai système d'exploitation. Par exemple, le système d'exploitation multi-cœurs Corey [100] intègre la pile réseau LWIP, conçue à l'origine pour les architectures embarquées. Des tests préliminaires sur ce système d'exploitation ont montré que cette pile réseau (ou tout du moins sa mise en œuvre dans Corey) a des performances bien inférieures à celles de la pile réseau de Linux et donc le réseau devient le goulot d'étranglement.

Ces deux axes de travail ont pour objectif de montrer que la performance des serveurs Web sur les architectures multi-cœurs peut être significativement augmentée, soit par l'amélioration des mécanismes d'ordonnancement de tâches, soit par l'optimisation de l'interaction entre les différents tiers au sein d'une même machine. Une des particularités de nos expériences est d'expliquer l'impact de nos optimisations sur la performance des applications en utilisant les compteurs fournis par les processeurs récents. L'utilisation de ces compteurs est un procédé récent et encore peu employé. Notamment, nous avons considéré dans nos études les compteurs de fautes de cache L2 et L3, les compteurs de cycles et d'instructions, les compteurs d'accès mémoire locaux et distants, les compteurs relatifs aux liens HyperTransport et les compteurs de latence d'accès mémoire. Montrer comment utiliser ces compteurs pour confirmer nos hypothèses est un des points intéressants de ces travaux.

Ouverture vers d'autres sujets

Les travaux que nous avons menés sur la performance des serveurs de données sur les architectures multi-cœurs nous ont amenés à considérer des pistes d'études futures pouvant être conduites sur des sujets connexes.

Premièrement, nous nous sommes aperçus que, dans Libasync-smp, le mécanisme de ramasse-miettes dégrade fortement les performances des différentes applications. Nous avons donc supprimé ce mécanisme lors de nos expériences, mais il peut être intéressant d'évaluer la performance des différents types de ramasse-miettes avec des processeurs multi-cœurs.

Une seconde piste de travaux futurs est d'étudier la performance du serveur Apache avec d'autres types de charge, notamment SPECWeb 2005 *Banking* et *E-commerce*. Ces deux types de charges peuvent mener à des conclusions radicalement différentes, car ils font moins appel aux primitives de bas niveau telles que `sendfile` et plus à la génération dynamique de pages PHP ainsi qu'au chiffrement des données via SSL. Par conséquent, le ratio entre le temps passé à faire des E/S réseau par rapport au temps passé à faire des calculs est très différent.

Une troisième piste potentielle est de déterminer si un modèle de programmation est plus adapté qu'un autre pour la conception de serveurs de données parallèles (en terme de performance et de simplicité). Des travaux précédents [76] ont montré que, dans le contexte des processeurs mono-cœurs, la programmation événementielle permet d'avoir de meilleures performances que la programmation

par threads. Aucune étude de ce type n'a été menée à ce jour dans le contexte multi-cœurs et il peut être intéressant de fournir une réponse à cette question.

Enfin, une quatrième piste possible est d'étudier les mécanismes de collecte de traces à des fins de recherche de bogues ou pour surveiller l'état de l'application (utilisation des processeurs et des cartes réseau, requêtes effectuées sur le serveur pour un serveur Web, etc.). En effet, nous nous sommes aperçus que la collecte de traces d'exécution peut avoir un impact non négligeable sur la performance des applications. La solution généralement choisie, et que nous avons mise en place, est de faire de l'échantillonnage ou de se contenter de moyennes sur l'exécution. Cette solution permet de réduire les coûts, mais réduit également la précision des observations. En améliorant l'efficacité des mécanismes de collecte de traces, il serait possible de réduire leur impact sur les performances et donc d'augmenter leur précision (augmentation de la fréquence d'échantillonage ou idéalement observation en continu).

Bibliographie

[1] *eaccelerator*, `http://eaccelerator.net/`.

[2] *Openmp*, `http://www.cisl.ucar.edu/css/software/multio/`.

[3] *Oprofile - a system profiler for linux*, `oprofile.sourceforge.net`.

[4] *The multio benchmark*, 2004, `http://www.cisl.ucar.edu/css/software/multio/`.

[5] *The µserver project*, 2007, http ://userver.uwaterloo.ca.

[6] *The Apache HTTP server project*, 2007, http ://httpd.apache.org.

[7] *Netcraft*, May 2010, `http://news.netcraft.com/`.

[8] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young, *Mach : A new kernel foundation for unix development*, USENIX Summer, 1986, pp. 93–113.

[9] Acme Labs, *thttpd : Tiny/turbo/throttling http server*, `http://www.acme.com/software/thttpd/`.

[10] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur, *Cooperative Task Management Without Manual Stack Management*, Proceedings of the the 2002 USENIX Annual Technical Conference (Monterey, CA, USA), USENIX Association, June 2002.

[11] Gul Agha, *Actors : a model of concurrent computation in distributed systems*, MIT Press, Cambridge, MA, USA, 1986.

[12] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, *Scheduler activations : effective kernel support for the user-level management of parallelism*, SOSP '91 : Proceedings of the thirteenth ACM symposium on Operating systems principles, ACM, 1991, pp. 95–109.

[13] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis, *Memory management in k42*, 2002.

[14] The Internet Traffic Archive, `http://ita.ee.lbl.gov/html/traces.html`.

[15] Joe Armstrong, *Erlang*, Commun. ACM **53** (2010), no. 9, 68–75.

[16] M. Azimi, N. Cherukuri, D.N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. Vaidya, *Integration Challenges and Tradeoffs for Tera-scale Architectures*, Intel Technology Journal (2007).

[17] Gaurav Banga and Peter Druschel, *Measuring the Capacity of a Web Server*, Proceedings of USITS'97 (Monterey, CA, USA), USENIX Association, 1997.

[18] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schuepbach, and Akhilesh Singhania, *The multikernel : a new OS architecture for scalable multicore systems*, SOSP '09 : Proceedings of the 22nd ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, October 2009.

[19] Andrew Baumann, Simon Peter, Adrian Schuepbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs, *Your computer is already a distributed system. Why isn't your OS ?*, Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII) (Monte Verita, Switzerland), May 2009.

[20] Emery D. Berger, Kathryn S. Mckinley, Robert D. Blumofe, and Paul R. Wilson, *Hoard : A scalable memory allocator for multithreaded applications*, In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), 2000.

[21] Sapan Bhatia, Charles Consel, and Julia L. Lawall, *Memory-Manager/Scheduler Co-Design : Optimizing Event-Driven Servers to Improve Cache Behavior*, Proceedings of ISMM'06 (Ottawa, Ontario, Canada), ACM Press, June 2006.

[22] Robert D. Blumofe, *Executing multithreaded programs efficiently*, Ph.D. thesis, Cambridge, MA, USA, 1995.

[23] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, *Cilk : An Efficient Multithreaded Runtime System*, J. Parallel Distrib. Comput. **37** (1996), no. 1, 55–69.

[24] Robert D. Blumofe and Charles E. Leiserson, *Scheduling multithreaded computations by work stealing*, J. ACM **46** (1999), no. 5, 720–748.

[25] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek, *Reinventing scheduling for multicore systems*, Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII) (Monte Verità, Switzerland), May 2009.

[26] Apple Inc. Technical Breif, *Grand central dispatch :a better way to do multicore*, 2009.

[27] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner, *Flux : A Language for Programming High-Performance Servers*, USENIX Annual Technical Conference (Boston, MA, USA), USENIX Association, May 2006.

[28] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee, *Software transactional memory : Why is it only a research toy ?*, ACM Queue **6** (2008), no. 5, 46–58.

[29] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin, *Predicting inter-thread cache contention on a chip multi-processor architecture*, High-Performance Computer Architecture, International Symposium on **0** (2005), 340–351.

[30] David Chase and Yossi Lev, *Dynamic circular work-stealing deque*, Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05) (Las Vegas, Nevada, USA), ACM, 2005.

[31] Benjie Chen and Robert Morris, *Flexible Control of Parallelism in a Multiprocessor PC Router*, Proceedings of the 2001 USENIX Annual Technical Conference (Boston, MA, USA), USENIX Association, June 2001.

[32] Intel Corporation, `http://techresearch.intel.com/articles/Tera-Scale/1421.htm`.

[33] Matthew Curtis-Maury and Tanping Wang, *Integrating multiple forms of multithreaded execution on multi-smt systems : A study with scientific applications*, QEST '05 : Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, IEEE Computer Society, 2005, p. 199.

[34] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, *Wide-area cooperative storage with cfs*, Proceedings of the eighteenth ACM symposium on Operating systems principles (Banff, Alberta, Canada), ACM Press, 2001.

[35] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris, *Event-Driven Programming for Robust Software*, Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion, France), ACM Press, September 2002.

[36] David Wentzlaff and Charles Gruenwald III and Nathan Beckmann and Kevin Modzelewski and Adam Belay and Lamia Youseff and Jason Miller and and Anant Agarwal, *An Operating System for Multicore and Clouds : Mechanisms and Implementation*, Tech. Report MIT-CSAIL-TR-2010-003, MIT, 2010.

[37] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce : simplified data processing on large clusters*, Commun. ACM **51** (2008), no. 1, 107–113.

[38] Mihai Dobrescu, Norbert Egi, Katerina J. Argyraki, Byung-Gon Chun, Kevin R. Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy, *Routebricks : exploiting parallelism to scale software routers*, SOSP, 2009, pp. 15–28.

[39] D. R. Engler and M. F. Kaashoek, *Exterminate all operating system abstractions*, HOTOS '95 : Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), IEEE Computer Society, 1995, p. 78.

[40] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., *Exokernel : an operating system architecture for application-level resource management*, SOSP '95 : Proceedings of the fifteenth ACM symposium on Operating systems principles, ACM, 1995, pp. 251–266.

[41] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi, *Language support for fast and reliable message-based communication in singularity os*, SIGOPS Oper. Syst. Rev. **40** (2006), no. 4, 177–190.

[42] Michael J. Freedman, Eric Freudenthal, and David Mazières, *Democratizing Content Publication with Coral*, Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI'04) (San Francisco, CA, USA), USENIX Association, 2004.

[43] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, *The implementation of the cilk-5 multithreaded language*, SIGPLAN Not. **33** (1998), no. 5, 212–223.

[44] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm, *Tornado : maximizing locality and concurrency in a shared memory multiprocessor operating system*, OSDI '99 : Proceedings of the third symposium on Operating systems design and implementation, USENIX Association, 1999, pp. 87–100.

[45] Daniel F. García and Javier García, *Tpc-w e-commerce benchmark evaluation*, Computer **36** (2003), no. 2, 42–48.

[46] Fabien Gaud, Sylvain Geneves, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma, *Efficient workstealing for multicore event-driven systems*, International Conference on Distributed Computing Systems, June 2010, pp. 516–525.

[47] Fabien Gaud, Sylvain Geneves, and Fabien Mottet, *Vol de tâches efficace pour systèmes Événementiels multi-cœurs*, Conférence Française en Systèmes d'Exploitation (CFSE'7), September 2009.

[48] Sanjay Ghemawat and Paul Menage, *Tcmalloc : Thread-caching malloc*, 2008, `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[49] Sebastien Godard, *Sysstat utilities*, `http://sebastien.godard.pagesperso-orange.fr/index.html`.

[50] Andreas Gustafsson, *Threads without the pain*, ACM Queue **3** (2005), no. 9, 34–41.

[51] Ashif Harji, *Performance comparison of uniprocessor and multiprocessor web server architectures*, Ph.D. thesis, January 2010.

[52] Maurice Herlihy and J. Eliot B. Moss, *Transactional memory : architectural support for lock-free data structures*, SIGARCH Comput. Archit. News **21** (1993), no. 2, 289–300.

[53] John Jannotti and Kiran Pamnany, *Safe at Any Speed : Fast, Safe Parallelism in Servers*, Proceedings of the 2nd USENIX Workshop on Hot Topics in System Dependability (HotDep'06) (Seattle, Washington, USA), USENIX Association, November 2006.

[54] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi, *Analysis and approximation of optimal co-scheduling on chip multiprocessors*, PACT '08 : Proceedings of the 17th international conference on Parallel architectures and compilation techniques (New York, NY, USA), ACM, 2008, pp. 220–229.

[55] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan, *The openmp implementation of nas parallel benchmarks and its performance*, Tech. report, 1999.

[56] Dan Kegel, *The c10k problem*, 2006, `http://www.kegel.com/c10k.html`.

[57] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn, *Using os observations to improve performance in multicore systems*, IEEE Micro **28** (2008), no. 3, 54–66.

[58] Orran Krieger, Marc A. Auslander, Bryan S. Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria A. Butrico, Mark F. Mergen, Amos Waterland, and Volkmar Uhlig, *K42 : building a complete operating system*, Proceedings of the 2006 EuroSys Conference, ACM Press, 2006.

[59] Maxwell Krohn, *Building Secure High-Performance Web Services with OKWS*, Proceedings of the 2004 USENIX Annual Conference (Boston, MA, USA), USENIX Association, June 2004.

[60] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek, *Events Can Make Sense*, Proceedings of the 2007 USENIX Annual Technical Conference (Santa Clara, CA, USA), USENIX Association, June 2007.

[61] Alexey Kukanov and Michael J. Voss, *The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks*, Intel Technology Journal **11** (2007).

[62] James Larus and Christos Kozyrakis, *Transactional memory*, Commununication of the ACM **51** (2008), no. 7, 80–88.

[63] Hugh C. Lauer and Roger M. Needham, *On the duality of operating system structures*, Operating Systems Review **13** (1979), no. 2, 3–19.

[64] Edward A. Lee, *The Problem with Threads*, IEEE Computer **39** (2006), no. 5.

[65] Peng Li and Steve Zdancewic, *Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives*, PLDI '07 : Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, ACM, 2007, pp. 189–199.

[66] J. Liedtke, *On micro-kernel construction*, SOSP '95 : Proceedings of the fifteenth ACM symposium on Operating systems principles, ACM, 1995, pp. 237–250.

[67] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton, *Hyper-threading technology architecture and microarchitecture*, Intel Technology Journal **6** (2002), no. 1, 4–15.

[68] David Mazières, *A Toolkit for User-Level File Systems*, Proceedings of the 2001 USENIX Annual Technical Conference (Boston, MA, USA), USENIX Association, 2001.

[69] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel, *Separating Key Management From File System Security*, Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99) (Kiawah Island, South Carolina, USA), ACM Press, December 1999.

[70] Maged M. Michael, *Scalable lock-free dynamic memory allocation*, PLDI '04 : Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, 2004, pp. 35–46.

[71] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek, *The click modular router*, SOSP '99 : Proceedings of the seventeenth ACM symposium on Operating systems principles, ACM, 1999, pp. 217–231.

[72] David Mosberger and Tai Jin, *Httperf, a tool for measuring web server performance*, First Workshop on Internet Server Performance **26** (1998), 31–37.

[73] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt, *Helios : heterogeneous multiprocessing with satellite kernels*, SOSP '09 : Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (New York, NY, USA), ACM, 2009, pp. 221–234.

[74] John K. Ousterhout, *Why threads are a bad idea (for most purposes)*, Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[75] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, *Flash : An efficient and portable Web server*, Proceedings of the 1999 USENIX Annual Technical Conference (Monterey, California, USA), June 1999.

[76] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla, *Comparing the Performance of Web Server Architectures*, Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys'07) (Lisbon, Portugal), ACM Press, June 2007.

[77] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and David Hitz, *NFS Version 3 Design and Implementation*, Proceedings of the USENIX 1994 Summer Conference (Boston, MA, USA), USENIX Association, June 1994.

[78] Niel Provos, *libevent — An Event Notification Library*, 2008, http ://monkey.org/ provos/libevent/.

[79] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting, *Profile-directed optimization of event-based programs*, SIGPLAN Not. **37** (2002), no. 5, 106–116.

[80] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, *Evaluating mapreduce for multi-core and multiprocessor systems*, HPCA '07 : Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (Washington, DC, USA), IEEE Computer Society, 2007, pp. 13–24.

[81] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Petersen, Vijay Menon, Brian Murphy, Tatiana Shpeisman, Eric Sprangle, Anwar Rohillah, Doug Carmean, and Jesse Fang, *Enabling Scalability and Performance in a Large Scale CMP Environment*, Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys'07) (Lisbon, Portugal), ACM Press, June 2007.

[82] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter, *Open Versus Closed : a Cautionary Tale*, Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI'06) (San Jose, CA), USENIX Association, May 2006.

[83] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan, *Larrabee : a many-core x86 architecture for visual computing*, ACM Trans. Graph. **27** (2008), no. 3, 1–15.

[84] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar, *Hass : a scheduler for heterogeneous multicore systems*, SIGOPS Operating Systems Review **43** (2009), no. 2, 66–75.

[85] SPECWeb 2005 Result, `http://www.spec.org/web2005/results/res2010q1/web2005-20100310-00146.html`.

[86] Standard Performance Evaluation Corporation (SPEC), *Specweb2005*, `http://www.spec.org/web2005/`.

[87] _____, *Specweb99*, `http://www.spec.org/web99/`.

[88] Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen, *Fast switching of threads between cores*, SIGOPS Oper. Syst. Rev. **43** (2009), no. 2, 35–45.

[89] Herb Sutter and James Larus, *Software and the concurrency revolution*, ACM Queue **3** (2005), no. 7, 54–62.

[90] Toyotaro Suzumura, Michiaki Tatsubori, Scott Trent, Akihiko Tozawa, and Tamiya Onodera, *Highly scalable web applications with zero-copy data transfer*, 18th International World Wide Web Conference, April 2009, pp. 921–921.

[91] David Tam, Reza Azimi, and Michael Stumm, *Thread clustering : sharing-aware scheduling on smp-cmp-smt multiprocessors*, EuroSys '07 : Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys European Conference on Computer Systems 2007 (EuroSys 2007), ACM, 2007.

[92] The Linux Foundation, *Bonding multiple devices*, 2009, `http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding`.

[93] J. Torrellas, H. S. Lam, and J. L. Hennessy, *False Sharing and Spatial Locality in Multiprocessor Caches*, IEEE Transactions on Computers **43** (1994), no. 6, 651–663.

[94] Theo Ungerer, Borut Robič, and Jurij Šilc, *A survey of processors with explicit multithreading*, ACM Computing Surveys **35** (2003), no. 1, 29–63.

[95] Bryan Veal and Annie Foong, *Performance Scalability of a Multi-Core Web Server*, Proceedings of ANCS '07 (Orlando, FL, USA), ACM Press, December 2007.

[96] Robert von Behren, Jeremy Condit, and Eric A. Brewer, *Why events are a bad idea (for high-concurrency servers)*, Proceedings of HOTOS'03 (Lihue, Hawaii), May 2003.

[97] Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer, *Capriccio : Scalable threads for internet services*, Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03) (Bolton Landing, New York, USA), ACM Press, October 2003.

[98] Matt Welsh, David Culler, and Eric Brewer, *SEDA : An architecture for well-conditioned scalable internet services*, Proceedings of SOSP 2001 (Banff Alberta Canada), ACM Press, October 2001.

[99] David Wentzlaff and Anant Agarwal, *Factored operating systems (fos) : the case for a scalable operating system for multicores*, SIGOPS Oper. Syst. Rev. (2009).

[100] Silas B. Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang, *Corey : An Operating System for Many Cores*, Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08) (San Diego, CA, USA), USENIX Association, December 2008.

[101] Yuejian Xie and Gabriel H Loh, *Dynamic Classification of Program Memory Behaviors in CMPs*, 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI), June 2008.

[102] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and M. Frans Kaashoek, *Multiprocessor Support for Event-Driven Programs*, Proceedings of the 2003 USENIX Annual Technical Conference (San Antonio, TX, USA), USENIX Association, June 2003.

[103] Zeus Technology, *Zeus Web Server*, `http://www.zeus.com/products/zws/`.

[104] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova, *Addressing shared resource contention in multicore processors via scheduling*, ASPLOS '10 : Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (New York, NY, USA), ACM, 2010, pp. 129–142.

Résumé

Cette thèse s'intéresse à la performance des serveurs de données sur les architectures multi-cœurs. Nous avons choisi d'étudier ce problème sous deux aspects différents. Premièrement, nous étudions un support d'exécution événementiel. Nous montrons notamment que le mécanisme de vol de tâches, utilisé pour équilibrer la charge entre les cœurs, peut pénaliser la performance d'un serveur Web. Nous proposons donc diverses optimisations pour améliorer les performances de ce mécanisme sur les processeurs multi-cœurs. Deuxièmement, nous étudions la performance du serveur Web Apache, exploitant à la fois un ensemble de threads et de processus, sur une architecture multi-cœurs NUMA. Nous montrons notamment que, sous une charge réaliste, ce serveur Web ne passe pas idéalement à l'échelle. Grâce à une analyse détaillée des coûts, nous déterminons les raisons de ce manque de passage à l'échelle et présentons un ensemble de propositions visant à améliorer la performance de ce serveur sur une architecture NUMA.

Mots-clés. Architectures multi-cœurs, Architectures NUMA, Performance des serveurs de données, Programmation événementielle, Vol de tâches, Serveurs Web, Serveur de fichiers, Apache, Analyse de performance

Abstract

This thesis focuses on data servers performance on multicore architectures. We study two different aspects of this problematic. First, we benchmark an event-driven multicore runtime. We especially show that the workstealing mechanism used for load balancing may sometimes degrade the performance of data servers. Consequently, we introduce a new runtime and new heuristics to improve the workstealing behavior. Second, we study the performance of the Apache Web server, which uses both threads and processes, on a NUMA architecture. We show that this Web server does not perfectly scale under a realistic workload. Thanks to a detailed analysis of costs using both hardware and software profiling, we determine the reasons of this lack of scalability and we present different propositions to improve the web server performance on NUMA architectures.

Keywords. Multicore architectures, NUMA architectures, Data server performance, Event-driven programming, Workstealing, Web servers, File servers, Apache, Performance analysis