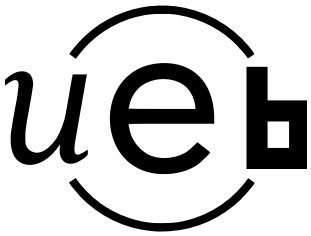
Exploiting heterogeneous manycores on sequential code

Bharath Narasimha Swamy

N^o d'ordre : ANNÉE 2015



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de
DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention : Informatique
École doctorale Matisse

présentée par

Bharath NARASIMHA SWAMY

préparée à l'unité de recherche INRIA – Bretagne Atlantique
Institut National de Recherche en Informatique et Automatique
Composante Universitaire (ISTIC)

Exploiting heterogeneous manycores on sequential code

Thèse soutenue à Rennes
le 05/03/2015

devant le jury composé de :

Pr Daniel Etiemble
Professeur, Université Paris 11 / Rapporteur

Pr Smail Niar
Professeur, Université de Valenciennes / Rapporteur

Dr Alain Ketterlin
Maître de conférences, Université de Strasbourg / Examinateur

Dr Erven Rohou
Directeur de recherche, INRIA Rennes / Examinateur

Pr Olivier Sentieys
Professeur, Université de Rennes 1 / Examinateur

Dr André Seznec
Directeur de recherche, INRIA Rennes / Directeur de thèse

Acknowledgements

I wish to thank Andre, firstly for the opportunity to work in the ALF team for preparing the thesis, and for his guidance and help during the course of work.

I thank Alain who helped in many ways in course of the thesis work.

I thank Pierre for making available his simulator, which is used for experiments in this thesis after extensions.

I thank Erven and Surya for the opportunity to collaborate, summaries of these studies are included in the thesis.

I thank Emmanuel for his help in writing an abstract of the thesis in French.

I thank my colleagues in the ALF team for their help and support.

I thank the members of the jury for the opportunity to defend the thesis.

Contents

Résumé en français

Dans la dernière décennie, les serveurs et les ordinateurs ont adopté les processeurs multi-coeurs (Chip Multi Processors , CMP) comme modèle d'architecture prédominante, et, plus récemment, même les devices mobiles ont fait le passage à multi-cœurs. En outre, comme la taille des transistors continuent de diminuer, les prévisions de l'industrie sont prometteurs quant à l'intégration de centaines de cœurs sur une seule puce dans un proche avenir.

Cependant, comme le nombre de cœurs augmente, les nouvelles 'application exigent une plus grande hétérogénéité des profils de puissance/performance, et des tailles des coeurs pour atteindre un modèle d'architecture optimisé. Dans un environnement hétérogène avec beaucoup de cœurs (Heterogeneous Many Core, HMC), quelques cœurs grands superscalaires, sont optimisés pour la latence d'exécution tandis que les petits coeurs sont optimisés pour l'exécution de code parallèle tout en étant petits et faiblement consommateurs d'énergie. Ainsi les systèmes HMC peuvent atteindre une efficacité énergétique plus élevé et permettent l'exécution de code à la fois séquentiel et parallèle en comparaison avec les systèmes multi-cœurs symétriques.

Pour être en mesure d'exploiter pleinement les nombreux cœurs disponibles dans les systèmes HMC, l'application doit se prêter à la parallélisation, soit à la décomposition du calcul en plusieurs threads permettant une exécution simultanée sur plusieurs coeurs. Cependant, dans un avenir prévisible, la plus grande partie du code des applications restera séquentiel, et donc difficile à paralléliser ne permettant pas de bénéficier de l'augmentation du nombre de cœurs. Par conséquent, l'amélioration de la performance séquentielle sur les systèmes HMC demeure une priorité. En plus de renforcer traditionnellement le grand coeur , il y a un regain d'intérêt dans l'utilisation des cœurs supplémentaires pour accélérer les performances des différents threads.

Nous proposons que les nombreux petits coeurs dans un système HMC soient utilisés comme coeurs auxilliaires de faible puissance pour accélérer les threads

séquentiels qui s'exécutent sur le grand coeur. Pour adapter les mécanismes de threads auxilliaires au système HMC , nous devons considérer les points suivants.

1. Le surcoût lié à l'exécution des threads auxilliaires pour des actions telles que thread-spawn et la synchronisation entrainent des surcoûts en raison des latences générées par la communication entre les processeurs via les caches partagés, et également des surcoûts liés au système d'exploitation.
2. En raison de la disparité des performances entre le grand et les petits coeurs, on ne sait pas encore si les petits coeurs sont adaptés pour exécuter des threads auxillaires pour faire du prefetching pour un coeur plus puissant.
3. Le prefetching par les threads auxilliaires est limité au dernier niveau de cache partagé (L3). Le thread principal entraîne une latence supplémentaire pour accéder aux lignes de cache préalablement chargées même lorsque le prefetching est fait à temps.

Dans cette thèse, nous nous concentrons sur les threads auxilliaires pour les petits /simples coeurs dans les systèmes HMC, pour améliorer la performance des codes séquentiels pour des programmes s'exécutant sur le cœur principal et ayant une utilisation intensive de la mémoire. Tout d'abord, nous développons un cadre matériel / logiciel pour faire tourner efficacement des threads auxilliaires. Nous ajoutons de nouvelles instructions en mode utilisateur comme interface (jouant le rôle d'une sorte de un co-processeur) pour les cœurs auxilliaires sur les systèmes HMC. Ce cadre permet au cœur principal de pouvoir lancer et contrôler directement l'exécution des threads auxillaires, et de transférer efficacement le contexte des applications nécessaire à l'exécution des threads auxilliaires. Ensuite, en utilisant la simulation basée sur des traces d'exécution, nous évaluons la pertinence des petits coeurs à exécuter les threads auxiliaires pour un thread séquentiel fonctionnant sur le coeur principal, qui est plus puissant. Nous constatons que sur un ensemble de programmes ayant une utilisation intensive de la mémoire, les threads auxilliaires s'exécutant sur des cœurs relativement petits, peuvent apporter une accélération significative par rapport à du prefetching hardware simple, et les petits coeurs fournir un bon compromis par rapport à l'utilisation d'un seul cœur puissant pour exécuter le thread auxilliaires. Enfin, un résumé des autres contributions, fait comme un deuxième auteur est inclus. Les études sont (1) la compréhension de l'impact des prédicteurs de branchement faisant partie de l'état de l'art sur la performance des interpréteurs et (2) une estimation de la performance des programmes multi-threads sur des architectures massivement parallèles.

Un cadre matériel / logiciel pour le thread auxilliaires dans les systèmes HMC

Dans cette thèse, nous présentons "core-tethering", un cadre / matériel et logiciel qui étroitement couples petits et grands processeurs dans un système HMC. Les éléments clés du cadre sont un ensemble de tampons architecturaux qui contiennent des paramètres nécessaires à l'exécution d'aide et de nouvelles instructions pour simplifier l'interaction entre les principaux et auxiliaires cœurs. Ensemble, ils fournissent une interface comme un co-processeur pour petits coeurs dans le système HMC et réduire les coûts de latence d'y accéder pour l'exécution d'aide en utilisant (1) un mécanisme basé sur le matériel pour initier threads auxiliaires et (2) un mécanisme de communication directe entre les cœurs qui contourne la hiérarchie de mémoire (illustré sur la figure Figure 1).

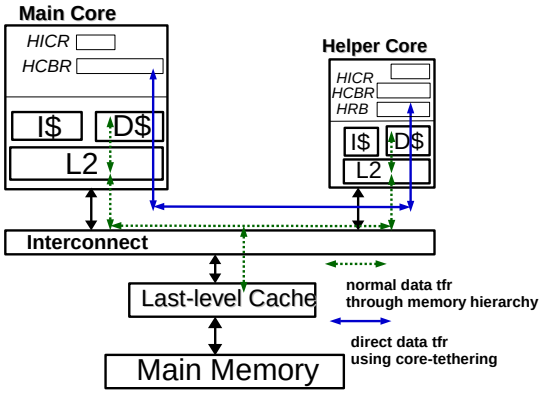


Figure 1: Illustration de la communication directe entre les cœurs.

Helper Control Block Register (HCBR) (HBR), une architecture de type tampon, détient l'action du thread auxillaire et le contexte de l'application nécessaires pour exécuter les actions de l'auxiliaire. *Helper Iteration Count Register* (HICR) est utilisé pour suivre les progrès dans le thread principal et le thread auxilliaire, et est utilisé lors de la synchronisation du thread auxiliaire.

Trois nouvelles instructions du mode utilisateur sont ajoutés (1) ISHCB - initier écriture du bloc de contrôle de auxiliaire (2) WRHCBR - écrire le registre pour le bloc de contrôle de auxiliaire (3) RDHCBR - lire le registre pour le bloc de contrôle de auxiliaire. Quand le cœur principal exécute l'instruction de ISHCB, contenu de son HCBR sont transférés à la base auxiliaire captif (petite) en utilisant le réseau d'interconnexion, et ont demandé des mesures d'auxiliaire

est lancé.

Helper Request Buffer (HRB), situé sur le petit coeur, détient demandes de ISHCB envoyés par le coeur principale sur l'interconnexion. Lorsque l'action des auxiliaires est SPAWN, il redirige le PC de la petite coeur de la cible-pc dans la demande de ISHCB et commence l'exécution de la fonction auxiliaire. Lorsque l'action de auxiliaire est KILL, l'exécution d'une fonction auxiliaire exécutant sur le petit coeur est terminée, le petit coeur peut maintenant accepter une autre demande de SPAWN. Lorsque l'action de auxiliaire est SYNC, seul le HCBR du petit cœur est rempli.

Évaluation des threads auxiliaires sur le système HMC

Notre cadre d'évaluation est basé sur des traces d'exécution et comprend trois étapes - analyse statique et construction auxiliaire, de traçage et de génération de liste d'instructions, et de simulation provenant des traces. Un lecteur de trace utilise les informations de l'étape de la construction auxiliaire pour générer des listes d'instructions pour les principaux et auxiliaires coeurs pour l'exécuter sur un simulateur.

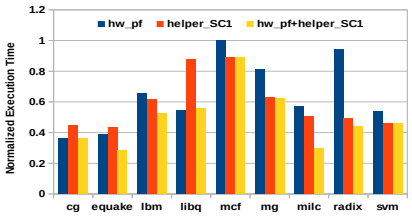
Nous modélisons grands et petits coeurs pour qu'ils exécutent la même ISA. Les paramètres des grandes coeurs sont semblables à processeurs superscalaires (out-of-order) de haute performance (un grand "reorder buffer(ROB)", prédiction agressive de branches, un prédicteur de dépendance et un prefetcher matériel pour le cache L1). Le petit coeur est un processeur avec une faible issue, avec des caches très petits, de petites files d'attente et, une très petite table de prédiction de branchement et aucunes fonctionnalités qui améliorent les performances. Le LargeCore-1 (LC1) modelise un processeur 6-issue avec la taille ROB 128, et le SmallCore-2 (SC2) modélise un processeur 2-issue avec 64 entrées ROB. Le LargeCore-2 (LC2) modélise un processeur beaucoup plus grande avec deux fois la taille du ROB de LC1, tandis que SmallCore-1 (SC1) modèles un processeur encore plus petit avec moitié de la taille de ROB et des ressources d'exécution de SC2.

Nous avons utilisé les programmes des suites standards de référence (SPEC2000, spec2006, NAS, minebench, splash2x), et les applications sélectionnées subissent plus de 10 MPKI (misses par kilo-instruction). Chaque application est simulée pour 500 millions d'instructions après l'avance rapide au delà de la phase d'initialisation, 50B (milliards) des instructions pour SPEC2000 et spec2006, 25B pour NAS, 40B et 10B pour radix et svm-rfe. Différents ensembles d'entrées ont

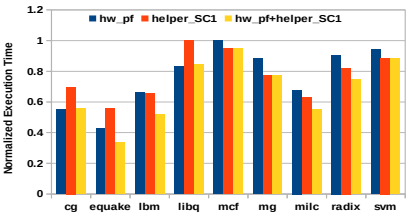
été utilisés pour le profil et pour la simulation, excepté pour deux applications, radix et svm.

Performance of Helper Prefetching on Small Cores

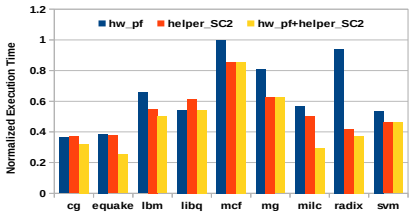
Pour les grands cœurs LC1 et LC2, Figure 2 compare le temps d'exécution (500M instructions) pour la configuration suivante: *hw_pf* - prefetching utilisant un prefetcher matériel pour le cache L1, *helper_X* - prefetching avec l'aide auxiliaire sur un des petits coeurs SC1 ou SC2, ou un grand coeur égale au coeur auxiliaire, et *hw_pf+helper_X* - combinés prefetching utilisant à la fois prefetcher matériel et un cœur auxiliaire. Temps d'exécution pour chaque configuration est normalisée à la configuration de base sans prefetching auxilliaire et prefetching matériel.



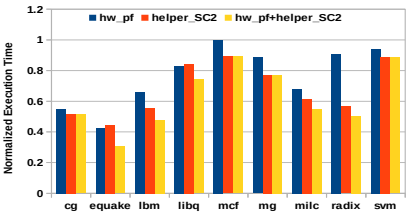
(i) coeur auxiliaire ***SC1***



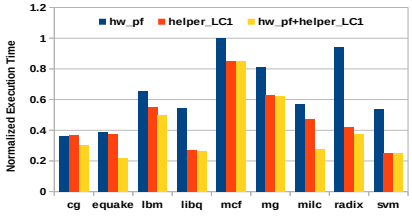
(i) coeur auxiliaire ***SC1***



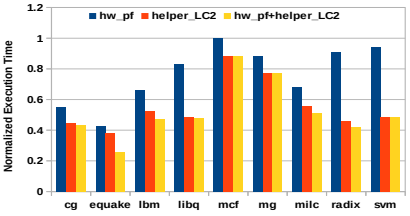
(ii) coeur auxiliaire ***SC2***



(ii) coeur auxiliaire ***SC2***



(iii) coeur auxiliaire ***LC1***



(iii) coeur auxiliaire ***LC2***

(a) auxiliaires pour le coeur principale ***LC1*** (b) auxiliaires pour le coeur principale ***LC2***

Figure 2: Threads auxiliaires dans le système HMC

Prefetching auxiliaire est capable de prefetching pour des applications avec des patterns d'accès séquentiels et irréguliers. Dans l'ensemble, pour la configuration LC1, le prefetching auxiliaire sur de petits coeurs fournit une performance additionnelle moyenne par rapport à hw_pf seule de 11,3% avec SC1, et 23,2% avec SC2. Pour le plus grand coeur LC2 (plus grande fenêtre d'instruction), nous constatons que le thread auxiliaire sur le petit coeur SC2 surpasse le prefetching matériel seul. Nous voyons aussi un effet synergique pour des patterns d'accès séquentiel où le thread auxiliaire est en avance et charge des lignes de cache dans le cache L3 partagé, et le prefetcher matériel récupère les lignes de cache dans le cache L1 et cache complètement la latence d'accès mémoire.

Nous comparons également l'efficacité de prefetching auxiliaire sur de petits coeurs avec une configuration qui utilise un grand coeur égal pour exécuter des threads auxiliaires. Dans nos benchmarks de référence, à l'exception de deux applications, libquantum et svm, SC2 offre un bon compromis comparé à un grand coeur égal.

Autres Contributions

Cette section présente un résumé des autres contributions, comme un deuxième auteur. Les études sont (1) la compréhension de l'impact des prédicteurs de branchement faisant partie de l'état de l'art sur la performance des interpréteurs et (2) une estimation de la performance des programmes multi-threads sur des architectures massivement parallèles.

Prédiction de branchement et de la performance des Interpreteurs

Malgré la popularité des languages qui sont JIT compilé, tels que Java, les interpreteurs restent largement répandue pour la mise en œuvre des languages de programmation tels que R, Python, Matlab. Les interpreteurs sont beaucoup plus faciles à développer, maintenir, et pour le portage d'applications sur de nouvelles architectures, mais cela se fait au détriment de la performance. Même avec une mise en œuvre efficace de interpreteurs, il y a un ralentissement de 10X dans la vitesse d'exécution par rapport au code natif produit dans un compilateur optimisé.

Le surcoût de la performance des interpreteurs est en raison de l'exécution de la boucle de dispatch qui lit le bytecode, décode, et effectue des mesures appropriées sur la base du bytecode, où la sélection de l'action est généralement mis en œuvre dans une instruction switch. Sur les processeurs haut de gamme, grande issuewidth, avec une exécution out-of-order à haute fréquence, couplé

avec de grands caches de premiers niveaux ont largement atténué la plupart des coûts dans l'exécution des interpreteurs. Mais le saut indirect qui met en oeuvre l'instruction switch reste difficile à prévoir, car il a des centaines de cibles (de un pour chaque niveau opcode bytecode) potentiellement, et la sagesse conventionnelle considère la performance des prédicteurs de branchement sur ce saut indirect comme un obstacle majeur.

Dans cette étude, nous revisitons les travaux antérieurs sur la prévisibilité des instructions de branchement dans interpreteurs dans trois des derniers processeurs de génération d'Intel, et un état de l'art du prédicteur indirecte ITTAGE, en utilisant des interprètes de python, javascript et cli. Nous trouvons que la précision de prédiction de branchement sur ces interprètes est considérablement améliorée au cours des trois dernières générations de processeurs Intel, et sur le processeur le plus récent appelé Haswell l'exactitude de la prédiction est tel qu'il ne peut plus être considérée comme un obstacle à la performance. Nous constatons également que sur Haswell, la pénalité de mauvaise prédiction ajoute seulement un surcoût limité au temps d'exécution des interpréteurs, en moyenne 7,8% des issues slots sont gaspillées en raison de mauvaises prédictions de branchement. Par conséquent, la sagesse conventionnelle sur le caractère imprévisible des branches indirectes dans la boucle de dispatch, et son impact sur la performance des interpreteurs, est injustifiée

Modélisation de la performance des programmes multi-thread dans l'ère many-coeurs

Deux modèles précédents, la loi d'Amdahl et Loi de Gustafson, sont largement utilisés pour extrapoler le potentiel de performance d'une application parallèle sur une machine avec un grand nombre de coeurs. La loi d'Amdahl suppose que la taille de l'ensemble des entrées pour une application reste fixe pour une exécution particulière. La loi de Gustafson suppose que la partie relative du calcul parallèle augmente avec la taille des entrées, mais ignore la section séquentielle. Cependant, pour de nombreuses applications, le temps d'exécution de la section séquentielle augmente significativement avec l'augmentation des entrées, mais également un peu avec l'augmentation du nombre de processeurs. Ces modèles de performance ne saisissent pas forcément avec précision la mise à l'échelle de la section séquentielle des applications, et peut donc mener à des estimations optimistes.

Nous construisons un modèle empirique appelé serial scaling model (SSM), pour étudier la mise à l'échelle des applications MT, en fonction de la taille des entrées et du nombre de coeurs. Quand on utilise le modèle SSM, la forme

générale de la durée d'exécution d'une application parallèle peut être représentée comme suit:

$$t(I,P) = c_{seq} I^{as} P^{bs} + c_{par} I^{ap} P^{bp} \quad (1)$$

Les six paramètres sont obtenus de manière empirique et représentent le temps d'exécution d'une application parallèle en tenant compte de ses entrées et du nombre de processeurs. c_{seq} , as and bs sont utilisés pour modéliser le temps d'exécution séquentiel et c_{par}, ap and bp sont utilisés pour modéliser le temps d'exécution parallèle. c_{seq} and c_{par} sont des constantes de la section séquentielle et parallèle qui confèrent à l'amplitude initiale du temps d'exécution. as and ap sont le *Input Serial Scaling* (ISS) paramètre et le *Input Parallel Scaling* (IPS) paramètre. bs and bp sont le *Processor Serial Scaling* paramètre et le *Processor Parallel Scaling* (PPS) paramètre.

Nous ciblons les programmes qui s'exécutent avec succès sur nos plates-formes matérielles (Xeon (E5645, jusqu'à 24 threads) et le Xeon-Phi (5110P, jusqu'à 240 threads)), et qui ont des entrée qui peuvent être générées avec des facteurs d'échelle connus. Quand on utilise les PMU matériels, l'activité d'une application sur la base du thread est analysée, et le temps passé dans l'exécution de code séquentiel et parallèle i, e. $t_{seq}(I,P)$ and $t_{par}(I,P)$ est obtenu en faisant varier le nombre de threads (P), la taille de l'ensemble des entrées (I). Ensuite, l'analyse de régression est réalisée avec la méthode des moindres carrés pour déterminer les six paramètres de SSM qui correspondent le mieux correspondent aux données expérimentales disponibles.

Nous avons étudié les accélérations possibles extrapolées pour quelques applications en faisant varier le nombre de processeurs de 1 à 1024 et en faisant varier la taille du problème de 1 à 10.000. Les applications comme swaptions, barneshut et bodytrack sur Xeon, ont une bonne mise à l'échelle en parallèle, et leur accélération peuvent être encore entre 1024 à 512 pour 1024 coeurs. De nombreuses applications ont une sublinéaire échelle, par exemple canneal, fluidanimate, survey, deltri, sssp et bfs, où l'accélération maximale possible sera comprise entre 512 et 64 avec 1024 coeurs.

Certaines applications, par exemple des swaptions, ont une section séquentielle négligeable et sont hautement évolutives. Certaines applications ont une partie de code séquentiel presque constante et une partie parallèle à croissance rapide pour chaque taille de l'ensemble des entrées, donc de grandes tailles d'entrée fixé sont nécessaires pour amortir la partie sequentielle constante. Dans canneal et fluidanimate (application complète), la section séquentielle est indépendante de

I et P, et la section parallèle passe à l'échelle quasi linéairement avec I et P. L'amélioration significative de l'accélération est obtenue avec une utilisation plus grande de I.

Dans d'autres applications, par exemple deltri, preflow, survey, boruvka, bodytrack (de applicaiton complète), le code séquentiel passe à l'échelle aussi bien voire un tout petit peu moins bien que le code parallèle. Ces applications ne bénéficient pas forcément des architectures multicoeurs, avec une accélération saturée avec P, malgré l'augmentation de I due à l'impact de la mise à l'échelle du code séquentiel.

Contributions

Voici les contributions de cette thèse: d'abord, nous développons un cadre matériel / logiciel appelé core-tethering pour permettre l'exécution de threads auxilliaires sur un système HMC (Heterogeneous Many Cores). Le Core-tethering ajoute de nouvelles instructions en mode utilisateur comme interface (jouant le rôle d'une sorte de un co-processeur) pour les cœurs auxilliaires sur les systèmes HMC. Ce cadre permet au cœur principal de pouvoir lancer et contrôler directement l'exécution des threads auxillaires, et de transférer efficacement le contexte des applications nécessaire à l'exécution des threads auxilliaires. Grâce à une association plus étroite entre les coeurs, le core-tethering évite le surcoût de la latence d'accès aux petits coeurs pour l'exécution des threads auxilliaires. Ensuite en utilisant la simulation basée sur des traces d'exécution, nous évaluons la pertinence des petits coeurs à exécuter les threads auxiliaires pour un thread séquentiel fonctionnant sur le coeur principal, qui est plus puissant. que sur un ensemble de programmes ayant une utilisation intensive de la mémoire, nous montrons que les threads auxiliaires fonctionnant sur des cœurs relativement petits permettent une accélération significative des coeurs principaux par rapport au prefetching matériel seul. De plus les petits coeurs permettent un bon compromis par rapport à l'utilisation d'un cœur d'une puissance similaire pour exécuter les threads auxilliaires. Nous montrons également que le prefetching effectué par les threads auxillaires sur les petits cœurs lorsqu'elle est utilisée avec le prefetching matériel, peut fournir une alternative à l'utilisation de cœurs plus grands et puissants pour des applications gourmandes en mémoire.

Introduction

This thesis work is done in the context of the ERC DAL project [Sez10], which focuses on enhancing single process performance in heterogeneous many-cores (HMC) processor chips that will feature few complex cores and many simple, silicon-area and power effective cores. Specifically, we investigate the use of simple cores in a HMC for executing helper threads to accelerate sequential execution on the complex (main) core.

In the past decade, mainstream server and desktop markets have adopted Chip Multi-Processors (CMP) i.e multi-cores as the predominant architecture template [PDG06]. More recently even the mobile client devices have made the shift to multi-cores. Further as transistor sizes continue to shrink, industry forecasts are promising the integration of hundreds of cores on a single die in the near future.

The move to multi-cores was motivated by the inability to translate additional transistors on chip to performance increases for the single core. Large monolithic processors are difficult to build due to increased hardware complexity and high power requirements [PJS97]. Additionally limitations to exploiting instruction level parallelism (ILP) put a bound on the achievable sequential throughput. Consequently design efforts were redirected towards chip multi processors comprising of identical, complexity effective and power/performance balanced cores to better utilize the available silicon real estate. Single-chip multi-cores provided a high-frequency high-performance processor for inherently sequential applications, and multiple execution engines with improved, low latency interprocessor communication for parallel applications [ONH+96]. By utilizing additional transistors on die to improve system performance and throughput, multi-cores served to hide the inability to scale single thread performance.

However, as increasing number of cores are integrated on a single die, application characteristics of emerging workloads are shaping the architecture of next generation many core processors towards increased heterogeneity. Emerging

workloads are expected to contain a mix of sequential and parallel applications, and place varied power/performance demands on the architecture. Simply replicating identical cores does not effectively address the needs of these mixed workloads, and hence Heterogeneous Many Core (HMC) [KFJ+03][SMQP09] designs that mix many simple/small cores with a few complex/large cores are emerging as an architecture template to efficiently execute both parallel and sequential workloads.

Heterogeneity in power/performance profiles and core sizes helps to achieve a workload optimized architecture template. In a HMC, few large cores, typically wide issue superscalar machines, are optimized for execution latency while small cores are optimized for execution parallelism and area/power efficiency. Predominantly sequential workloads requiring high ILP (instruction level parallelism) are mapped to high performance large cores, while the power-efficient and throughput oriented small cores target parallel workloads demanding high TLP (thread level parallelism). In addition, to minimize energy consumption, workloads running on large cores can be migrated down to small cores at run-time based on their resource requirements. Thus HMCs can achieve higher energy efficiency and better support diverse workload characteristics compared to symmetric many-cores.

Despite the widespread prevalence of multi-cores, development of scalable parallel applications has been limited to specific application segments where parallelism is easy to find such as scientific computing, transaction procession, graphics, etc - very few applications in general purpose computing have been successfully parallelized. To be able to fully exploit the many cores available in the HMC architecture, the application has to be amenable to parallelization, i.e decomposition of computation into many threads that can be executed concurrently on different cores. However, in the foreseeable future a majority of applications are still expected to feature significant sequential code sections, either as legacy sequential codes or as difficult to parallelize sections in parallel code and cannot otherwise benefit from increasing the core count. Additionally, even on applications that are parallelized into multiple threads, sequential sections of execution such as critical sections, etc constitute bottlenecks to performance scalability on large number of cores. Consequently, improving sequential performance on HMCs will remain key both for single threaded codes and scalability on parallel codes.

In addition to traditionally strengthening the large core itself, there is a renewed interest in utilizing additional cores to accelerate single thread performance, especially for memory intensive programs. We propose that the many small cores in a HMC be utilized as low-power helper cores to accelerate the

sequential thread running on the large core. Specifically we explore the porting of the helper threading paradigm [DS98][CSK+99] to the HMC architecture template, small cores can be used to run precomputational code and generate prefetch requests on behalf of the main thread. Instead of turning off small cores when sequential(main) threads are mapped to large cores, helper threading allows them to be employed for achieving even higher sequential performance on the main thread.

When using small cores to execute helper threads in a HMC, helper-execution related actions such as thread-spawn and synchronization incur operating system overhead and overheads due to inter-processor communication latencies through the shared caches. In addition, there is a performance disparity between the large and the small cores, and it is not clearly understood if helpers executing on the small cores can provide sufficient lookahead to benefit the main thread running on a much powerful large core. In this thesis, we focus on efficient helper threading on small/simple cores in a HMC processor to improve sequential performance on memory intensive programs running on the large core in a HMC. We propose a hardware/software framework for helper threading that reduces overheads for spawning and controlling helper threads, and thus enables efficient execution of helpers in the HMC. Using trace based simulation and programs from standard benchmark suites, we also study the suitability of using small cores in the HMC for executing helper threads to accelerate the application thread running on a much larger, powerful main core.

Contributions

The following are the contributions of this thesis: First, we develop a hardware/software framework called core-tethering to support efficient helper threading on heterogeneous many-cores. Core-tethering adds new user mode instructions that provide a co-processor like interface to the helper cores in a HMC, allowing a large core to directly initiate and control helper execution, and to efficiently transfer application context needed for execution to the helper core. Through tighter coupling of cores, core-tethering overcomes the latency overheads of accessing the small cores for helper execution. Next, using trace based simulation we evaluate the suitability of small cores to execute helper threads that can prefetch for a sequential thread running on the larger main core. On a set of memory intensive programs from the standard benchmark suites, we show that

helper threads running on a moderately sized small cores can significantly accelerate much larger main cores compared to using a hardware prefetcher alone, and that small cores provide a good trade-off against using an equivalent large core to execute the helper. We also find that helper prefetching on small cores when used together with hardware prefetching, can provide an alternative to growing the core size for achieving higher performance on memory intensive applications.

Organization of the thesis

This thesis is organized as follows: Chapter 1 provides background information related to Heterogeneous Many Cores architectures, and Helper threading techniques. Chapter 2 presents a survey of prior work on using additional cores for improving sequential performance, and various hardware techniques for closer coupling between cores in loosely coupled systems. Chapter 3 presents a description of core-tethering, the hardware/software framework we propose to support efficient helper threading on heterogeneous many cores. Chapter 4 presents our trace based methodology for simulating helper threading on HMCs and an evaluation of the suitability of small cores to execute helper threads in a HMC. Material from Chapter 3 and Chapter 4 were presented at SBAC-PAD 2014 [SKS14]. Lastly Chapter 5 presents a summary of other contributions, as a second author, to studies in (1) understanding impact of state of the art branch predictors on performance of interpreters and (2) performance estimation of multi-threaded programs on massively parallel many-cores architectures. Material from Chapter 5 are from publications in CGO 2015 [RSS15](to appear) and WAMCA 2014 workshop [NSS14] respectively.

Chapter 1

Background

As the number of cores on die increases with the scaling of silicon process technology, the strategy of replicating identical cores does not scale to meet the performance needs of emerging workloads that are a mix of sequential and parallel applications. Heterogeneous Many Cores (HMC) [KFJ+03][SMQP09] that mix many simple cores with a few complex cores are emerging as a design alternative that can provide both fast sequential performance for single threaded workloads, and power-efficient execution for parallel workloads. However, on applications that are sequential or contain predominantly sequential sections of code, especially those that are memory bound, increasing core count does not translate into performance. Consequently, there is a renewed interest in techniques such as speculative multithreading and helper threading that can exploit additional cores to deliver performance for sequential code sections.

In this chapter we provide background information related to Heterogeneous Many Cores architectures, and Helper threading techniques. We also discuss issues involved in applying the helper threading technique and the various overheads when helper threading is implemented on loosely coupled systems such as the HMC.

1.1 Heterogeneous Many Cores

With the emergence of multi-core architectures, architects have integrated increasing number of cores on-chip. Depending on the target application workloads, and based on whether single thread performance or workload throughput is to be maximized, two typical approaches to grow core count on-chip can be identified.

1. Large Core CMP.
   In the large core CMP design template approach, architects replicate a small-medium (2-16) number of large cores. Designs of commerical general purpose CMPs such as Opteron [BBSG11] from AMD, Xeon [RTM+10] from Intel, and POWER [SKS+11] from IBM are typical examples of this design template. This design point is targeted at maximizing single thread performance and is best suited to handling a mix of single threaded workloads, or applications with small levels of thread level parallelism, and has been predominantly adopted in the desktop (and laptop) and the mainstream server markets. With SMT support, each core is able to support multiple threads (typically 2 or 4) and consequently designs can be scaled to support applications with moderate levels of thread level parallelism.
2. Small Core CMP.
   In the small core CMP design template approach, architects integrate a large (32-64+) number of small cores on-chip. Designs of commerical microprocessors such as TILEPro [TIL] from Tilera, Niagara [KAO05] from SUN microsystems, Xeon-Phi [CE12] from Intel, and GPGPU compute capable graphics processors [MH11][LNOM08] from AMD and NVIDIA are typical examples of this design template. This design point is clearly targeted at applications that have large levels of thread level parallelism and can be easily decomposed into many threads, but is low on single thread performance. This approach has found acceptance in general purpose but specialized throughput oriented application domains such as high performance scientific computing, graphics, and networking tasks that involve packet processing such as intrusion detection, deep packet inspection.

In the third emerging design template called the Heterogeneous Many Core (HMC) [KFJ+03][SMQP09], architects make a balanced trade-off between core-count and raw single thread performance. Instead of replicating homogeneous, either only large or only small cores, a few large cores are integrated with many small cores.

Heterogeneity in core size and performance capability allows to achieve area-performance goals for diverse workloads under fixed power-dissipation constraints. The large cores are complex superscalar processors, and feature high-end micro-architecture design elements such as a deep, high frequency pipeline, large issue width, out of order execution, several functional units, and aggressive and powerful prefetchers, data-dependence and branch prediction. The small cores are simple processors, featuring shallow pipelines and less-aggressive implementations of

branch predictors, etc. The large cores target sequential code sections demanding high ILP (instruction level parallelism) and are designed to support fast execution on single threaded applications and performance limiting sequential sections in parallel applications. The small cores target parallel code sections demanding high TLP (thread level parallelism) and are designed to support energy efficient execution for parallel workloads.

HMC is a shared memory architecture template, the cores share memory and private caches are maintained cache coherent using a hardware coherence protocol. Cache hierarchy in a HMC consists of multiple levels of private and shared caches. Each small and large cores have their own private L1 and L2 caches, and share a large last level (L3) cache. The cores share an interconnect fabric for low latency high bandwidth inter-core communication.

In the design of HMC architectures, two templates can be observed in how the large cores and the small cores are differentiated, both in terms of micro-architecture and specialization, to target application needs.

1. Performance Optimized and Power Optimized cores.

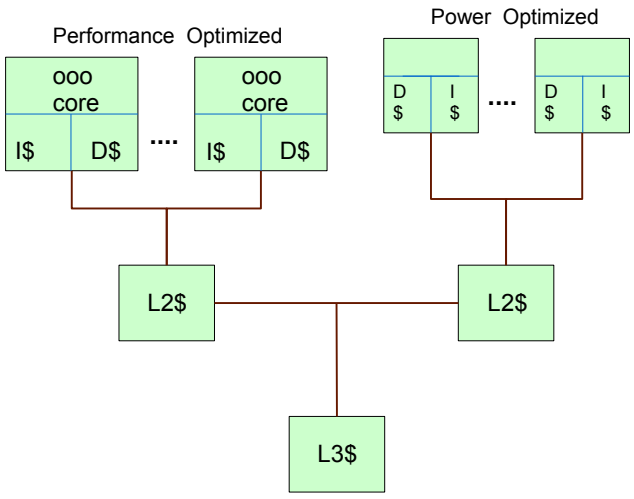


Figure 1.1: HMC Architecture Template - Performance/Power optimized

In this architecture template, cores are differently optimized on the axis of power efficiency. Large performance optimized cores, typically high perfor-

mance and complex out of order superscalar cores are combined with small power optimized cores, typically in-order or simpler out of order processors. Figure 1.1 shows a schematic illustration of the HMC design template where cores are optimized to trade-off power-performance. In processors that target energy efficiency such as ARM [ARM] and Variable SMP from Nvidia [Nvic], the workload is dynamically migrated between energy efficient (low performance) processors and high performance processors based on the performance needs. Alternately, low power cores can be used to offload and execute parallel code sections of the application in a power efficient manner. For example, the Cell Broadband Engine Architecture microprocessor [CRDI07] from the STI (Sony, Toshiba, IBM) alliance, was cost-effective in executing computationally demanding tasks from a wide range of applications, particularly video processing and graphics rendering for consumer appliances like game consoles.

2. Latency Optimized and Throughput Optimized cores.

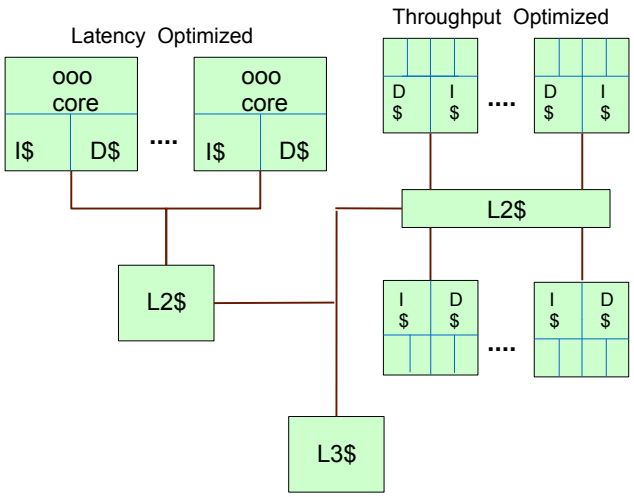


Figure 1.2: HMC Architecture Template - Latency/Throughput optimized

In this architecture template, cores are differently optimized to target either providing the fastest execution latency or better execution throughput. Figure 1.2 shows a schematic illustration of the HMC design template where

cores are optimized along the compute latency-throughput axis. Throughput oriented simple cores that can sustain a large number of application threads are integrated with complex latency oriented cores that provide fast execution by employing high frequency, deeply pipelined and out of order execution cores. Applications that expose parallelism are run on the small cores and can benefit from the enormous raw compute capability of these throughput optimized cores. Commercial microprocessors such as the Fusion Accelerated Processing Unit from AMD [FSB+11], Nvidia TegraK1 processor [Nvia] and Integrated CPU/GPU processors from Intel are widely used to offload tasks in domains such as scientific computing, graphics, media and image processing to the small cores. For example, the recently introduced TegraK1 processor offloads computationally intensive computer vision tasks in the automotive domain to the smaller cores of the Kepler based GPU [Nvib].

An orthogonal dimension of heterogeneity in cores is the diversity offered by different ISAs, where processors that execute completely different instruction sets are integrated on the same die. From this perspective, integrated CPU-GPU processors from AMD, Intel and NVIDIA could be considered as having separate ISAs for the large cores based (x86/ARM), and a custom ISA for the small GPU cores. Recently heterogeneous-ISA processors that exploit multiple different ISAs have been shown to be better than same-ISA heterogeneous (micro-architecture based) architectures in both performance and energy savings [VT14]. In this thesis, the target baseline architecture is a single-ISA heterogeneous many cores and is similar to the ARM big.LITTLE template. The focus of our study is the efficient execution of pre-computation based helper threads on small cores to improve sequential performance of memory intensive programs mapped to the large cores.

1.1.1 Improving Sequential Performance on Many Cores

HMC architecture targets the needs of a diverse workload that is a mix of serial and parallel applications. Large cores are optimized for execution latency, while simple cores are optimized for execution parallelism and area/power efficiency. When executing a workload with low thread level parallelism, threads are mapped to large cores and benefit from fast sequential execution on the complex microarchitecture. When executing parallel workloads, threads are mapped to the numerous simple cores and realize performance by exploiting the high thread

level parallelism available in the architecture. Additionally when the compute demands are low, threads can be migrated from the large core to energy efficient small cores to achieve greater execution efficiency.

To be able to effectively utilize the many cores in the HMC architecture for performance gains, an application has to be amenable to parallelization, ie decomposition of computation into multiple threads that can be executed concurrently. However, many applications are still expected to feature significant sequential code sections, either due to prevalence of legacy sequential code or due to difficulty in fully parallelizing applications, and therefore improving sequential performance will remain critical to realizing the full performance potential of the HMC architecture template. Here we categorize previous studies in the literature that focus on improving sequential performance in many core architectures.

1. Growing core complexity to increase single thread performance.

   Traditionally with every new generation of process technology shrink, architects have relied on increasing core clock frequency, and the use of additional transistors for improved core-IPC, to increase single thread performance. However, clock frequency increases have plateaued because of power dissipation concerns, and increasing IPC remains as the lever to realize higher single thread performance. To improve core-IPC, in addition to widening the execution pipe by increasing the instruction issue width [STR02], recent works have focused on bettering the execution efficiency of the pipeline. By improving branch prediction mechanisms [JL01] [Sez11], and by designing more effective data prefetchers [KKS+14] these techniques reduce the amount of cycles wasted in execution when branches mispredict or when memory accesses miss in on-chip caches thereby reducing the average number of cycles required to process an instruction. Further, with the availability of a large number of transistors, architects have proposed specializing few cores in a HMC specifically for the high performance execution of sequential code sections [MSS10].

2. Dynamically Merging Simple Cores to improve single thread performance.

   Many core architecture templates that are designed to target parallel workloads use simple multi-threaded cores in order to achieve maximum execution throughput for a given silicon area size. However, for applications that are only weakly parallelized and hence still feature significant sequential execution phases, performance improvement is more likely achieved by using fewer, but more capable complex cores. In order to improve sequen-

tial performance on throughput oriented many core architectures, architects have proposed techniques that combine two or more simple cores dynamically at runtime to function as a high performance out-of-order core. In [TBS08] Tarjan et al. propose Federation cores, a technique for creating a 2 way out of order superscalar processor by combining two neighboring multithreaded inorder cores. The large register file of the multithreaded core is repurposed to implement out of order structures, also other associative structures are reworked into simpler to implement lookup tables. Core fusion [IKKM07] proposes a reconfigurable CMP architecture that dynamically morphs groups of independent 2 issue out of order cores (upto 4 adjacent cores and their I and D caches) into a more complex and capable larger CPU.

3. Utilizing additional cores for single thread performance.
   In a many core architecture, when executing sequential code sections, only one core is active while the remaining cores remain idle. Instead of remaining idle, additional cores can be used to accelerate the sequential execution on the active core. Previous works that exploit additional cores for improving single thread performance can be broadly categorized as either thread level speculation or helper threading.
   (a) Thread level speculation
   Thread level speculation (TLS) [SCZM05] or speculative multithreading [KT99] aims to exploit parallelism that exists in sequential code sections at a granularity that is much larger than a typical hardware instruction window (of the order of 128 - 256 instructions). In traditional out of order execution processors, the micro data flow engine tracks independent instructions dynamically within a hardware instruction window and initiates parallel execution of multiple independent instructions every cycle. The execution engine uses branch prediction and data dependence prediction to speculate that instructions are independent, dynamically tracks instruction execution for dependence violation and re-executes instructions violating dependence relation to respect sequential semantic of instruction execution. TLS mechanisms similarly exploit speculative parallelism that exist at much larger granularity - such as loop iteration level [WDY+06] or function level [CO98] granularity.
   A compiler or profiler marks code sections that can be speculatively executed in parallel, such as multiple iterations of a loop nest, or over-

lapping code following a function call with the execution of the call. At run time, speculative threads are spawned to execute in parallel with the main application thread, and sequential semantic is enforced by hardware i.e. execution is monitored for violation of dependence relations in which case threads are re-executed to respect dependences. Thus additional cores are utilized to execute speculative threads extracted from a sequential code section, and can result in significant speed ups when dependences between threads can be correctly predicated.

(b) Helper threading

Helper threading uses a specially constructed speculative thread that executes on an otherwise idle core, in parallel with the main thread without modifying its semantic. The aim of helper thread execution is to accelerate the execution of the main thread by providing performance hints such as direction-prediction for hard to predict branches or by prefetching cache lines that will be referenced by the main thread. The instructions executed by the helper thread themselves do not carry out any of the computation for the main thread, but only provide performance hints. This is in contrast with speculative multithreading where the work done by the additional (speculative) threads are committed to the execution state of the main thread when speculation is validated to be correct. Helper threading is discussed in more detail in Section 1.2.

We study the porting of the helper threading paradigm to the Heterogeneous Many Cores architecture template, i.e the use of small cores as helper cores to benefit sequential applications mapped to the large core. We are interested in improving sequential performance of memory intensive applications that frequently miss in the on-chip cache and spend a large portion of the execution cycles waiting for the memory. While computer architects use prefetching techniques to anticipate future memory access requests, prefetching accurately and sufficiently in advance for applications with complex memory access patterns still remains a challenge. Alternately, helper threading uses small cores to execute backward program slices targeting specific loads/stores to pre-compute memory addresses and generate prefetch requests.

1.2 Helper Threads

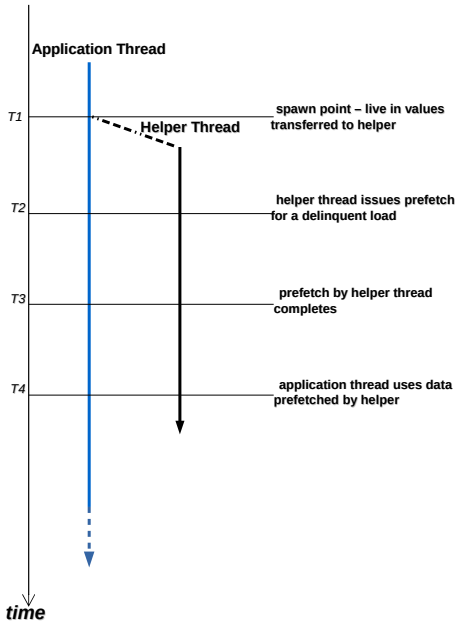
Helper threading can be considered as a prefetching technique where a specially created pre-compute thread runs ahead of the main thread and initiates future memory accesses for the main thread ahead of time. Figure 1.3a illustrates the working of helper threading based prefetching. The main thread triggers execution of the helper thread by delivering the live in variable needed to initiate helper execution (T1). Once initiated, the helper thread runs concurrently with the main thread, executes precomputation code to prefetch addresses that a targeted load (in the main thread) will request in future(T2,T3). When future memory accesses are initiated sufficiently ahead of time, main thread benefits from the latency hiding effects of a prefetch, ie a demand access by the main thread thread finds the prefetched data in the on-chip cache and avoids the memory access penalty(T4). As illustrated, in figure 1.3b helper executes only the code that is necessary to generate prefetches (back-slice) for the targeted loads/stores.

Memory accesses targeted for prefetching by helper threads are typically difficult to prefetch using traditional prefetchers. Such memory accesses may have unpredictable access patterns or a chain of dependent loads (such as pointer-chasing accesses) and are hence difficult to track by hardware prefetchers. Alternately, they may have complex access patterns that require significant computation to generate future memory accesses and hence incur large overheads when targeted by software prefetching. In a helper thread based prefetch mechanism, the overheads of generating computation are hived off to the precompute code in the helper without incurring performance overheads in the main thread.

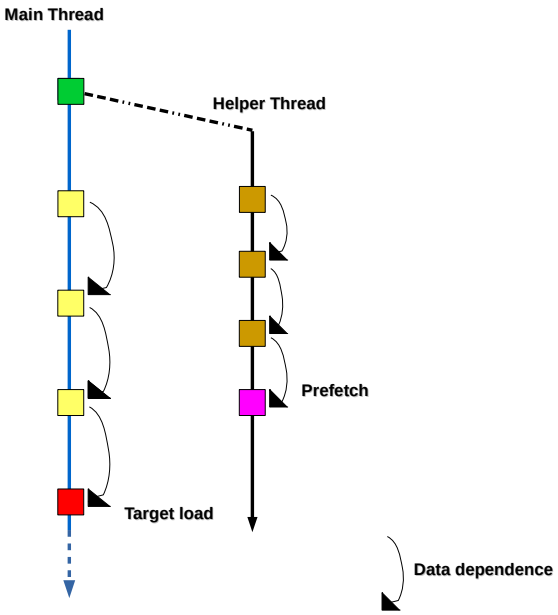
1.2.1 Issues in applying helper threading to accelerate a sequential thread

1. Identifying target instructions and Generation of helper thread

   In order to keep the helper thread running ahead of the main thread, it is necessary that the helper only execute a limited subset of the instructions executed by the main thread. Therefore, only a small set of static loads/stores in the main thread that frequently miss in the on-chip cache and cause most of the cache related stalls - the delinquent loads/stores - are targeted for helper prefetching. Next, starting from the target load/store, a back-slice of instructions that participate in target address generation is



(a) Timeline for Helper Thread Operation



(b) Slicing to construct helper threads

Figure 1.3: Prefetching for an application thread using helper thread

identified. This back-slice along with the instructions required to maintain correct control flow from the pre-computation code in the helper thread. Additionally, loads that are not used later in the helper thread, either to generate address or to maintain control flow, are converted into software prefetches.

Target identification and helper generation techniques can be broadly classified into

(a) static/offline techniques

Helper threads can be generated statically using offline profile information to guide delinquent selection and precompute slice construction. Here a profile run of the target application collects information on the loads and stores that cause the most cache misses, which is then used by a static analysis phase which performs back-slice construction starting at these target instructions. In [BWC+02] Brown et al. used static analysis of program source code in a compiler to generate pre-compute code for the helper. Alternately, in [LWW+02] Liao et al. used static analysis directly on the binary to identify the back slice.

(b) dynamic/runtime techniques.

At runtime, delinquent loads can be identified and helper can be constructed dynamically either using special hardware support or as part of a software dynamic optimization system. In [CTWS01] Collins et al. use a hardware structure known as the Delinquent Load Identification Table to identify delinquent loads, and a Retired Instruction Buffer (RIB) that stores a trace of committed instructions to construct the back-slice. In [LDH+05] Lu et al. implement helper threading within a software dynamic optimization system on an unmodified CMP. Profiling and delinquent load identification are implemented within runtime performance monitoring code, and slice construction and helper generation as part of the JIT compiler.

Helper threading based on statically contructed pre-compute slices are easily adopted with existing hardware architectures, and have found support from microprocessor manufacturers [PSG+09][WWW+02].

2. Maintaining the semantic of the main thread

Helper threads are speculative in nature, they must respect the semantics of sequential execution of the main thread and cannot change its program execution state. To implement helper threading, the processor has to support

a speculative pre-execution model where the results computed in the helper threads are never integrated into the main thread, and any exceptions signaled in the pre-execution context of the helper thread never disrupt the execution of the main thread. In addition, helper threads are not allowed to write to memory locations since a write to memory can potentially change the architectural state of the main thread. Instead store instructions targeted in the helper thread are either converted to prefetches, or localized to a separate memory location within the helper thread.

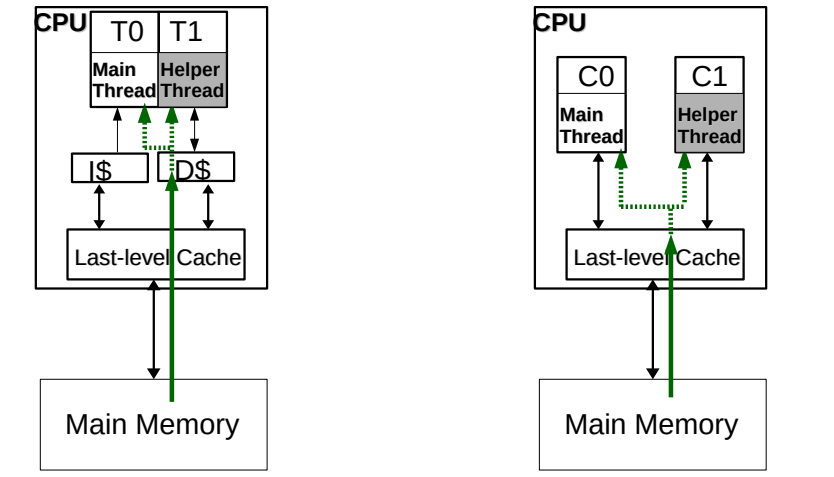
3. Maintaining efficiency of prefetching.

   For effective prefetching, the helper threads ideally runs only as far ahead as is adequate to prefetch in time for the main thread. When the helper thread lags behind the main thread, the prefetches it generates are ineffective. When the helper threads runs too far ahead of the main thread, it risks polluting the cache by evicting useful lines that are still needed by the main thread. Therefore during execution, the helper thread synchronizes periodically with the main thread to ensure it is sufficiently ahead of the main thread to be able to launch in time prefetch requests, called the look ahead distance. To implement synchronization, typically for a helper thread targeting a loop region, the main and the helper threads both maintain a counter to track the iteration count they are currently executing. Periodically, the helper thread reads the counter of the main thread and compares it with its own counter. When the helper is too far ahead of the main thread, ie its advance over the main thread is greater than the desired look ahead distance, the helper pauses execution of the precomputation slice and generation of prefetch requests until the main thread makes sufficient progress. When the helper lags the main thread, it resynchronizes its execution state with the main thread, copies over updated values for its live-in variables and restarts execution.

1.2.2 Overheads of helper threading on loosely coupled systems

Helper threading was originally proposed as a technique that could utilize unused thread contexts in a SMT machine to speedup serial execution. In a SMT processor, the threads are tightly coupled and share the first level cache, and also execution resources such as functional unit, register file, issue queues, etc. Since the register file is shared, register values from the main thread can be easily

passed to the helper thread as live in variables by using a register to register copy. Fast synchronization is possible since the two threads can communicate through the shared L1 cache. Additionally, the helper thread can prefetch directly into the shared L1 data cache, and a successful hit in the main thread can completely avoid the memory access latency. The main thread and the helper thread are located together on the same core, lesser overheads are incurred during helper initiation and synchronization. However since the main thread shares execution resources with the helper, contention for shared resources can slow down the execution of the main thread. Careful design of the helper is needed to minimize resource requirements. Figure 1.4(a) illustrates helper threading in a tightly coupled SMT machine.



(a) Helper Threading in closely-coupled SMT (b) Helper Threading in loosely-coupled CMP

Figure 1.4: Prefetching for SMT and CMP machines

In a loosely coupled system such as the CMP or the HMC, helper threading techniques are extended to harness the unused core for helper execution. Helper threads now get an execution context that does not contend for resources and slow down the main thread. However, helper thread and the main thread are not located on the same core, and hence can only share a lower level in the cache hierarchy. This implies a greater penalty for intra-thread communication and

higher overheads for thread spawn and thread synchronization. Figure 1.4(b) illustrates helper threading in a loosely coupled machine.

On a loosely coupled architecture such as the HMC architecture, significant operating system overhead is incurred in spawning the helper thread. Unlike in a tightly coupled architecture where values can be communicated through a register to register copy with the core, in a loosely coupled machine the main thread has to copy over values that are live-in variables for helper execution. This incurs a significant additional communication overhead during thread initialization.

Communication between the helper thread and the main thread is limited to the shared lower level of cache and incurs large latency penalties. When the helper thread and the main thread synchronize periodically to control helper look ahead for prefetch effectiveness, the value of the progress counter in the main thread is communicated to the helper thread. In addition, live-in values have to be communicated again from the main thread if the helper requires reinitialization to catch up with the main thread. Lastly, the cache lines prefetched by the helper thread reside in a shared lower level cache, and the main thread incurs an additional latency to access the shared cache even when the prefetch is successful.

Chapter 2

State of the art

In this thesis, we are studying the porting of helper threading to the emerging HMC template. In the previous chapter, we presented background information related to heterogeneous many cores and the helper threading technique. This chapter presents a survey of the prior work on using multi-cores for improving sequential performance, with a focus mostly on implementations of helper threading. We present a categorization of previous literature based on how the additional cores are utilized, and summarize select relevant papers. The second section highlights specific use cases where a small core is used as helper to a larger core. The third section presents a summary of recent work in hardware support for tighter coupling of cores.

2.1 Exploiting additional cores for sequential performance

Before the shift towards multi-cores, utra-wide issue, high frequency processors with large instruction windows were pursued as the design direction to dramatically increase sequential throughput [PPE+97]. In the multi-core era, wide issue processors have not materialized due to design complexity issues [BG04][PJS97], and single core frequencies have largely decreased to accommodate additional cores on chip at the same thermal design points. Instead, increases in sequential performance have been driven by traditional optimizations for IPC. ISA extensions such as SSE/AVX from Intel and NEON from ARM have improved performance on floating point intensive multimedia, scientific , and sig-

nal processing applications. Further, improvements from large on-die caches and increased on die-integration have driven down memory and I/O latency and enhanced performance [CH07].

As an alternative to a monolithic complex core approach to boost single thread performance, research efforts have explored mechanisms to exploit additional on chip resources of a multi-core, both compute (cores/threads) and cache capacity, to improve sequential performance. In this section, we categorize these previous research efforts based on the way in which additional resources are employed to augment the performance of the main sequential thread and present a summary of select relevant papers.

2.1.1 Using additional cores to run pre-computation based helper threads

In this approach, traditional slicing based helper threading techniques for SMT processors [CSK+99] are extended to work in a CMP (chip multi-processor) framework. Typically in a CMP processor, when a sequential thread executes on one of the cores, the other cores remain idle and are clock gated to conserve power. In order to boost sequential performance of the main core, the idle cores are utilized to construct and execute helper threads. Unlike in a SMT execution, resources and pipeline stages are not shared between the main thread and the helper thread and hence helper threads can be spawned and executed without concerns of slowing down the main thread.

Summary of relevant papers

In [BWC+02] Brown et al. first proposed that speculative precomputation can be applied to loosely coupled CMP architectures by using one core to run the main thread and the other cores to execute precomputation code and generate prefetches for the main core. It was demonstrated that despite sharing only the last level L3 cache, speculative precomputation can provide non trivial speedups on memory intensive single threaded workloads. Inter-core resource independence, ie lack of shared resources closer to the core was identified as a major obstacle to fully realize the benefits of helper generated prefetches, and two improvements were proposed to improve performance effectiveness of speculative precomputation on CMPs by reducing the access latency to prefetched cache lines (1) *Cache Return Broadcast*, when the shared L3 cache services one core's L2 miss, the cache fill is broadcast to all active cores (2) *Peer-core L2 Cross-feeding*, when one core sees a L2 miss and submits a L3 request, other ac-

tive peer cores snoop on the L3 request and respond with the cache line if it is found in their private L2.

In [LDH+05] Lu et al. present a dynamic optimization system for helper thread based prefetching on CMP systems. A light weight dynamic optimizer runs on the additional core and monitors program performance to identify code regions that have poor memory behavior for optimization using helper threads. The optimizer also identifies delinquent loads and stores within the target region, generates the helper thread and inserts calls to the spawn the helper thread. The execution of the helper threads are interleaved with the dynamic optimization system, thus the helper thread run on an independent processor (different from the main thread) to issue prefetches that bring in data into the shared L2. A software mail box mapped to shared memory is used to communicate values between the main thread and the helper, both during initiation of the helper thread and during the periodic synchronization between the threads. To ensure that the helper thread executes along the same path as the main thread, an asynchronous protocol that avoids cross checking progress of threads is used. After a certain number of iterations (synchronization interval), the helper thread terminates and yields to the dynamic optimization system while the main thread passes updated values of live-in variables and continues execution in the next synchronization interval. This allows the helper to finish within a certain time limit so that tasks in the dynamic optimizer such as phase detection, etc are performed in a timely manner and also ensures that the main thread never waits on the helper for synchronization. When the helper thread is restarted, it compares its progress with the main thread, synchronizes with the main thread if needed and continues execution of precomputation code.

In [SKT05] Song et al. present a compiler framework to automatically construct helper threads for CMPs. Their helper thread methodology works without special hardware support and targeted loops in the program, including techniques to select candidate loops, and to decide which candidate loops are profitable for helper prefetching. They employed a fork-join model based auto-parallelization infrastructure to manage dispatching of helper threads. For synchronization between the threads, communication of live in variables to the helper thread is handled by the parallelization runtime and is allocated in a shared memory region. Compiler generated code inserts checks to verify if the the helper thread is running behind the main thread and terminates execution of the helper thread when the helper trails the main thread in execution. Prefetches issued by the helper thread bring in data to the shared L2 cache, and experimental results showed that their

production compiler working on an unmodified dual-core SPARC microprocessor improved performance on codes suffering large L2 cache miss penalties by upto 22%.

In [SKKC09] Son et al. present a scalable, compiler directed helper based prefetching scheme on CMPs for multi-threaded programs. For multi-threaded programs, a naive implementation of helper threading would assign one helper thread for each thread in the application. Alternately, the authors use static compiler analysis techniques to identify program phases, then divide application threads into groups within each phase and assign a custom generated helper thread to each group of threads. Synchronization between the main thread and the helper thread is managed explicitly by the compiler. By using a helper thread per group, instead of a helper thread per thread, overheads associated with prefetching as well as harmful prefetches are reduced rendering helper directed prefetching a scalable optimization for increased core counts.

In [LJLS09] Lee et al. targeted the overheads of helper thread based prefetching in loosely coupled systems such as CMPs and in-memory processors. The authors proposed to use large loops for helper based prefetching, instead of extracting a program slice per delinquent load instruction, thus amortizing the overheads of communication and thread management. Also, interprocessor communication overheads through a synchronization mechanism that uses lightweight semaphores and is tailored for helper threads targeting loop nests that synchronizes at the granularity of a few loop iterations. Synchronization precisely controls how far ahead the execution of the helper thread can be with respect to the application, and is helpful for prefetch timeliness and to avoid cache pollution. With reduced overheads, they demonstrate that helper based prefetching can be done effectively in an unmodified off-the-shelf CMP.

Lee et al [LJLS09] also propose that a loosely coupled small core embedded in memory (either in the DIMMs or DRAM chips) be used to run helper threads. The helper core is now closer to memory and is able to prefetch data with much lower memory access latencies. Special synchronization registers in the memory processor are accessed by the main processor as co-processor registers to communicate synchronization variables between main and memory processors with low overhead. The memory processor executes precomputation back slices and prefetched cache lines are directly installed in the L2 cache of the main processor.

Yang et al. [YXMZ12] presents a preexecution scheme where the CPU executes helper threads and prefetches cache lines for a kernel running on the GPU, thus improving the performance of GPU compute in fused CPU-GPU architec-

tures. After launching a compute kernel on the GPU, the CPU executes a pre-execution thread that is constructed automatically by the compiler from the GPU kernel, and generates prefetches for memory accesses in the compute kernel. The pre-execution threads runs ahead of the compute kernel both because it executes only a subset of instructions executed on the GPU and also because the CPU runs at a higher frequency and better exploits instruction level parallelism. Hence, memory accesses from the GPU hit in the shared L3 cache reducing the memory access latency for the compute kernel resulting in an average performance improvement of 21.4%.

Kamruzzaman et al. [KST11] present Intercore Prefetching where multicore based helper threading is coupled with thread migration that allows the main thread to access prefetched data in L1 data cache of its core and completely avoid the memory access latency. Multiple processor cores are used to accelerate a single thread of execution, one of the cores to execute the main thread i.e perform the computation and other cores to run helper threads that prefetch data for the main thread. In intercore prefetching program execution is divided into chunks, where each chunk roughly corresponds to a set of loop iterations. While the main core executes the original code in chunks, the helper threads execute a distilled version with precompute code to compute addresses and bring data into the cache. Both the main and prefetch threads execute one chunk at a time, with the prefetch threads always executing the precompute slice for a chunk ahead of the main thread. When the main thread has finished executing the current chunk, it executes a migration routine and is switched to the core that executed the prefetch thread for the subsequent chunk. The main thread and the helper threads continue migrating between cores, with the main thread following the helper threads so that the data needed for execution is already in the caches.

Hassanein et al. [HFE04] present In-Memory Precomputation Threads (IMPT) that combines precomputation-based techniques with the low memory access latency of processing-in-memory. A simple in-memory processor is used to precompute future, critical load instructions and forward the resulting load values directly to the main processor. Slice generation and insertion of trigger instruction into main thread to invoke the slice are implemented in software at compile time. Initially during slice construction individual slices are restricted to basic block boundaries, and later slices requiring the initialization of the same registers are combined to form slices that spawn basic blocks. Trigger instructions for a slice are inserted after the most recent definition of the registers needed for

slice execution. These triggers spawn the execution of precompute slice in the memory processor, and synchronize registers between main and memory threads by supplying register initialization values needed by the slice. A trigger history table (THT) monitors the time between a precompute slice is triggered and the execution of the corresponding slice in the main thread (trigger lead time), and executes those slices that have a large lead time so that prefetches are in time and precompute execution is profitable. Additionally, IMPT bypasses the caches and directly forwards load values to the main processor using an Instruction Validation Table (IVT) if the value is not stale (i.e. address does not exist in dirty state in the L1/L2 caches or Load-Store Queue). When the main processor generates the address of a targeted load instruction, the address is checked against entries in the IVT and on a valid match the value is directly transfered to the destination register of the load. Overall in-memory pre-execution has the advantage of direct access to data in memory at low latency, decreasing average load access latency by up to 55% resulting in a performance gain of up to 1.47 over an aggressive superscalar processor.

Lau et al. [LMC+11] proposed Partner Cores, small cores optimized for low area and power consumption that are paired with larger, faster compute cores and perform optimizations to improve performance of the main core. Partner cores are tightly coupled to each main core, and can directly inspect key hardware such as L2 cache, status registers, etc. Partner cores can be programed to run helper threads that can prefetch cache lines on behalf of the main core. On receiving a helper trigger and the live in register data, helpers run independently on the partner core and generate prefetch commands that are communicated from the partner core to the main core via a FIFO interface. Since the partner core and the main core share the L2 cache, a prefetcher on the main core reads these prefetch commands and prefetches cache lines directly into the L2 cache.

2.1.2 Decoupled LookAhead Processors

Similar to the helper based prefetching scheme, decoupled processors use an additional core for look ahead, i.e to anticipate branch mis-predictions or cache misses, to improve performance for the main thread. While in helper threads prefetching is targeted to specific subset of instructions called delinquent loads/-stores, in decoupled processors look ahead is a continuous process that targets to mitigate almost all cache misses and branch mispredictions seen by the main thread. In helper threads, multiple helper threads are individually spawned to tar-

get one or more delinquent loads/stores and the pre-computation code consists of back-slices of the main thread leading to the delinquent accesses. In contrast, decoupled look ahead processors employ a standalone lookahead instruction stream that runs ahead of the main thread and initiate prefetches. The lookahead thread is typically a reduced (skeleton) version of the main program that only helps to prefetch and to help branch predictions.

Summary of relevant papers

In [Zho05] Zhou presents a decoupled execution technique called Dual Core Execution (DCE) where two relatively simple super-scalar cores are coupled with a queue. The 'front' processor executes instructions normally, except for L2 misses which generate an invalid value instead of blocking the pipeline, and retires instructions in-order into a result queue. The 'front' processor does not update memory, instead store results are buffered in a small 'run-ahead cache' for later use by load instructions. The 'back' processor consumes instructions from the result queue and updates memory and architectural state. The 'front' processor is able to run-ahead since it does not stall on long latency L2 cache misses, also 'back' processor is helped by 'front' processor because most branch predictions are resolved and data caches are warmed-up. When a branch mis-prediction is detected in the 'back' processor, the 'front' processor recovers from the misprediction by using the architectural state of the 'back' processor. All instructions in both the cores are squashed, result queue and run-ahead cache are flushed, and 'front' processor resumes execution after being synchronized to the precise architecture state maintained by the 'back' processor.

In [GH08] Garg and Huang present the Explicitly Decoupled Architecture (EDA) to conceptually and physically decouple the machine into a performance domain (core) and a correctness domain (core). A skeletal version of the program is run on a separate performance core and acts as the lookahead thread that provides performance hints. The execution of the lookahead thread is optimistic, off chip memory stalls are avoided by feeding an arbitrary value to the stalled load instruction. Branch resolution results are communicated to the correctness core using a hardware queue, and data prefetches issued by the performance core bring data into the shared L1 cache for use by the correctness core. Lookahead is completely standalone, and no communication or synchronization is required between the two cores. When the correctness core discovers that branch paths in the two cores have diverged, a recovery is initiated in the performance core, pipeline is drained and performance core restarts execution after copying architectural state from the correctness core.

In [GPH11] Garg et al. present Speculative Parallelization of Decoupled Lookahead an extension to EDA, the look-ahead agent itself is speculatively parallelized to run ahead of the main thread. Lookahead provided by continuous slice based pre-execution is limited by the speed of the look-ahead code itself and the lead that it provides (called lookahead depth), especially in irregular programs. In the EDA proposal, the skeletal thread that provides the look-ahead is speculatively parallelized allowing the lookahead to speculatively overcome data-dependences and run father ahead of the main thread. The lookahead thread delivers more timely branch hints and data prefetches into the shared cache, which helps to speed up the main thread. Additional hardware for thread spawn/merge and data-cache versioning support to detect dependence violation is required in the core implementing lookahead to support multiple thread contexts and speculatively parallel execution. Similar to EDA, when the correctness core discovers that the branch paths in the lookahead and main threads have diverged, a recovery in initiated in the core executing the lookahead thread and execution restarts after synchronization with the architectural state of the main thread.

In [GB05] Ganusov and Burtscher present a technique called 'Future Execution' that proposes to use the additional core to execute-ahead and generate prefetches for the main core. Instead of executing a specially created precompute based helper thread, the additional core executes non-control instructions that are likely to be executed in the future by the main thread. When a load instruction completes execution in the main thread, a copy of the instruction is executed on the additional core with the address for the *nth* next instance to prefetch data into the shared L2 cache. The address of the *nth* next instance is obtained with the aid of a value predictor - the value predictor predicts input values to the data flow graph (backward slice) leading up-to the load instruction, and the actual address is obtained by pre-executing the backward slice. Future execution is recovery-free, there is no need verify results produced by the lookahead thread or synchronize with the main thread since results produced during future execution are only used to generate prefetches.

2.1.3 Using additional cores to emulate hardware structures

In this approach, additional cores in a CMP are utilized are helper engines, but instead of executing a pre-compute thread or a look ahead thread, they emulate complex hardware performance accelerators such as data prefetchers or hardware

structures such as caches, Additional hardware support is added to a stock CMP to collect and pipe cache miss information of the main thread to the helper core. Ability to emulate prefetchers in software, allows use of prefetch algorithms such as the Markov prefetcher that are considered complex or expensive to be implemented in hardware. Additionally, flexibility of a software implementation allows the generation of prefetchers that are tuned to a particular application.

Summary of relevant papers

Ganusov and Burtscher [GB06] present the event-driven helper threading (EDHT) framework to use idle cores in a CMP as helper engines to emulate prediction based prefetching algorithms and generate prefetches for the main thread. Prefetch tables that hold cache miss history are stored in memory and table sizes are chosen to fit in the first level data cache of the helper core. A hardware event buffer connects the two cores, it receives cache miss events from the main core and provides an ISA level abstraction to the helper thread to access these events. A cache miss in the main thread triggers execution of the helper, which issues an I/O read instruction to read the miss address and associated PC from the event buffer, and computes and issues prefetches to bring data into the shared L2 cache. Emulation based implementation of different prefetching schemes are presented: global and local stride prefetching, differential finite-context method prefetching (dfcm)[GVDB01], and Markov prefetching[JG97], and reported results are within 5% of pure hardware implementations.

Mars et al. [MWU+08] present unobtrusive reactive prefetching (URPref) where a neighboring idle processor is used as a helper core to decouple the tasks of profiling, pattern detection and prefetching from the main application core. A hardware/software interface called the 'snoopy buffer' uses the cache coherence snooping mechanisms to relay cache miss information to the helper core, and a new ISA instruction allows the helper core to read information out of the snoopy buffer. The helper core observes miss patterns collected by the snoopy buffer, and uses the sequitur algorithm [16] to extract hot data access streams. When the start of a known miss pattern is observed, prefetches are issued to the addresses belonging to the sequitur detected pattern. A special prefetch instruction (ISA extension) prefetches cache lines directly into the L1 cache of the main core. Use of the sequitur algorithm allows URPref to continuously profile and adapt to the application's cache miss behavior, predict more complex miss patterns than a hardware based stride prefetcher, and prefetch with a high accuracy based on knowledge of previous misses.

[WL10] presented COMPASS (compute shader-assisted) to use the GPU cores

to emulate prefetcher algorithms in an integrated CPU-GPU processor. Each entry of a prefetch table is emulated with GPU threads, its registers and constant caches are used to record miss history information. Programmable shaders emulate prefetcher algorithms and use a specially added GPU prefetch instruction to prefetch to a cache shared with the CPU. A tight coupling between the two is enabled using Miss Address Provider (MAP), a hardware/software interface that is located between the L2 cache and the GPU. MAP receives the address of the L2 miss (or first hit to a prefetched line) and the program counter (PC) generating this memory request, and sends a GPU command to trigger the execution of a COMPASS generated shader thread. Various prefetching algorithms were emulated and simulation results showed an improved single thread performance on memory-intensive applications.

Woo et al. [WFKL10] present the Chameleon architecture framework to utilize the throughput cores (GPU shaders) to provide additional cache capacity (last level cache) to the main thread. Scratchpad memories (SRAM arrays) in the shader core are converted into a soft cache and microcode (written in the shader ISA) executed on the shader engines provide the logic for cache lookup and control operations. A hardware controller receives the miss address and broadcasts the miss-address to the shader cores. The shader cores then execute microcode that performs cache operations such as index calculation and tag match, and responds to the controller with a hit/miss signal and the the cache line (on a cache hit). On a cache miss, the Chameleon hardware controller initiates an off-chip memory request through its own MSHR. Microcode also maintains status bits needed by the cache replacement algorithm.

Solihin et al. [SLT02] propose a memory-side prefetching scheme called User-Level Memory Thread (ULMT) where a simple general purpose processor in memory (either in the memory controller or DRAM chip) performs correlation prefetching in software, and sends the prefetch data directly into the L2 cache of the main processor. Correlation prefetching involves (a) a learning step where a correlation table records either stride patterns in the address sequence or pair-based correlations for a miss and a sequence of subsequent misses and (b) a prefetching step - when a cache miss is observed, all address correlated with the miss in the correlation table are prefetched. In the ULMT scheme, the correlation table is a software data structure that resides in main memory, with inexpensive accesses to table entries because the memory processor transparently caches the table in its cache and the prefetching algorithm itself with the learning and the prefetching steps are emulated in hardware and executed on the memory proces-

sor. Additionally, the interface between the main processor's L2 cache and the memory is modified to accept prefetched cache lines forwarded by the memory processor so that in case of a successful prefetch the main processor finds that cache line already installed in its L2 cache.

2.2 Utilizing small cores

With the advent of heterogeneous architectures, many studies have used small cores to accelerate the sequential main thread running on large cores. One obvious challenge with this approach is that the small cores are weaker in performance compared to the large cores, and it is not clear if the small cores can provide a sufficiently large advance over the sequential thread.

In case of the decoupled lookahead approach (Section 2.1.2), the helper core has to run a standalone lookahead thread and continuously prefetch for the main thread. Here previous studies have shown that even when an equivalently large core is used as the helper, it is difficult to maintain sufficient advance over the main thread because the lookahead thread stalls on cache misses and branch mispredictions [GPH11] and becomes a bottleneck to achieve large performance benefits. Techniques such as specualtive look ahead parallelization [GPH11] have been proposed to accelerate the lookahead thread itself. It is very unlikely that a small core can be successful as a helper core for decoupled look ahead based techniques, therefore studies using small cores as helpers have focussed on the other two techniques i.e precomputation based helper threads targeted to delinquent loads, and emulation of hardware structures. These were discussed in more detail in Section 2.1.1 and Section 2.1.3, here we attempt to highlight specific use cases relating to small cores as helper cores.

Woo and Lee [WL10] presented a scheme called COMPASS that used idle programmable shaders in a GPU to emulate prefetch algorithms in software and generate prefetches for sequential applications running on the CPU. The shader code reads miss history stored in GPU register files, computes prefetch addresses and issues prefetch instructions to bring data into the L2 cache. Sohilin et al. [SLT02] proposed a memory side correlation prefetching scheme using a small core placed close to memory. Memory processor executes a software thread that observes misses incurred in main processor's L2 cache, and uses correlation between past sequences of addresses to predict and prefetch future misses directly to the main processors L2 cache (the L2 cache is modified to accept prefetch lines sent from

memory).

Lee et al. [LJLS09] propose a small core embedded in memory (either in the DIMMs or DRAM chips) to run helper threads. The helper core executes pre-computation back slices targeting specific delinquent loads and prefetched cache lines are directly sent to the L2 cache of the main processor. Since the small core is placed closer to memory, it has a much lower memory access latency and is able to access data without interfering with the accesses of the main thread. Lau et al. [LMC+11] propose partner cores where a larger compute core is paired with a low area/power partner core that runs optimization code to trigger cache misses ahead of time on behalf of the compute core. Unlike in our study, partner cores are tightly paired with compute cores sharing the L2 cache and use a unified request queue to launch prefetch requests, prefetched lines are directly installed in the main processors data cache.

2.3 Hardware support for tighter coupling of cores

When using additional cores in a loosely coupled system such as CMP or HMC to accelerate the sequential thread running on one core, inter-processor communication latencies and operating system overhead can limit the utility of techniques such as helper threading. Consequently, architects have proposed various hardware techniques for closer coupling between cores. Many of these techniques were discussed in detail along with their respective helper threading techniques in Section 2.1. Here we present a short discussion of these techniques to highlight hardware support techniques that enable tighter coupling of the main and the helper cores.

Support for thread spawn and synchronization

Thread spawn incurs significant operating system overhead and in addition, copying live-in register values from the main thread to the helper adds to the thread spawn latency. Consequently, the bulk of hardware support proposals made for tighter coupling between processor cores target to overcome these overheads. Hall et al. [HLSS13] propose hardware based spawning of the helper thread using a *branch to assist thread* instruction to reduce thread-spawn overheads. Wong et al. [WBS+08] propose Pangaea that uses instruction set extensions to allow a thread on the CPU to directly spawn a user level thread on a gpu core bypassing the overheads of graphics fixed function hardware. To initiate thread spawn, the cpu stores task information (instruction pointer to the task to be ex-

ecuted, and data pointer to read in task input) to an address that is monitored by the thread spawner hardware; thread spawner is an additional hardware unit that monitors thread spawns executed on cpu cores, and initiates thread execution on a gpu core. Hassanein et al. [HFE04] used special trigger instructions to spawn the execution of helper thread on the in-memory processor and to initialize live-in registers in the helper thread. Brown et al. [BWBJ11] employ instruction set extensions in the wire speed processor that allow cpu threads executing on the A2 processor core to efficiently dispatch requests to on-die accelerators. A new instruction ICSWX performs an initiate coprocessor operation by sending a cache aligned coprocessor request block (CRB) to a coprocessor in the system. CRB contains parameters for the co-processor command to initiate execution of an acceleration task, such as the Function Code (FC) to identify the operation requested and an operand block that provides inputs.

Support for synchronization between threads

Efficient implementation of helper threading requires periodic synchronization between the main and helper threads to precisely control the advance helper execution has over the main thread, this is key to in-time prefetching and to avoid cache pollution [LJLS09]. However, in a loosely coupled system such as the CMP or HMC, synchronization is accomplished using shared variables through the cache hierarchy and incur significant overheads due to cache to cache transfer latency. Gschwind et al. [GOSS13] propose direct hardware-assisted communication between threads to bypass the memory hierarchy and enable fast communication of values between main and helper threads. Lee et al. [LJLS09] use special synchronization registers in the memory processor running helper threads, which are accessed as co-processor registers to communicate synchronization variables between main and helper with low overhead.

Support for direct cache injection

In a SMT processor, lines prefetched by the helper are available to the main thread in the shared L1 cache without any additional latency. In contrast, on loosely coupled systems the main thread incurs an additional L3 access latency before it can access the lines prefetched by the helper. To overcome this additional latency overhead, previous studies have evaluated various mechanisms to directly inject prefetched data in the private caches of the main core. Brown et al. [BWC+02] present a technique called 'Cache Return Broadcast', wherein when the shared L3 cache services the helper core's L2 miss, the cache fill is broadcast to all active cores so that the main thread finds the prefeched lines in its private caches. Solihin et al. [SLT02] and Lee et al [LJLS09] modify the interface between

the main processor's L2 cache and the memory to accept cache lines forwarded by the memory processor, such that in case of a successful prefetch the main processor finds that cache line already installed in its private L2 cache. Hassanein et al. [HFE04] use an Instruction Validation Table (IVT) to bypass caches and directly receive load values forwarded by an in-memory processor, and a subsequent targeted load instruction its address is checked against entries in the IVT and on a valid match the value is directly transfered to the destination register.

ISA support for communicating hardware events to user level threads

Helper threads that emulate prefetch algorithms in software need the ability to read the stream of hardware events such as cache miss, etc generated in the main core. To facilitate a low overhead access to the event stream, a special hardware event buffer is added, and new instructions in the ISA to allow a user level helper thread to read from the buffer are provided. In the event-driven helper threading (EDHT) framework of Ganusov and Burtscher [GB06], a hardware event buffer that connects the two cores receives cache miss events from the ROB (reorder buffer) of the main core, and special I/O instructions allow the helper to read the miss address and the associated PC from the event buffer. Mars et al. [MWU+08] use a hardware structure called the 'snoopy buffer' that collects cache miss information from the cache coherence snooping mechanisms, and a new ISA instruction that allows the helper core to read miss information out of the snoopy buffer. Woo et al. [WL10] used a presented Miss Address Provider (MAP) located between the L2 cache and the GPU. MAP receives the address and program counter (PC) of the L2 miss and sends a GPU command to trigger the execution of a COMPASS generated helper thread on the GPU shaders.

2.4 Summary

In this chapter we categorized various different ways in which additional cores can be utilized to improve sequential performance - to run precomputation based helper threads targeting specifically selected loads/stores, to run a decoupled lookahead thread that continuously runs ahead of the main thread and generates performance hints, and as helper engines that emulate in software hardware structures such as prefetchers.

Executing a slice based precompute helper thread is the most studied way to harness the additional core to improve sequential performance on the main thread. Various different techniques are used to generate and execute helper

threads. [BWC+02], [SKT05], [SKKC09], [LJLS09], [KST11] present a static approach where the compiler generates the helper threads , while in [LDH+05] Lu et al. generate helper threads using the runtime JIT compiler within a dynamic optimization system. To run helper threads [BWC+02], [SKT05], [SKKC09], [KST11], [LDH+05] use an additional core in an unmodified CMP, while [LJLS09], [HFE04] use a small but in-memory core that reduces memory latency for helper execution, and [LMC+11] uses a dedicated small partner core tightly coupled to the main core to run the helper.

In this thesis, we base our pre-compute thread construction methodology on static analysis of a compiler generated binary. The cores targeted for helper execution are neither dedicated or specially designed cores, nor are they placed closer to memory. Instead they are unmodified small cores within the HMC architecture that are otherwise idle during sequential execution phases on the main (large) cores. In the next chapter, we present details of our hardware/software framework called core-tethering that reduces overheads for spawning and controlling helper threads, and enables efficient execution of helpers in the HMC.

Chapter 3

A Hardware/Software Framework for Helper Threading on Heterogeneous Many Cores

In this thesis, our aim is to explore the porting of helper threading techniques to HMCs, with consideration to efficient execution of helper threads on the small cores. By executing the helper on a small core in a HMC, helper execution utilizes a separate hardware context that is independent from the main application thread. Unlike in a SMT processor, helper execution does not contend for pipeline resources with the main thread, and therefore does not interfere with its execution. However, cores in a HMC are loosely coupled and helper-execution actions such as thread-spawn and synchronization incur inter-processor communication latencies and operating system overhead. Operands needed for helper thread execution are communicated through shared caches, and incur tens of cycles of cache-to-cache transfer latency. In addition, the performance trade-offs of using a smaller core for helper execution are not well understood. It is not clear if small cores can execute helper threads sufficiently in advance to benefit the main thread running on a much powerful core.

In this chapter, we present a hardware/software framework called *core-tethering* to support efficient helper threading on heterogeneous many-cores. *Core-tethering* adds new user mode instructions that provide a co-processor like interface to the helper cores in a HMC, allowing a large core to directly initiate and control helper execution, and to efficiently transfer application context needed for execution to the helper core. Through tighter coupling of cores, *core-tethering* overcomes the

latency overheads of accessing small cores for helper execution. In the next chapter, we present our evaluation of helper thread prefetching on heterogeneous many cores using trace based simulation of a set of memory intensive programs chosen from standard benchmark suites.

The rest of the chapter is organized as follows: Section 3.1 presents the baseline HMC architecture and the methodology we adopt to construct helper threads. Section 3.2 presents details of our *core-tethering* framework, and Section 3.3 illustrates how *core-tethering* instructions are used to spawn helper execution and to synchronize between the main and helper threads.

3.1 Baseline architecture and Helper construction methodology

The baseline HMC architecture used in this study is a single-ISA shared memory many core similar to the ARM big.LITTLE template [Gre11]. Two sets of cores, large cores optimized for performance and small cores optimized for energy efficiency, are connected to a shared L3 cache through an interconnect fabric. Large cores are similar in design to a traditional high performance out-of-order superscalar processor, while small cores are reduced issue width processor with smaller reorder buffer and cache sizes. Each of the cores have private L1 and L2 caches, with the larger cores having a much larger L2 cache than the small cores. Table 4.2 from Chapter 4 provides details of the parameters we choose for the large and small cores in our experimental study.

In the helper threading mode, a sequential application utilizes two cores in a HMC - the main application thread runs on the complex/large core, while a simple/small core executes a pre-compute thread that generates addresses and prefetches cache lines on behalf of the main thread. We assume software based triggering for helper threads, i.e the main thread running on the large core explicitly initiates the execution of a statically constructed helper thread. Our focus in this thesis is on the interface between the cores in the HMC to reduce the latency of accessing the small cores from the large core to initiate and control helper execution, and to enable fast communication between the cores. The cores themselves remain unmodified, expect for the changes required to implement the *core-tethering* framework introduced in Section 3.2.

We based our pre-compute thread construction methodology on static analysis of a compiler generated binary, and use profile information to identify instructions

targeted for prefetching. First the application is profiled and critical load/store instructions and target loops are identified. In the next step, static analysis is performed offline on binary code directly, to compute data-flow information and to track address computations. For one or more target instructions, a back-slice or trace generator [KC11] is extracted, containing all instructions that participate in address computations and forms the body of the helper code.

Static analysis starts by building a control-flow graph (CFG), which is used to perform data-analysis. In our implementation, instructions are parsed to extract used and defined registers, and a standard algorithm is applied to build a Static Single Assignment (SSA) form of the program [CFR+91]. To reconstruct helper code, instructions participating in address computations, as well as instructions contributing to all branches that address computation is control-dependent on, are marked using the CFG and dominator information. Slicing starts on selected delinquent loads, and continues backward until reaching the boundaries of the region of interest (typically, a loop or a function). Use-def links that cross this boundary identify exactly the set of *live-ins*, i.e. register values that are used as input to the trace generator, and therefore need to be provided by the main program during the helper initiation phase.

3.2 Core tethering

When porting helper threads to a loosely coupled system such as a HMC, helper-actions, namely thread-spawn and synchronization, incur inter-processor communication latencies and operating system overhead which can limit the utility of helper generated prefetches. Consequently, architects have explored hardware support to overcome these overheads: Hall et al. [HLSS13] propose hardware based spawning of the helper thread using a *branch to assist thread* instruction to reduce thread-spawn overheads, Gschwind et al. [GOSS13] propose direct hardware-assisted communication between threads to bypass the memory hierarchy and enable fast communication of values between main and helper threads, Lee et al. [LJLS09] use special synchronization registers in the memory processor running helper threads, which are accessed as co-processor registers to communicate synchronization variables between main and helper with low overhead.

In this thesis, we present core-tethering as a hardware/software framework that enables tighter coupling of large and small cores in a HMC. The key elements of the framework are a set of architectural buffers that hold parameters required

for helper execution and new instructions that simplify interaction between main core and helper cores. Together, they provide a co-processor like interface to smaller cores in the HMC and reduce latency overhead of accessing them for helper execution using (1) a hardware based mechanism for initiating helper threads and (2) a direct communication mechanism between cores that bypasses the memory hierarchy (illustrated in Figure 3.1).

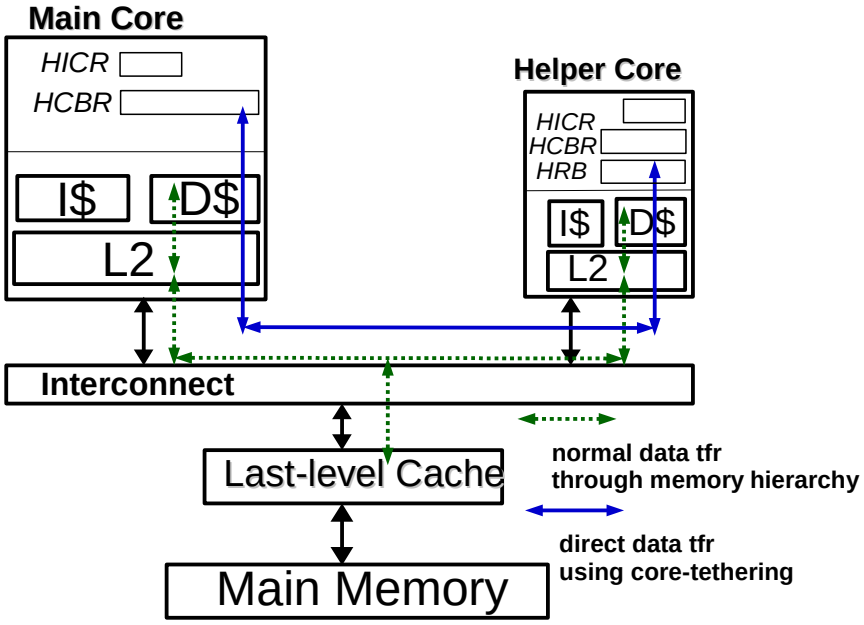


Figure 3.1: Illustration of direct communication between cores.

An architectural buffer called *Helper Control Block Register (HCBR)* holds the requested helper action and application context needed to execute the helper action. HCBR is a 64 byte structure, and is divided into 8 fields of 8 bytes each, *field-0* to *field-7*. The contents of the HCBR is called a *Helper Control Block (HCB)*. The first field of the HCB, *field-0* (8 bytes), is used to specify the required helper-action. The next 7 fields of the HCB are used to transfer application context needed for helper execution or helper synchronization from the main core to the tethered helper core. Depending on the helper action requested, some or all of the fields *field-1* to *field-7* can be valid. A new architectural register called *Helper Iteration Count Register (HICR)* helps to track the progress in the main thread and helper thread, and is used during helper synchronization.

Three new user mode instructions are added to allow a complex core to di-

rectly initiate and control helper execution, and to efficiently transfer application context (variables) needed for helper execution and helper synchronization between main core and helper cores.

1. *ISHCB* - initiate store helper control block
2. *WRHCBR* - write helper control block register
3. *RDHCBR* - read helper control block register

WRHCBR and *RDHCBR* support write and read operations to HCBR register. WRHCBR copies values of argument registers to a field within the HCBR, an immediate argument specifies the index of the field (*field-0* to *field-7*). RDHCBR copies the value of a HCBR field (specified by an immediate argument) to an argument register.

ISHCB initiates and controls helper execution on small cores. In our framework, the OS monitors utilization of small cores and provides to the large core identity of an idle core for use as tethered helper core. If no such idle core is available the ISHCB request is dropped. When the large core executes *ISHCB*, contents of its HCBR (i.e the Helper Control Block(HCB)) are transferred to the tethered helper(small) core using the interconnect network. Execution of the ISHCB instruction initiates a helper action on the tethered helper core. The HCB specifies the required helper-action and (optionally) values of input registers required to perform the helper action. The first 4 bits of field-0 specifies the type of helper action requested, and the next 60 bits are used for additional information related to the requested helper action (Table 3.1). ISHCB instructions are treated like stores, and write the HCB to tethered helper-cores only when they are non-speculative. Thereby no helper threads get spawned spuriously due to speculative execution in the main core.

A hardware structure called Helper Request Buffer (HRB), located on the small core, acts as a holding buffer and receives from the interconnect ISHCB requests sent by the large core. HRB has an associated state machine logic that fills the HCBR of the small core with the received HCB and also interprets the fields as follows. When the helper-action field is set to SPAWN, it retargets the PC of the small core to the target-pc specified by the ISHCB request and starts execution of the helper function. When the helper-action field is set to KILL, execution of a helper function currently running on the small core is terminated and the small core is now ready to accept another SPAWN request. When the helper-action field is set to SYNC, the state machine only fills the HCBR of the small core.

Table 3.1 different helper actions triggered by ISHCB instruction

Helper action	Contents of HCB *field-0*
SPAWN : start execution of a helper codelet	bits 0-3 = 0000
	bits 4-7: number of input required as live-ins, determines which HCB *fields1-7* are valid
	bits 0-63: specifies the instruction address of the helper codelet to execute
KILL : stop helper execution	bits 0-3 = 0001
SYNC : synchronize execution of main and helper threads	bits 0-3 = 0010
	bits 4-7: number of inputs needed for synchronization, determines which HCB *fields1-7* are valid

Support for similar instructions exist in current microprocessors. For example, IBM PowerPC-A2 processor includes an instruction called *icswx* (Initiate Coprocessor Store Word) that copies an aligned 64-byte co-processor request block (CRB) to the targeted accelerator [KHL+12]. CRB contains all parameters needed for execution of accelerator functions such as the command to execute, and input and output data addresses.

3.3 Helper threading with Core-tethering instructions

Figure 3.2 Figure 3.3 show a function *refresh_potential* from the benchmark *mcf* selected for helper prefetching and its corresponding helper thread executing pre-compute code. For illustration purposes we list code at the source level, however actual static analysis and helper construction are carried out at the binary level in this study. In the helper thread, delinquent memory accesses are identified and converted to prefetches. Variables *node* and *tmp* are privatized so that helper thread can run independently (line 9).

Figure 3.2 shows original function instrumented with core tethering instructions to initiate and control helper thread execution. To initiate helper execution

```
1  long refresh_potential( network_t *net )
2  {
3      node_t *node, *tmp;
4      node_t *root = net->nodes;
5      long checksum = 0;
6      root->potential = (cost_t) MAX_ART_COST;
7      tmp = node = root->child;
8
9      CT_CLEAR_HICR();
10     CT_WRITE_HCBR(SPAWN,&refresh_potential_helper,root);
11     CT_ISHCB();
12
13     while( node != root ) {
14       while (node) {
15           if (node->orientation == UP )
16               node->potential = node->basic_arc->cost + node->pred->potential;
17           else {/* == DOWN */
18               node->potential = node->pred->potential - node->basic_arc->cost;
19               checksum++;
20           }
21           tmp = node;
22           node = node->child;
23       }
24       node = tmp;
25       while( node->pred) {
26           tmp = node->sibling;
27           if( tmp ) {
28               node = tmp;
29               break;
30           }
31           else node = node->pred;
32       }
33       CT_SYNC_MAIN (node);
34     }
35
36     CT_WRITE_HCBR(KILL);
37     CT_ISHCB();
38
39     return checksum;
40 }
41
42 CT_SYNC_MAIN ( node_t * node)
43 {
44   INCR_HICR();
45   if (HICR % LOOP_SYNC_INTERVAL == 0) {
46     CT_WRITE_HCBR( SYNC, HICR, node);
47     CT_ISHCB();
48   }
49 }
```

Figure 3.2: Illustration of helper threading using core-tethering instructions for the benchmark *mcf* - target function *refresh_potential* in the main thread

on the small core, main thread sets up a HRB with requested action field set to *SPAWN* and the identifier (PC) of the helper code (line 10). Other fields in the HRB are used to copy the values of operands to be used in helper code execution. To synchronize with the helper thread, the main thread sets up the HRB with requested action field set to *SYNC*(line 46) and copies the values of *HICR* register and live-in variables needed to catch up with the main thread. To kill execution of the helper thread, the main thread sets up the HRB with requested action field set to *KILL* (line 33).

Figure 3.3 shows the helper thread version of the function using core tethering

```
1  void refresh_potential_helper( )
2  {
3      node_t *node_pvt, *tmp_pvt;
4      node_t *root;
5
6      CT_CLEAR_HICR();
7      CT_READ_HCBR(&root);
8
9      tmp_pvt = node_pvt = root->child;
10
11     while( node_pvt != root ) {
12       while(node_pvt) {
13         if (node->orientation == UP) {
14           prefetch(node_pvt);
15           prefetch(node_pvt->pred);
16           prefetch(node_pvt->basic_arc);
17         } else {
18           prefetch(node_pvt);
19           prefetch(node_pvt->pred);
20           prefetch(node_pvt->basic_arc);
21         }
22         tmp_pvt = node_pvt;
23         node_pvt = node_pvt->child;
24       }
25       node_pvt = tmp_pvt;
26       while( node_pvt->pred) {
27           tmp_pvt = node_pvt->sibling;
28           if( tmp_pvt ) {
29               node_pvt = tmp_pvt;
30               break;
31           }
32           else  node_pvt = node_pvt->pred;
33       }
34       CT_SYNC_HELPER(node_pvt);
35     }
36 }
37
38
39 CT_SYNC_HELPER( node_t * node_pvt)
40 {
41   INCR_HICR( );
42   if (HICR % LOOP_SYNC_INTERVAL == 0) {
43     CT_READ_HCBR(&hicr_main, &node_main);
44     if ( HICR -- hicr_main < MAX_DIST) {
45       HICR = hicr_main;
46       node_pvt = node_main; //catch up with main
47     }
48   }
49 }
```

Figure 3.3: Illustration of helper threading using core-tethering instructions for the benchmark *mcf* - helper code for function *refresh_potential*

instructions to interact with the main thread. At the beginning of execution, the helper thread reads the values of live-in variable from the HCB transfered by main thread (during *SPAWN*) (line 7) and initializes its local variables. To synchronize with the main thread, helper reads the HICR transfered by the main thread (line 44) and compares execution of the helper thread with the progress of the main thread. If the helper thread is running behind the main thread, it synchronizes with the application thread by skipping over the iterations that the application thread has already executed. Helper execution restarts at the current iteration of the application thread by copying over value of the *HICR* register and the live-in

variable.

3.4 Summary

Helper threading appears as a promising technique to improve sequential performance on HMC architectures by utilizing the otherwise idle small cores to execute precompute slices that can prefetch for targeted loads/stores in the main thread. However, the latency overheads of accessing the small cores in the loosely coupled HMCs make it difficult to fully realize their potential as helper cores. In this chapter, we presented a hardware/software framework called *core-tethering* that enables tighter coupling between cores in a HMC to reduce latency overheads and support efficient helper threading on heterogeneous many-cores. *Core-tethering* adds new user mode instructions that provide a co-processor like interface to the helper cores in a HMC, allowing a large core to directly initiate and control helper execution, and to efficiently transfer application context needed for execution to the helper core. The next chapter presents an evaluation of suitablility of small cores for helper thread prefetching using trace based simulation of a set of memory intensive programs.

Chapter 4

Evaluation of Helper Threading on Heterogeneous Many Cores

Executing the helper thread on the small cores in the HMC provides a power-efficient way to utilize the otherwise idle cores to improve sequential performance on the large cores. The *core-tethering* framework presented in the previous chapter overcomes latency overheads of helper execution in a HMC by providing direct access to the small cores so that a large core can launch and control helper execution. Due to the performance disparity between cores, small cores may not be able to execute helper threads sufficiently in advance to generate prefetches in time for the benefit of the main thread running on a much powerful core. Additionally, even on a successful in-time prefetch, the main thread still incurs additional access penalty since prefetches are limited to the shared L3 cache. Thus the performance tradeoffs of using small cores for helper threading in a HMC are not clearly understood.

In this chapter we evaluate helper thread prefetching on heterogeneous many cores using trace based simulation on a set of memory intensive programs chosen from standard benchmark suites. Section 4.1 explains the trace based methodology we adopt to simulate execution of helper threads on HMCs, including details of the benchmark programs used in evaluation. To study the suitability of small cores to execute helper threads, we experiment with two variants of small and large cores by essentially varying the size and width of the out-of-order engine. Section 4.2 presents experimental results on the performance of helper prefetching on small cores.

4.1 Evaluation Methodology

We use trace based simulation to model and evaluate helper threads running on small cores in a HMC. Our evaluation framework is comprised of three steps - static analysis and helper construction, tracing and instruction stream generation, and trace driven simulation. A trace reader utilizes information from the helper construction step to generate instruction streams for main and helper cores for execution on a trace driven simulator (Figure 4.1).

For the purpose of simulation, we isolate the intrinsic pre-compute slice of the helper from helper-control operations. Instructions in the pre-compute slice are executed on the helper core and participate in address computation for prefetching. Helper-control operations - helper-spawn, helper-kill and synchronization - are a set of instructions that are executed by main and helpers core to control execution of the pre-compute slice. We insert helper-control operations into the instruction streams based on information provided by the helper construction step. The result of execution of some helper-control operations can depend on state of execution at run-time. For example, outcome of synchronization operation which decides if the helper is sufficiently ahead, depends on the value of HICR register which tracks progress of execution in main and helper cores. Helper-operations are handled in a dynamic component in the simulator, similar to the methodology proposed by Rico et al. [RDC+11] for trace based simulation of multi-threaded applications. The simulator reads instructions from the instruction stream sequentially and executes them till it reaches a helper-operation. The trace driven simulator exposes its architectural execution state (specifically values of HICR register) to a dynamic component in the simulator, which then correctly handles execution of helper actions. Using this runtime information, helper operations carry out functionality depending on the current architectural state (as if executed dynamically) and their timing effects are simulated.

4.1.1 Static analysis and helper construction

The goal of helper construction step is to generate a list of instructions to be included in the helper. Using the methodology detailed in Section 3.1, instructions to be included in the pre-compute slice are determined and all other unnecessary instructions are eliminated. Instructions in the helper are not allowed to change execution state of the main thread, so store instructions targeted in the helper thread are either converted to prefetches, or localized to a separate

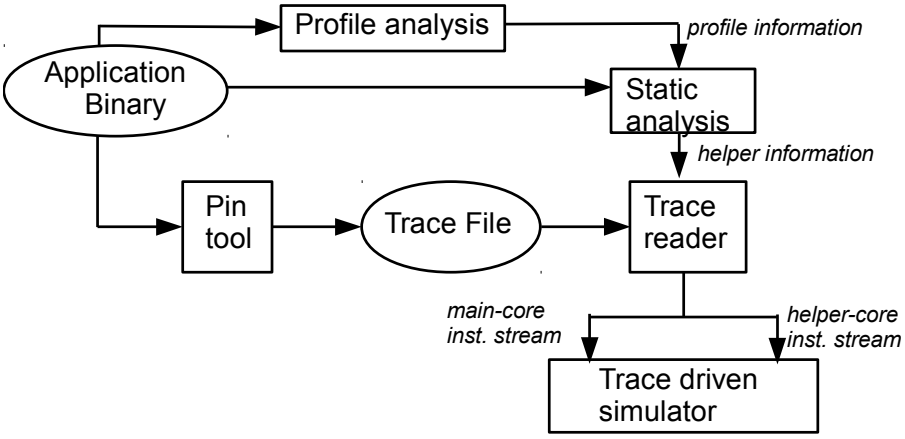


Figure 4.1: Evaluation Framework.

memory location within the helper. Load instructions whose load values are not used later in the slice are converted to prefetch instructions. During static analysis, helper spawn point at entry of the target loop, synchronization point at end of a loop iteration, and the live-in registers (values needed by the pre-compute slice but not produced within the slice) are identified. Helper related information determined by static analysis is passed-on to the trace reader for generating and instrumenting main and helper instruction streams.

4.1.2 Tracing and instruction stream generation

The goal of instruction stream generation step is to generate instruction streams for main and helper cores for simulation in the trace-based simulation engine. The target binary is traced using a Pintool [LCM+05a] to generate an application trace. A trace reader reads this application trace-file at simulation time and generates instruction streams for main and helper cores utilizing information generated by the helper construction step. In case of the main core, instruction stream consists of all instructions from the original application trace. The instruction stream for the helper consists of the pre-compute slice identified during helper construction step. Once unwanted instructions are eliminated from the helper instruction stream, prefetch conversion is applied to loads/stores identified during the helper construction step.

Both instruction streams are instrumented with helper-operations such as *helper-spawn*, *helper-kill*, and *synchronization* at the spawn and synchronization

points identified by the helper construction step.

1. *helper-spawn* is inserted into main-core instruction stream at the spawn point. It requests initiation of a helper thread on the helper core, and specifies the helper function to be executed, and the list of live-in variables (registers) to be transferred to the helper core.
2. *helper-kill* is inserted into main-core instruction stream at the end of the helper-target region. It requests termination of the helper thread currently running on the helper core.
3. *sync-main* is inserted into main-core instruction stream at the synchronization point. It communicates progress of main thread execution to the helper core, along with live-in variables needed for helper to synchronize its execution with the main thread.
4. *sync-helper* is inserted into helper-core instruction stream at the synchronization point. It compares progress of helper execution with main and determines a synchronization response.

4.1.3 Trace driven multi-core simulator

The instruction streams for main and helper cores are simulated using a trace based simulator. We model different core parameters for large and small cores as shown in Table 4.2. The simulator models cycle by cycle execution of main and helper cores (instruction pipeline and private caches), a shared L3 cache and a fixed latency main memory. Since the simulator is trace driven, on a branch mis-prediction, appropriate penalties are modeled but wrong path effects are not simulated. Bus contention and bandwidth are modeled at the shared cache interface and memory interface.

The application and helper threads share data, and caches have to be maintained coherent to ensure a read to shared memory space returns the most recent write. Since we localize writes to shared data in the helper thread and helper synchronization uses ISA support, coherence overhead on helper threading is not significant. We model an inclusive shared L3 cache, all data stored in each individual L1 and L2 cache is duplicated in the L3. L3 cache has additional flag bits (core-valid) which track if cache lines may be present in a particular core. The inclusive L3 cache can respond to requests for shared lines without snooping other cores. Lines in L3 that are exclusive to a core (only one core valid bit set) may have been modified in a higher level cache, and require a snoop before L3

can respond. Similarly a snoop is required for modified cache lines if data is still present in the higher level. We model an additional latency penalty of 11 cycles for snoop requests which increases L3 latency to 40 cycles.

4.1.4 Benchmarks

Helper threading improves performance only when applications are memory intensive. For our evaluation, we chose applications that miss significantly in the last level cache (L3), and thus spend a considerable portion of their execution time waiting for memory. We used programs from standard benchmark suites (spec2000, spec2006, NAS, minebench, splash2x), and a software utility that can read performance monitoring counters [1] to analyse application execution on a workstation with Intel Nehalem processor to measure L3 miss rates. Applications incurring more than 10 MPKI (misses per kilo-instruction) are included in our benchmark set (Table 4.1). Different input sets were used for profiling and simulation apart from two applications, *radix* and *svm*. To identify delinquent instructions and construct the pre-compute thread, spec2006 and spec2000 programs were profiled with train inputs, and NAS programs with class-W inputs. Each benchmark is simulated for 500 million instructions after fast-forwarding beyond application initialization phase, 50B (Billion) instructions for spec2000 and spec2006, 25B for NAS, 40B for radix and 10B for svm-rfe. As illustrated in Table 4.1, on this class of applications most code is encapsulated in only a few loops, and the amount of instructions that can be removed from the original loop body for execution in the helper is quite substantial.

4.1.5 Modeling Large and Small Cores

We model large and small cores executing the same ISA. Our large core parameters are similar to a traditional high performance out-of-order superscalar processor, and feature a highly pipelined design and a large reorder buffer. Pipeline structures such as schedulers and instruction buffers are optimistically sized to limit resource stalls. An aggressive branch predictor [SM06] and an oracle dependence predictor minimize mis-prediction stalls related to branch and memory speculation respectively. A hardware prefetcher based on stream buffers [FCJV97] that can prefetch directly to the L1 cache for upto eight streams concurrently is included in the large cores. We modeled 8 stream buffers of 4 entries each, and

1. http://tiptop.gforge.inria.fr/

Table 4.1 Memory intensive applications in the benchmark set

Application name	Benchmark Suite, input	Helper		
		# target loops	% inst covered by target loops	% inst in helper
CG	NAS, B	2	99	30.5
MG	NAS, B	1	99	6.4
lbm	spec2006, ref	1	89	7.2
libquantum	spec2006, ref	2	98	24.5
mcf	spec2006, ref	2	59	55
milc	spec2006, ref	5	96	5
equake	spec2000, ref	3	90	10.2
radix	splash2x, native	1	92	11.5
svm-rfe	minebench	1	99	16.8

a 256 entry fully associative stride table that uses the double delta mechanism (check for three consecutive accesses with same stride) to identify prefetchable access patterns. We model a branch mispredict penalty of 12 cycles [2] for both large and small cores. The small core is a reduced issue width processor with smaller queue and cache sizes, a smaller branch prediction table and no complex performance enhancing features such as hardware prefetching or memory dependence prediction.

With respect to generating pre-compute code for helper threading, behavior of software prefetch instruction is important. Software prefetches can be dropped (turned into nops dynamically) when there is contention for miss-handling resources. To get around this, previous studies have used normal loads to implement prefetch for addresses generated by the helper [KST11]. We found this to be particularly inefficient when used with small cores that have limited ROB sizes. Normal loads implementing a prefetch block retirement when they are the oldest instruction in the pipeline and can lead to unnecessary ROB-full related stalls. Instead, we use a variant where software prefetch instructions wait until miss-handling resources are available instead of being turned into a nop. Software prefetch instructions complete execution without blocking retirement after a MSHR is allocated (or in case of cache-hit). The variant of software prefetch

2. mispredict penalty is comparable to that of a bob-cat class core [BCD+11] for the small-core, but is aggressive for the wider large-core

Table 4.2 Large Core and Small Core parameters

	LargeCore-1 (***LC1***)	SmallCore-2 (***SC2***)
Clock Frequency	3Ghz	3Ghz
decode/rename	4 (1 complex + 3 simple inst)	1
dispatch	6 uops	2uops
retirement	6 uops	2uops
structure sizes - ROB	128 uops	64 uops
load/store	64/32	16/8
post-retire store queue	32uop	8uop
schedulers - int	4 of 16uop each	2 of 8uop each
fp/sse	2 of 16uop each	1 of 8uop
execution - int	4 (1mul, 1div)	2 (1mul, 1div)
fp/sse	2 (1 div)	1
loads/stores	2	1
predictors - branch predictors	64Kbit-TAGE/18Kbit-ITTAGE [SM06]	16Kbit TAGE
br. misp. latency	12 cycles minimum	12 cycles minimum
mem dep pred	oracle prediction	none
hw prefetcher	directly to L1, stream-buffer based [FCJV97]	none
cache - line size	64b	
level-1 Icache	32kb/8way, upto 6 outstanding misses	8 kb, 2 way
level-1 Dcache	32 kb/8way, 2 cycle latency, 16 MSHRS	8 kb, 4 way, 2 cycle latency, 8 MSHRS
level-2 cache	512kb/8way, 11 cycle latency	64kb, 4 way, 11 cycle latency
level-3 cache	4 mb, 16 way, 29 cycle latency	
page size	4mb	
memory	210 cycles, 16 bytes/cycle bandwidth	

we model is similar to that available on UltraSPARC IV+ [Gre04].

To evaluate helper threading implementations on HMCs, we model two variants each for large and small cores by essentially varying size and width of the out-of-order engine (ROB, instruction queues and execution resources), and a latency of 11 cycles for transferring the HCB between cores. LargeCore-1 (*LC1*)

models a 6-issue processor with ROB size *128*, and SmallCore-2 (*SC2*) models a 2-issue processor with *64* ROB entries, similar to a bobcat-class core (ROB size *54*) [BCD+11]. LargeCore-2 (*LC2*) models a much larger processor with twice the ROB size of *LC1*, while SmallCore-1 (*SC1*) models an even smaller processor with half the ROB size and execution resources of *SC2*. Table 4.2 shows the parameters we modeled for *LC1* and *SC2*, the base large and small core respectively. Figure 4.2 shows the relative performance difference between small cores and large cores (normalized to small core SC2) on applications in our benchmark set. LC1 and LC2 are on average 3.1X and 3.7X faster than SC2 respectively, while SC1 is 0.7X slower than SC2. On applications with sequential access patterns, the huge performance advantage for large cores can be mostly attributed to the hardware prefetcher and larger instruction window sizes.

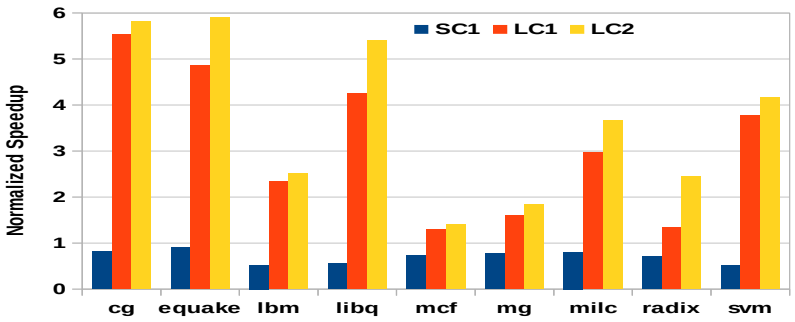


Figure 4.2: Comparison of small and large cores (Normalized to ***SC2***)

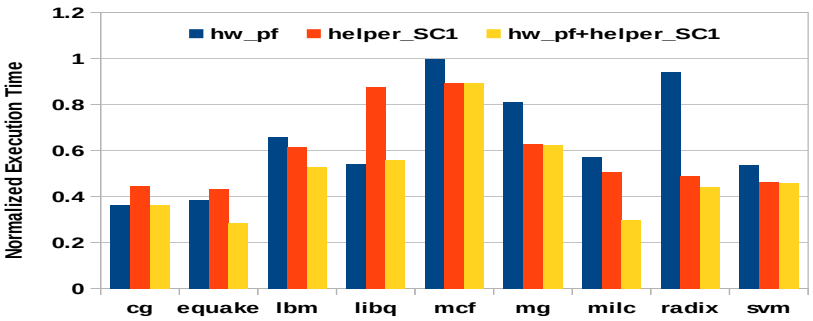
4.2 Experimental Results

4.2.1 Performance of Helper Prefetching on Small Cores

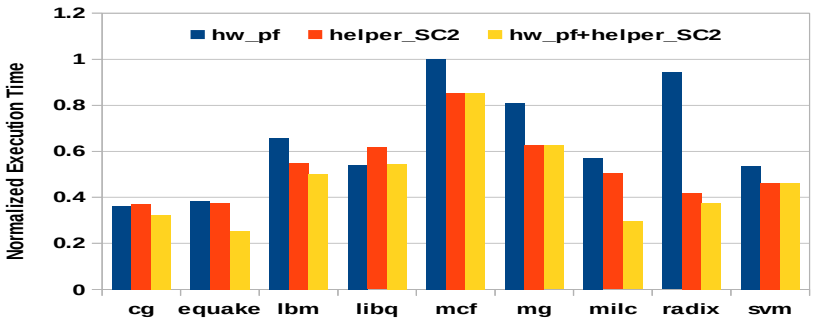
For the large cores LC1 and LC2, Figures 4.3 and 4.4 compare execution time (500M instructions) for the following configurations. Execution time for each configuration is normalized to the *base* configuration without helper-prefetching and hardware-prefetching.

1. *hw_pf* prefetching using a processor side hardware prefetcher to the L1 cache.
2. *helper_X* helper prefetching using one of the small cores SC1 or SC2, or an equally large core as the helper core.

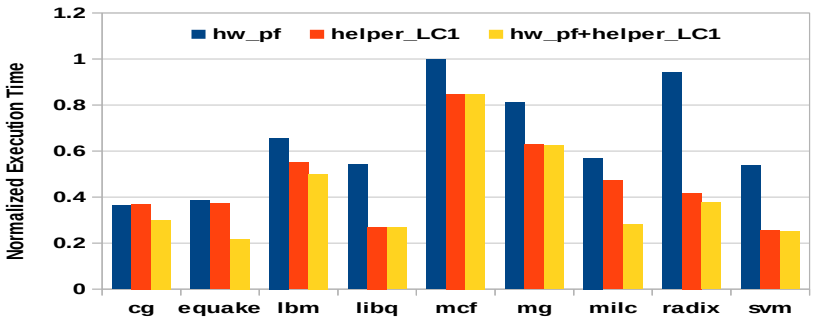
3. *hw_pf+helper_X* combined prefetching using both processor side hardware prefetcher and a helper core.



(i) helper core ***SC1***



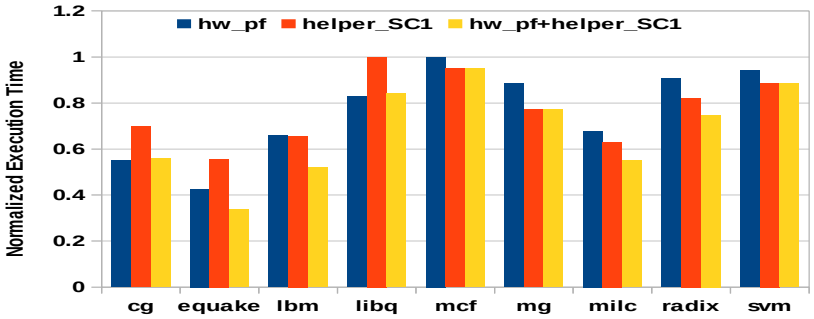
(ii) helper core ***SC2***



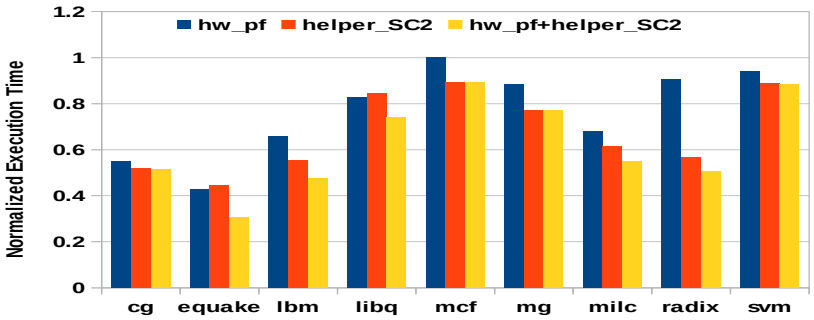
(iii) helper core ***LC1***

Figure 4.3: Helper Threads on Heterogeneous Many Cores - main core ***LC1***

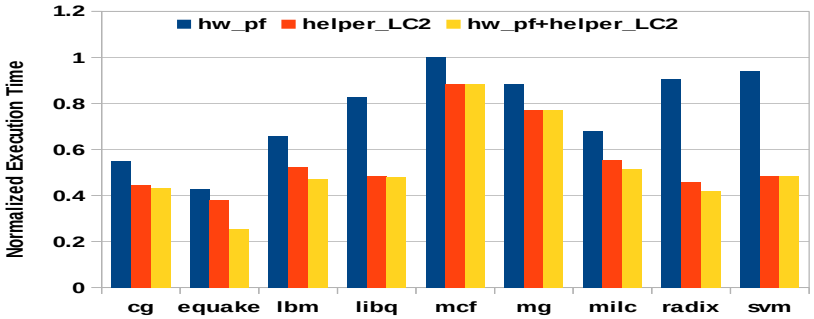
For applications that have predominantly sequential access patterns, i.e. cg, equake, lbm, libquantum, mg, milc and svm, the stream buffer based hardware



(i) helper core ***SC1***



(ii) helper core ***SC2***



(iii) helper core ***LC2***

Figure 4.4: Helper Threads on Heterogeneous Many Cores - main core ***LC2***

prefetcher(hw_pf) performs effectively. But, for applications that have irregular access patterns such as mcf, radix, hw_pf is unable to track cache miss patterns.

Helper prefetching is able to prefetch for applications with both sequential and irregular access patterns. For mcf, which is a pointer chasing application,

helper threading applied to LC1 improves performance by 1.12X(1.19X) when using SC1(SC2) (Figure 4.3). Even higher gains are realized for radix where one of the delinquent loads has a sequential access pattern and the other has irregular access pattern.

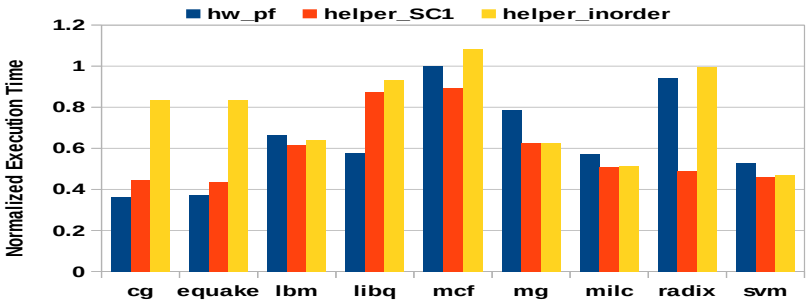
On analysing the results for helper prefetching using the smaller core SC1 for applications cg, equake and libquantum, benefits of helper generated prefetches are smaller than benefits from the hardware prefetcher. We find that despite having a sequential access pattern for delinquent instructions, these applications use the load-value of one or more delinquent instructions later in the pre-compute thread (either for address calculation or for control flow). Since prefetch conversion cannot be applied, these instruction utilize a normal load which ends up stalling helper execution when it reaches the head of the reorder buffer; whereas the hardware prefetcher is able to track sequential access patterns and predictively generate prefetch requests. This impact is more pronounced when the number of ROB entries is small, as in the case with SC1. With an increase in ROB size with SC2, helper threading is able to deliver performance benefits close to or exceeding the hardware prefetcher. In the case of milc, we find that even when helper prefetching covers most of the delinquent misses, it still underperforms hw_pf by a small margin. This can be attributed to two reasons, the hardware prefetcher tracks sequential accesses outside targeted loops whereas helper prefetching is restricted to target loops, also the hardware prefetcher is able to prefetch to the L1 cache directly, hiding more of the cache miss latency than helper prefetches to the shared L3 cache.

Combined with processor side hw_pf, helper prefetching gives better gains than either of them alone. For the LC1 configuration, helper prefetching combined with hw_pf gives an average additional gain (over helper prefetching alone) of 35.6% for SC1 and 44% for SC2. We see a synergistic effect for patterns with sequential access patterns where the helper thread runs ahead and prefetches to the shared L3 cache, and processor side prefetcher fetches the cache line to the L1 cache completely hiding memory access latency.

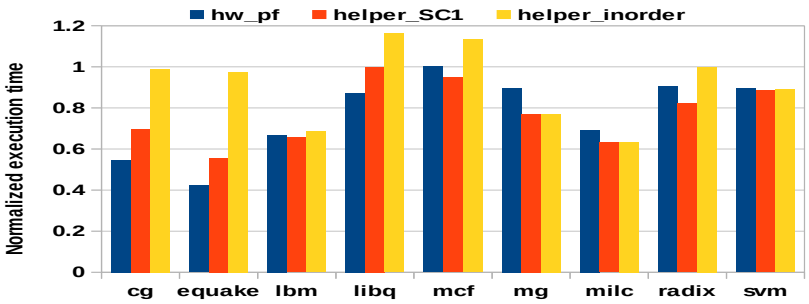
For the large core LC2 that features a larger instruction window, we find that helper threading running on the small core SC2 outperforms hardware prefetching alone (Figure 4.4(ii)). When combined with processor side hardware prefetching, average additional gains over hardware prefetching are 13.6% and 26% for SC1 and SC2 respectively. The larger instruction window on LC2 results in fewer ROB full related stalls, consequently gains due to helper prefetching are smaller compared to the LC1 configuration.

We also compare efficiency of helper prefetching using small cores with a configuration that uses an equivalent large core to run helper threads. In our benchmark set, excepting two applications, libquantum and svm, SC2 provides a good tradeoff against an equivalent large core. For the LC1 configuration, helper prefetching using the small core SC2 give gains within 7% of helper prefetching using an equivalent large core. For the LC2 configuration, much larger ROB size widens performance gap between an equivalent large core and SC2 to as much as 19%. Libquantum and svm feature tight loops, and larger instruction window in LC1/LC2 enables the helper thread to deliver much larger benefits when run on an equivalent large core. For these applications, helper prefetching on SC2 delivers only half the gains of an equivalent large core (LC1/LC2).

4.2.2 Performance of Helper Prefetching on In-order Cores



(i) main core ***LC1***



(ii) main core ***LC2***

Figure 4.5: Helper Threads running on in-order cores

In-order cores are smaller in size and consume lesser energy, and are the preferred design option for many embedded applications. Even when in-order cores are enhanced to issue multiple instructions per cycle for improved performance, such as the Cortex A8 from ARM [Wil] and Xeon-Phi from Intel [CE12], they provide a complexity effective design point with a balance of performance and area/power efficiency. For example, three Xeon-phi like cores can be accommodated within the same area of one high performance Xeon processor [HBT13]. Additionally, in combination with Simultaneous Multithreading (SMT), in-order cores can tolerate memory latency by keeping the pipeline busy with instructions from multiple threads. Consequently in-order cores are very popular in processors such as the Niagara from SUN microsystems [KAO05], Xeon-Phi from Intel [CE12] targeted at throughput oriented applications.

We evaluate the use of in-order 2 issue cores as helper cores in comparison to the small core SC1 (32 entry ROB, 2 issue out of order core) (Figure 4.5). The trend on in-order processors is quite similar to that on the smaller core SC1, any dependence within the helper thread itself results in insufficient advance for the helper over the main thread. We find that applications with regular access behavior where prefetch conversion in the helper is completely successful, i.e. lbm, mg, milc, svm, an inorder helper core performs comparably to an out of order core with equivalent issue width (SC1).

In applications like cg, equake and libquantum prefetch conversion cannot be applied completely, and consequently the in-order core stalls waiting on outstanding memory requests. This is similarly the case on irregular applications such as mcf and radix. In all these applications, the in-order helper core is quite ineffective in comparison to SC1.

4.2.3 An alternative design point for memory intensive workloads

Performance on memory intensive applications is largely limited by their MLP (memory level parallelism) i.e. number of memory accesses that can be sustained in parallel [Gle98]. On modern super-scalar machines, independent instructions can be processed till the instruction window is full. When a memory access missing in the on-chip caches reaches the head of the instructions window, it stall waiting for the memory response and blocks the following instructions from committing. Growing the processor size on large cores by increasing the instruction window size and adding more miss handling resources (MSHRs) renders the

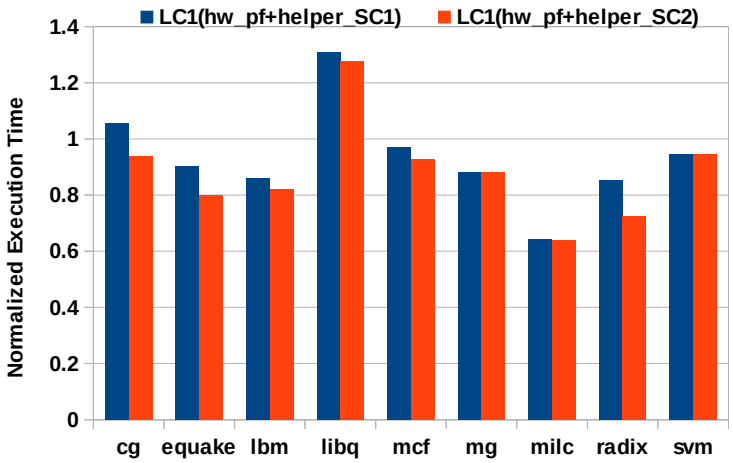


Figure 4.6: Helper threading for LC1 Vs larger LC2

machine inherently tolerant to memory latency. The out of order engine can now select independent instructions (not dependent on another memory request) from the large window and send out more requests to memory in parallel, thereby increasing sequential performance. However, this performance is realized at increased hardware cost, since growing the instruction window on wide-issue cores leads to a significant increase in hardware complexity of associated structures such as register files and instruction queues [ONH+96].

We compared performance of the large core LC1 accelerated by helper prefetching, against a 2X larger core LC2 (Figure 4.6). On the benchmark set we evaluated, we find using helper prefetching on small cores SC1(SC2) improved performance of LC1 by 6.9%(13.2%) when compared to the larger core LC2 (with hardware prefetcher enabled). We find that helper prefetching on small cores used in conjunction with hardware prefetching, can provide an alternate design point to growing the instruction window size for achieving higher sequential performance on memory intensive applications.

4.3 Summary

Small cores in a HMC provide an opportunity to use them as low power helpers to accelerate single thread performance. This is particularly attractive for workloads that feature significant sequential code sections, and cannot otherwise benefit from increasing the core count. In this chapter, we used a trace based

simulation methodology to evaluate the utility of small cores to run helpers that prefetch for a larger, much powerful core on a set of memory intensive programs chosen from the standard benchmark suites.

We experiment with two variants of small and large cores by essentially varying the size and width of the out-of-order engine. From our simulations, we find that helper threads running on a moderately sized small core(ROB size 64, 2-issue) can significantly accelerate much larger(6-issue) cores compared to using a hardware prefetcher alone for improving performance on programs that extensively access memory. Also, these small cores provide a good trade-off against using an equivalent large core to run helper threads. In our benchmark set, excepting two applications, helper threads on small core show gains within 7%(19%) of helper threads running on an equivalent 2X(4X) larger core. Additionally, helper thread prefetching on small cores when used in conjunction with hardware prefetching, can provide an alternative to growing the core size for higher performance on memory intensive applications.

Chapter 5

Other Contributions

This chapter presents a summary of other contributions, as a second author, to studies in (1) understanding impact of state of the art branch predictors on performance of interpreters and (2) performance estimation of multi-threaded programs on massively parallel many-cores architectures. Material presented here are from publications in CGO 2015 [RSS15](to appear) and WAMCA 2014 workshop [NSS14] respectively.

5.1 Branch Prediction and Performance of Interpreters

Despite the popularity of JITed languages such as Java, interpreters remain widely prevalent for implementing programming languages such as R, Python, Matlab. Interpreters are much easier to develop, maintain, and port applications on new architectures, but this comes at the cost of performance. Even on an efficient implementation of interpreters, there is a 10X slowdown in execution speed in comparison to native code produced in an optimizing compiler [EG03].

The performance overhead of interpreters comes from the execution of the dispatch loop that reads the bytecode, decodes it, and based on the bytecode performs the appropriate action, the selection of the action typically being implemented as a switch statement. Every bytecode requires ten instructions when compiled directly from standard C in comparison to the (typically) single native instruction needed when JIT compiled. Also operands must be accessed from the evaluation stack, while native code would have operands in registers in

Table 5.1 Branch predictor parameters

Parameter	TAGE	ITTAGE 2	ITTAGE 1
min history length	5	2	2
max history length	75	80	80
num tables (N)	5	8	8
num entries table T_0	4096	256	512
num entries tables $T_1 - T_{N-1}$	1024	64	128
storage (kilobytes)	8 KB	6.31 KB	12.62 KB

most cases. On high end processors, wide issue, out of order and high frequency execution backed by large first level caches have largely mitigated most of the overheads in interpreter execution. But the indirect jump that implements the switch statement remains difficult to predict since it has potentially hundreds of targets (one for each bytecode level opcode) [CEG07][EG01], and conventional wisdom considers the performance of branch predictors on this indirect jump as a major bottleneck.

In this study, we revisit previous work on the predictability of the branch instructions in interpreters in three of the latest generation processors from Intel and a state of art indirect predictor ITTAGE [SM06], using interpreters for python, javascript and cli. We find that the accuracy of branch prediction on interpreters has dramatically improved over the three last generation of Intel processors, and on the most recent processor called Haswell the prediction accuracy is such that it can no more be considered as an obstacle for performance.

5.1.1 Experimental Methodology

Interpreters and Benchmarks

We used Python 3.3.2 with Unladen Swallow Benchmark as input, Javascript interpreter from SpiderMonkey 1.8.5 with Google's octane suite and kraken from Mozilla for Javascrip as input , and GCC4CLI with SPEC 2000 benchmarks as input (excepting 176.gcc, 253.perlbmk, 254.gap, 255.vortex, 300.twolf which were not supported by our CLI interpreter). The interpreters are compiled with Intel icc version 13, using flag -xHost that targets the highest ISA and processor available on the compilation host machine. All benchmarks are run to completion, including on the simulator.

Branch Predictors

On existing hardware, branch prediction data is collected from the PMU (per-

formance monitoring unit) on Nehalem (Xeon W3550), Sandy Bridge (Core i7-2620M), and Haswell (Core i7-4770) machines running Linux. Counters for cycles, retired instructions, retired branch instructions, and mispredicted branch instructions are provided in the PMU, Tiptop [Roh11] is used to collect events per process (not machine wide). For the state-of-the-art branch predictor TAGE and ITTAGE [SM06], we used simulation on traces produced by Pin [LCM+05b] to collect data on prediction accuracy. We used two (TAGE+ITTAGE) predictor configurations (Table 5.1) : TAGE1 assumes a 8KB TAGE and a 12.62 KB ITTAGE, TAGE2 assumes a 8KB TAGE and a 6.31 KB ITTAGE.

Comparing data from real hardware and simulations

Since we compare data between real hardware and simulation results, we confirmed that these tools report comparable instruction counts. Pin can only collect user-mode instructions, hence the PMU is configured correspondingly to report user-mode events. We confirmed that the main loop of interpreters is identical on all architectures, and the average variation of the number of executed instructions, due to slightly different releases of the operating system and libraries, is of the order of 1%. Also, PMU and Pin also report counts within 1%.

5.1.2 Experimental Results

For each interpreter, tables 5.2,5.3,5.4 report on the branch prediction accuracy on Nehalem, Sandy Bridge, Haswell, TAGE1 and TAGE2.

1. **Python** Python is implemented in C, and supports roughly 110 bytecodes. The dispatch loop consists of 24 instructions for bytecodes without arguments, and 6 additional instructions to handle an argument. Only one indirect branch is part of the dispatch loop.

   Between 120 to 150 instructions are needed to execute a bytecode, and considering the overhead of the dispatch, about 100 instructions are needed to execute the payload of a bytecode. This rather high number is due to dynamic typing, since even a simple instruction/byte-code such as add must check the types of the arguments (numbers or strings). There is generally a single indirect branch per bytecode.

   In fastpickle and regex, the number of instructions per bytecode is much higher. Also number of indirect branches per bytecode is significantly larger than 1, and correlates with a high number of instructions per bytecode, indicating execution in a dedicated routine.

Table 5.2 Python : branch prediction for Nehalem (Neh.), Sandy Bridge (SB), Haswell (Has.) and ITAGE (IT)

benchmark	MPKI					benchmark	MPKI				
	Neh.	SB	Has.	IT1	IT2		Neh.	SB	Has.	IT1	IT2
call_method	4.8	0.6	0.1	0.067	0.067	meteor_contest	16.7	6.9	5.5	3.507	3.519
call_method_slots	5.9	0.7	0.1	0.068	0.068	nbody	13.8	5.9	2.1	0.700	0.701
call_method_unknown	4.2	0.3	0.1	0.058	0.058	nqueens	16.5	3.9	0.9	0.549	0.549
call_simple	4.1	0.5	0.1	0.086	0.086	pathlib	16.5	4.7	1.2	0.397	0.633
chaos	18.4	4.4	1.8	0.680	2.548	pidigits	1.2	0.5	0.4	0.356	0.363
django_v2	15.9	3.9	1.5	0.529	1.829	raytrace	15.2	3.6	1.8	0.577	1.017
fannkuch	18.4	6.1	0.9	0.578	0.592	regex_compile	15.1	5.3	2.1	1.588	2.257
fastpickle	19	2.6	1.4	2.258	2.290	regex_effbot	6.1	0.1	0.0	0.026	0.027
fastunpickle	16.5	3	1.7	2.365	2.673	regex_v8	9.8	1.1	0.6	0.506	0.534
float	12	3	1.5	0.364	0.365	richards	13.2	5.8	1.8	0.824	1.518
formatted_logging	17.5	5.4	2.8	0.633	4.220	silent_logging	10.3	3.7	0.4	0.035	0.035
go	14	5.2	2.4	1.121	1.979	simple_logging	17.4	5.3	2.8	0.669	4.748
hexiom2	11.9	2.9	0.8	0.563	0.832	telco	15.6	5.7	1.5	1.143	1.150
json_dump_v2	17.2	3.5	0.6	0.827	0.859	unpack_seq	8.9	4.3	2.2	0.056	0.057
json_load	15.7	3.3	2.1	3.074	3.198						
average	12.8	3.5	1.4	0.8	1.3						

Looking at the MPKI for each benchmark and branch predictor in Table 5.2, Sandy Bridge's predictor significantly outperforms Nehalem's, and Haswell and TAGE outperform Sandy Bridge.

2. **Javascript** We used SpiderMonkey which is implemented in C++, and compiled it without JIT support. SpiderMonkey supports roughly 244 bytecodes and the dispatch loop consists of 16 instructions, significantly shorter than Python. Indirect branches come mainly from the switch statement, excepting code-load in octane, and parse-financial and stringify-tinderbox in kraken, which also have a outstanding number of instructions per bytecode. Excluding them, there are on average appoximately 60 instructions per bytecode.

   Table 5.3 reports the MPKI on the javascript interpreter. Similar to Python, Haswell and TAGE consistently outperform Sandy Bridge, which outperforms Nehalem, and with the exception of three outliers, $TAGE{+}ITTAGE$ predicts branches in the interpreter almost perfectly.

3. **CLI** The CLI interpreter is written in standard C, dispatch is implemented using a switch statement, and supports 478 bytecodes. CLI operators are not typed, but the standard requires that types can be statically derived

Table 5.3 Javascript : branch prediction for Nehalem (Neh.), Sandy Bridge (SB), Haswell (Has.) and ITAGE (IT)

benchmark	MPKI					benchmark	MPKI				
kraken	Neh.	SB	Has.	IT1	IT2	octane	Neh.	SB	Has.	IT1	IT2
ai-astar	17.5	6.4	0.0	0.01	0.01	box2d	16.5	9.2	3.2	1.64	2.46
audio-beat-detection	13.4	6.6	1.5	0.14	0.16	code-load	13.0	5.5	5.0	4.44	4.53
audio-dft	14.5	4.3	1.1	0.01	0.01	crypto	15.4	13.3	0.2	0.11	0.2
audio-fft	13.5	6.7	1.3	0.12	0.12	deltablue	11.1	5.7	2.0	0.13	0.44
audio-oscillator	10.5	5.0	1.0	0.01	0.01	earley-boyer	13.8	6.5	1.4	0.48	1.15
imaging-darkroom	14.7	9.3	2.1	0.07	0.08	gbemu	13.0	5.4	1.5	0.37	0.53
imaging-desaturate	7.4	4.1	1.2	0.01	0.01	mandreel	13.9	6.2	2.3	0.74	1.29
imaging-gaussian-blur	15.8	6.6	1.1	0.17	0.17	navier-stokes	12.1	6.1	1.0	0.01	0.01
json-parse-financial	17.3	1.1	1.0	1.76	1.77	pdf	14.1	3.9	0.7	0.37	0.54
json-stringify-tinderbox	12.0	1.3	1.2	1.71	1.71	raytrace	14.7	5.7	2.2	0.99	2.47
crypto-aes	15.0	9.3	1.8	0.29	2.13	regexp	10.7	1.7	1.0	0.85	0.9
crypto-ccm	14.7	9.4	2.9	1.07	1.62	richards	12.4	8.8	1.9	0.42	0.71
crypto-pbkdf2	14.4	9.4	2.3	0.71	1.24	splay	10.8	3.3	1.4	0.79	1.13
crypto-sha256-iterative	13.9	8.5	2.2	0.87	1.01	typescript	14.2	6.5	3.0	2.64	3.86
						zlib	13.5	8.0	2.0	0.46	1.40
average							13.6	6.3	1.7	0.7	1.1

Table 5.4 CLI : branch prediction for Nehalem (Neh.), Sandy Bridge (SB), Haswell (Has.) and TAGE

benchmark	MPKI					
	Neh.	SB	Has.	T1	T2	T3
164.gzip	18.7	14.5	0.5	0.64	1.58	0.62
175.vpr	24.8	21.9	6.3	6.49	13.62	1.08
177.mesa	13.8	11.1	2.6	0.21	0.59	0.20
179.art	7.3	9.3	0.2	0.38	0.38	0.37
181.mcf	17.5	15.1	1.1	1.33	2.09	1.08
183.equake	20.8	19	1.7	0.47	0.68	0.43
186.crafty	21.1	19.2	11.6	11.87	16.31	4.01
188.ammp	19.5	14.4	2.9	0.39	1.14	0.30
197.parser	14.4	10.8	1.4	1.01	2.75	0.70
256.bzip2	17.5	12.7	1.6	1.55	2.15	0.44
average	17.5	14.8	3.0	2.4	4.1	0.9

to prove correctness before execution. The bytecode specializes the operators with computed types to remove type resolution overhead from the interpreter execute loop, and this results in a large number of different bytecodes.

The dispatch loop consists of only seven instructions, this is possible because each opcode is very low level (strongly typed, and derived from C operators). Across all benchmarks, on average 21 instructions are needed to execute one bytecode. The short loop is also the reason why the fraction of indirect branch instructions (between 1.01 and 1.07 per bytecode) is higher than Javascript or Python.

Similar to other two interpreters, SandyBridge's predictor is better than Nehalem's, and TAGE and Haswell are better than SandyBridge. Haswell's predictor is comparable to TAGE, with occasional wins for TAGE (2.6 vs 0.21 MPKI on 177.mesa) and for Haswell (0.2 vs 0.38 on 179.art).

The accuracy of *ITTAGE* on the CLI interpreter is particularly poor on vpr and crafty, i.e. MPKI >1. This low accuracy is associated with interpreter footprint issues, and using an ITTAGE of size 50 KB (TAGE3) we confirm that a larger predictor allows to reduce the misprediction rate except for crafty which would still need a larger ITTAGE. The large footprint required by the interpreter on ITTAGE is associated with the huge number of possible targets (478) in the main switch of the interpreter.

5.1.3 Revisiting conventional wisdom

Simulation allows us to observe the individual behavior of specific branch instructions for ITTAGE, and we report on the misprediction ratio of the indirect branch of the dispatch loop in each interpreter (Table 5.5). For Python, we also considered a conditional branch in the macro HAS_ARG (checks if an opcode has an argument) that the source code reports as difficult to predict.

On Python, indirect jumps are very well predicted for most benchmarks, even by the 6KB ITTAGE. However, in several cases the prediction of indirect jump is poor (for example in chaos, django-v2,formatted-log, go). However, these are near perfectly predicted by the 12KB configuration, or with the 50 KB configuration for go. HAS_ARG is also easily predicted by the conditional branch predictor TAGE, with few outliers with 1% to 4% mispredictions.

For Javascript, the indirect branch is again easily predicted with misprediction rates generally lower than 1% with the 12KB ITTAGE. Outliers are again better predicted with a 50KB predictor, except code-load. However, code-load executes more than 4,000 instructions per bytecode on average, and the predictability of the indirect jump has a very limited impact on overall performance.

With the CLI interpreter, the main indirect branch suffers from a rather high

Table 5.5 (IT)TAGE misprediction results for "hard to predict" branch, TAGE 1, TAGE 2 and TAGE 3 (all numbers in %)

Python	indirect			ARG	Javascript			
	IT 1	IT 2	IT3	TAGE		IT 1	IT 2	IT3
call-meth	0.827	0.827	0.827	0.000	ai-astar	0.012	0.012	0.012
call-meth-slots	0.827	0.827	0.827	0.000	a-beat-detec.	0.064	0.130	0.063
call-meth-unk	0.626	0.626	0.626	0.000	audio-dft	0.003	0.003	0.003
call-simple	0.991	0.991	0.990	0.000	audio-fft	0.064	0.064	0.064
chaos	0.165	21.362	0.163	3.456	a-oscillator	0.001	0.001	0.001
django-v2	0.308	22.421	0.016	0.372	i-darkroom	0.023	0.083	0.023
fannkuch	0.652	0.718	0.630	0.001	i-desaturate	0.000	0.000	0.000
fastpickle	0.478	0.634	0.397	0.042	i-gaussian-blur	0.062	0.062	0.062
fastunpickle	0.723	3.730	0.547	0.060	j-parse-financial	0.317	0.572	0.163
float	0.008	0.010	0.005	0.821	j-s-tinderbox	2.460	2.753	2.043
formatted-log	0.136	48.212	0.021	1.216	c-aes	0.323	7.725	0.288
go	4.407	13.521	1.728	0.980	c-ccm	3.429	5.666	0.252
hexiom2	1.749	4.510	0.571	1.042	c-pbkdf2	0.100	2.039	0.094
json-dump-v2	0.015	0.200	0.001	0.841	c-sha256-it.	0.305	0.827	0.085
json-load	2.580	3.676	0.057	1.630	box2d	7.362	12.618	0.936
meteor-contest	0.460	0.558	0.281	0.583	code-load	31.662	36.134	24.257
nbody	0.002	0.003	0.002	0.758	crypto	0.166	0.494	0.108
nqueens	0.518	0.526	0.515	0.357	deltablue	0.380	3.042	0.032
pathlib	0.104	3.635	0.027	0.965	earley-boyer	0.902	5.264	0.748
pidigits	1.719	3.169	0.719	2.958	gbemu	1.690	2.688	0.569
raytrace	0.873	6.691	0.228	3.861	mandreel	2.356	4.742	0.366
regex-compile	9.852	16.284	0.567	0.589	navier-stokes	0.024	0.029	0.022
regex-effbot	1.311	1.608	0.848	0.177	pdf	0.506	1.424	0.351
regex-v8	1.678	2.374	0.918	0.112	raytrace	6.100	20.337	0.439
richards	0.420	6.775	0.337	1.602	regexp	0.427	0.813	0.324
silent-logging	0.316	0.321	0.308	0.004	richards	0.544	2.769	0.474
simple-logging	0.030	53.552	0.021	1.197	splay	1.016	4.509	0.895
telco	0.264	0.342	0.172	0.044	typescript	18.291	29.630	4.100
unpack-sequence	0.027	0.029	0.025	0.001	zlib	2.079	6.771	0.228

CLI							
	IT 1	IT 2	IT 3		IT 1	IT 2	IT 3
164.gzip	0.612	2.698	0.569	183.equake	0.185	0.541	0.113
175.vpr	12.527	27.812	0.905	186.crafty	32.405	45.311	9.688
177.mesa	0.050	1.026	0.019	188.ammp	0.382	1.752	0.222
179.art	0.077	0.083	0.075	197.parser	1.881	7.939	0.786
181.mcf	1.020	1.994	0.681	256.bzip2	2.190	3.102	0.470

misprediction rate when executing vpr, crafty and bzip2 with a 12KB ITTAGE. But a 50KB ITTAGE predictor strictly reduces this misprediction rate except for

crafty which would need an even larger ITTAGE predictor.

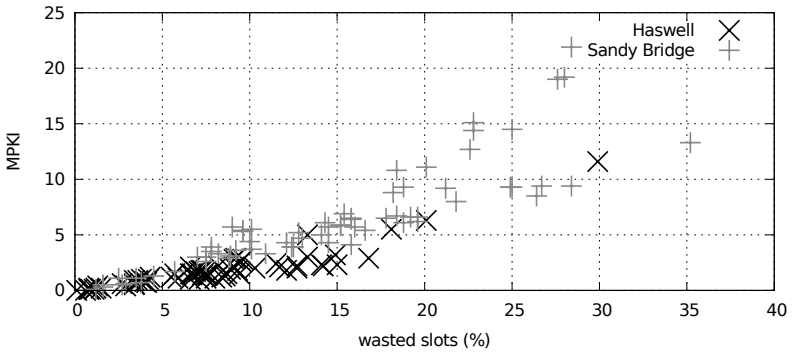


Figure 5.1: Correlation between MPKI and lost slots

Next, we compute the number of wasted instruction issue slots in the processor front end due the mispredicted indirect branch using a methodology described by Intel [MCY]. We relied on Andi Kleen's implementation pmu-tools [1] and backported the formulas, only Sandy Bridge and Haswell PMUs are supported. In the front-end, issue slots can be wasted in several cases: branch misprediction, but also memory ordering violations, self modifying code (SMC), and AVX related events. We confirmed that AVX never occurs and SMC is negligible in our experiments. Also, excepting very few cases, the number of memory ordering violations is two to three orders of magnitude less than the number of branch mispredictions. Thus nearly all wasted issue slots can be attributed to branch mispredictions. Figure 5.1 shows, for all our benchmarks, how MPKI relates to wasted slots, with lower MPKI correlating with fewer wasted slots. On average Haswell wastes 7.8% of the instruction slots due to branch mispredictions, while Sandy Bridge wastes 14.5%, thus better branch prediction on Haswell results in 50% fewer wasted slots. This confirms that, on last generation Intel processors, Haswell, branch misprediction penalty has only a limited impact on interpreted applications execution time.

We have seen that on three interpreters and modern benchmarks, the indirect branch is very predictable provided the usage of a reasonably large enough efficient indirect jump predictor and the misprediction penalty itself adds a limited overhead to the the execution time. Therefore the conventional wisdom on the unpredictability of the indirect branch in the dispatch loop, and its impact on the performance of interpreters, is unjustified.

1. https://github.com/andikleen/pmu-tools

5.2 Modeling the performance of multi-threaded programs in the many-core era

With the growing trend of offloading demanding computation tasks to many-cores based accelerators such as GPGPUs and Xeon-Phi, there is a need to understand and estimate the performance potential of multithread (MT) applications on these massively parallel architectures. Two previous models Amdahl's law [Amd67] and Gustafson's law [Gus88] (Eq. 5.1 and Eq. 5.2, where s is the speedup, f is parallel fraction in the program, and P is the number of cores) are widely used to extrapolate the potential performance of a parallel application on a large machine.

$$s_{Amdahl} = \frac{1}{(1-f) + \frac{f}{P}} \quad (5.1) \qquad s_{Gustafson} = (1-f) + f * P \quad (5.2)$$

Amdahl's law presents a fixed workload perspective of performance and assumes that the input set size (workload) of an application remains fixed for a particular execution. Gustafson's law presents a scaled workload perspective and assumes that the relative part of the parallel computation grows with the problem or input set size but ignores the serial section. These performance models may not accurately capture the scaling behavior in the serial section of the applications, and thus may lead to optimistic estimates.

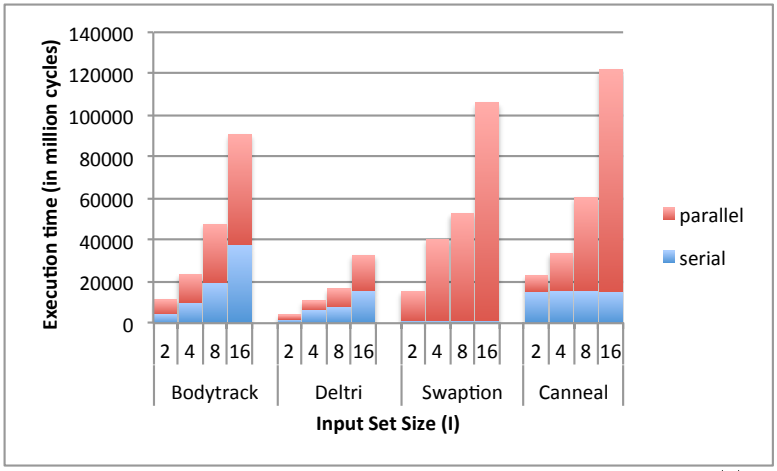


Figure 5.2: Serial scaling behaviour with variation in input set size (I) on Xeon-phi.

For some applications, the execution time of the serial-section increases significantly with the increase in input size, but also slightly with the increase in number of processors (Figure 5.2). In data parallel applications (swaptions, canneal) once threads are spawned they work until the assigned job is complete without any intervention, i.e. the Region Of Interest (ROI) is totally parallel and serial section is independent of I. In swaptions the serial section is ignorable, whereas in canneal contribution of serial section cannot be ignored. On the other hand, applications that uses pipeline parallelism or a worker thread pool based implementation (bodytrack, deltri) have a ROI which contributes to the serial part. The master thread does some work to feed the worker threads in ROI, which can be a significant contribution to serial section and scales with Input set size. In bodytrack, both serial and parallel execution time grows at different rate, whereas in deltri both grow linearly with I.

Here we present an empirical study of serial scaling behavior of MT applications as a function of input workload size and the number of cores. For some MT applications in the benchmark suites we analysed, our study shows that the serial fraction in the program increases with input workload size and can be a scalability-limiting factor. Similar to previous studies [HM08], we show using an area-performance model that heterogeneous architectures where a powerful core executes the serial part can mitigate the impact of serial scaling and improve overall performance of applications.

5.2.1 Serial Scaling Model

The serial scaling model (SSM) only uses parameters which are obtained empirically to represent the execution time of a parallel application. To keep the model simple, we consider the execution time is dependent only on input set size I and the number of processors/cores P. Our model assumes a uniform parallel section and a uniform serial section, i.e, the total execution time is modeled as the sum of serial and parallel execution times (Eq. 5.3).

$$t(I,P) = t_{seq}(I,P) + t_{par}(I,P) \tag{5.3}$$

For both execution times, scaling with input set size (I) and scaling with number of processors (P) are independent, so we model $t_{par}(I,P) = F_{par}(I) * G_{par}(P)$ and $t_{seq}(I,P) = F_{seq}(I) * G_{seq}(P)$. Generally execution time of an application with constant input set size reduces with number of cores, and execution time increases gradually when input set size is increased with fixed number of cores.

Using a non-linear power model F and G can be represented by a function of the form $h(x) = x^{\alpha}$. Thus, the general form of execution time of a parallel application is:

$$t(I, P) = c_{seq} I^{as} P^{bs} + c_{par} I^{ap} P^{bp} \tag{5.4}$$

Thus 6 empirically obtained parameters represent the execution time of a parallel application, taking into account its input set and the number of processors. c_{seq} , as and bs are used to model the serial execution time and c_{par}, ap and bp are used to model the parallel execution time. c_{seq} and c_{par} are serial and parallel section constants which give the initial magnitude of the execution time. as and ap are the *Input Serial Scaling* (ISS) parameter and *Input Parallel Scaling* (IPS) parameter. bs and bp are the *Processor Serial Scaling* (PSS) parameter and *Processor Parallel Scaling* (PPS) parameter.

5.2.2 Methodology for model construction and validation

SSM model should be used to extrapolate performance on large many cores. We studied programs from PARSEC [BKSL08] and LONESTAR [KBCP09] suite on two platforms, the out-of-order Xeon (E5645, upto 24 threads) and the in-order Xeon-Phi (5110P, upto 240 threads) that represent the large-core and small-core approach of many core architectures respectively. We target programs that could run successfully on the hardware platforms, and input sets could be generated with known scaling factors : Bodytrack (body), Canneal (can), Fluidanimate (fluid) and Swaptions (swap) from PARSEC, and Delaunay triangulation (deltri), preflowpush (preflow), Boruvka's Algorithm (Bourvka), barneshut (barnes), Surveypropogation (survey) from LONESTAR.

The following methodology was used to build and validate the model on target applications to obtain the 6 SSM parameters (Table 5.6).

Data collection: Application execution is monitored for Performance Monitoring Unit (PMU) samples (number of instructions executed , number of unhalted clock cycles) collected on per thread basis using tiptop [Roh11] at a regular interval of 1ms.

Post processing: Per thread activity of the application is analyzed and execution time spent in the serial and parallel parts are calculated using counts for PMU event 'unhalted clock cycle'.

Model building For each application and hardware platform, time spent in serial and parallel execution i,e. $t_{seq}(I, P)$ and $t_{par}(I, P)$ is obtained by varying the

number of threads(P), the input set size(I). Then, regression analysis is performed with the least-square method to determine the 6 SSM parameters that best fit the available experimental data.

Validation Per application SSM models are validated using holdout cross-validation method [GOÖ09]. Experimental data is divided into training and validation sets, training set ($I <= 16$, $P <= 16$) is used to drive regression analysis and fit model parameters, and validation set is is used to validate prediction capability of the constructed models. On Xeon, model is validated with ($I = 32$,$P <= 24$), the prediction error lies in the range +/- 13%. On Xeon-phi, model is validated with ($I = 32$,$P <= 128$), the prediction error lies in the range +/-30%.

Table 5.6 SSM parameters for Xeon-Phi and Xeon [a]

	Xeon-Phi		
	Complete application	ROI	
	serial section	Serial section	Parallel section
can	$14725.8I^{0.001}P^{0.003}$	0	$32138.1I^{0.95}P^{-0.873}$
swap	0	0	$33367.4I^{1.035}P^{-0.744}$
fluid	$1163.46I^{0.002}P^{0.076}$	0	$6438.6I^{1.024}P^{-0.783}$
body	N.A	N.A	N.A
deltri	$2716.9I^{0.994}P^{-0.007}$	$1669.8I^{0.998}P^{-0.012}$	$72750.5I^{1.03}P^{-0.602}$
preflow	$1334.8I^{0.965}P^{-0.001}$	$41.205I^{0.919}P^{-0.002}$	$103915I^{0.978}P^{-0.979}$
boruvka	$492.7I^{0.978}P^{-0.023}$	$364.2I^{0.153}P^{-0.179}$	$27935.0I^{1.061}P^{-0.709}$
barnes	$10.078I^{1.004}P^{-0.027}$	$2.023I^{1.288}P^{-0.148}$	$593.015I^{2.119}P^{-0.896}$
survey	$937.96I^{1.094}P^{-0.024}$	$159.024I^{1.079}P^{-0.041}$	$100371I^{1.114}P^{-0.752}$

	Xeon		
	Complete application	ROI	
	serial section	Serial section	Parallel section
can	$5223I^{0.002}P^{-0.003}$	0	$14005.1I^{0.962}P^{-0.843}$
swap	0	0	$8362.0I^{1.027}P^{-0.984}$
fluid	$1013.901I^{0.1}P^{0.173}$	0	$2372.0I^{0.984}P^{-0.738}$
body	$1227.83I^{0.997}P^{0.005}$	$1184.76I^{0.988}P^{0.027}$	$22743.9I^{1.012}P^{-0.989}$
deltri	$951.53I^{1.028}P^{0.027}$	$99.96I^{1.076}P^{0.019}$	$1130.2I^{1.039}P^{-0.614}$
preflow	$134.69I^{1.026}P^{0.102}$	$130.408I^{1.115}P^{0.088}$	$4512.7I^{1.057}P^{-0.633}$
boruvka	$456.407I^{0.902}P^{0.066}$	0	$11247I^{1.066}P^{-0.936}$
barnes	$54.459I^{1.015}P^{-0.012}$	$3.023I^{1.33}P^{-0.054}$	$6187.1I^{1.964}P^{-0.971}$
survey	$454.205I^{1.026}P^{0.002}$	$42.113I^{1.006}P^{0.073}$	$16486.7I^{1.092}P^{-0.549}$

a. (N.A: program was not build-able for the architecture, 0 : negligible serial section.)

5.2.3 Sub-linear performance scaling

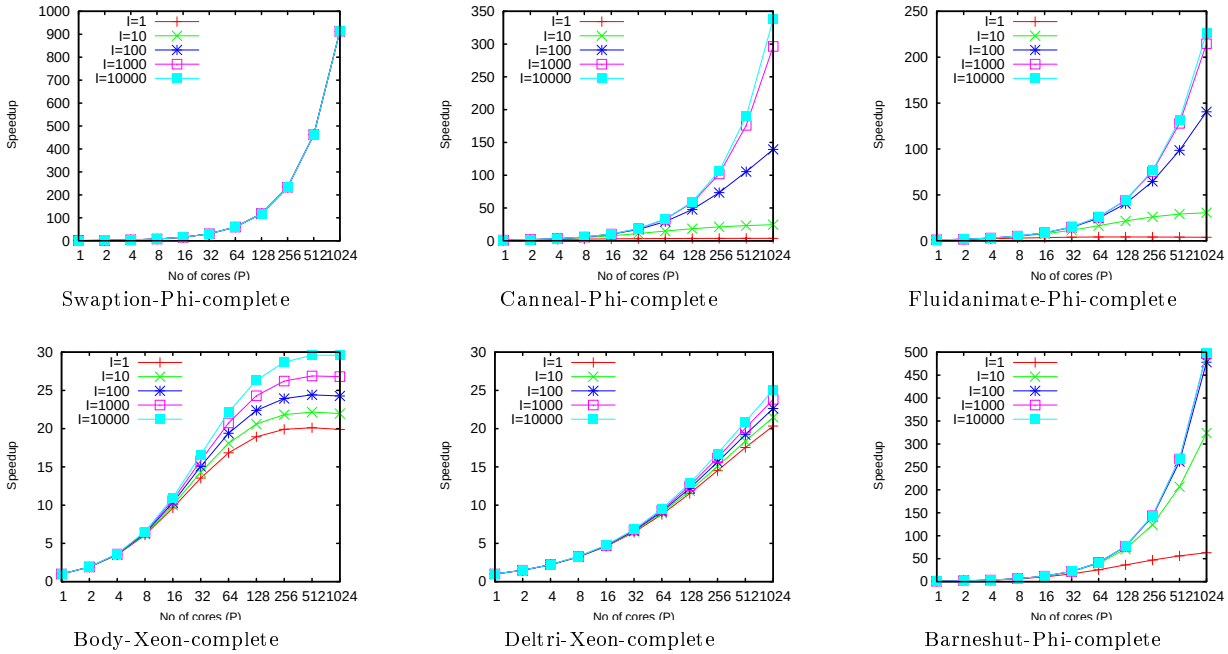


Figure 5.3: Extrapolated speedups for varying P and I, P=1 to 1,024 and I=1 to 10,000.

Figure 5.3 illustrates the potential speedups extrapolated for a few benchmarks varying the processor number from 1 to 1,024 and varying the problem size from 1 to 10,000. The illustrated examples are representative of the behaviours that were encountered among both the chosen architectures.

SSM takes into account that potential speed-up on the parallel section is sub-linear i.e., $PPS > -1$ in most of the benchmarks. Few benchmarks like *swaptions, barneshut and bodytrack* in Xeon have good parallel scaling with $-1 \leq PPS \leq -0.9$, and their speedup can be still in between 1024 to 512 for 1024 cores. Many benchmarks have sublinear scaling in the range $-0.9 \leq PPS \leq -0.6$, e.g. *canneal, fluidanimate, survey, deltri, sssp and bfs* where the maximum achievable speedup will be between 512 and 64 with 1024 cores.

Some applications have a negligible serial section and are highly scalable. As an example, in *swaptions*, the serial section can be ignored in both ROI and complete application, and parallel section scales almost linearly in Xeon i,e $PPS = -0.984$, so the application may achieve nearly perfect scaling. How-

ever, the same application scales sublinearly when executed in Xeon-phi with $PPS = -0.744$, which can be attributed to architecural impact on application scalability.

Some applications have almost constant serial part and rapidly growing parallel part for every input set size, large input set sizes are needed to amortize the constant serial part. In *canneal complete and fluidanimate complete*, large $\frac{c_s}{c_p}$, small ISS and PSS makes the serial section independent of I and P. Parallel section scales quasi linearly with I and P, and significant improvement in speedup is obtained when using larger I.

In certain other applications like *deltri, preflow, survey, boruvka, bodytrack complete*, serial part scales equally or a little lesser than the parallel part i,e $ISS \approx IPS$ and PSS is sublinear. These applications hardly benefit from manycore architectures, with speedup saturating with P despite increasing I due to impact of serial scaling.

Even when execution time of serial section in an application increases with input set, it does not always impact scalability. In *barneshut complete*, the execution of serial section increases with input set size ($ISS = 1.004$), but at a much lower rate than execution time of parallel section ($IPS = 2.1$).

5.2.4 Serial scaling and Heterogeneous architectures

Hill and Marty [HM08] show that heterogeneous multicores can offer potential speedups that are much greater than homogeneous multicore chips. Heterogeneous cores featuring few very large cores, allow the use of a powerful core to speedup the serial section to amortize/reduce the impact of serial scaling on overall performance. In this study, we consider a heterogeneous core consisting of one Xeon like large-core and many Xeon-phi like small-cores (only those applications that successfully execute same input sizes on both hardware platforms were considered). Similar to [HBT13], we assume three small cores are area-equivalent to one large core and present area-performance plots (Figure 5.4) for three design points: (1) large-cores, large Xeon-like cores (2) small-cores, small Xeon-Phi like cores (3) hybrid-cores, one large core executes serial section and small cores execute parallel section.

In canneal and survey, hybrid-cores scales better since the serial section is executed on a faster Xeon-like core. Swaption does not have any serial section (Table. 5.6) and does not benefit from a larger core to execute the serial section, but large-cores have good speedup since parallel section scales better on

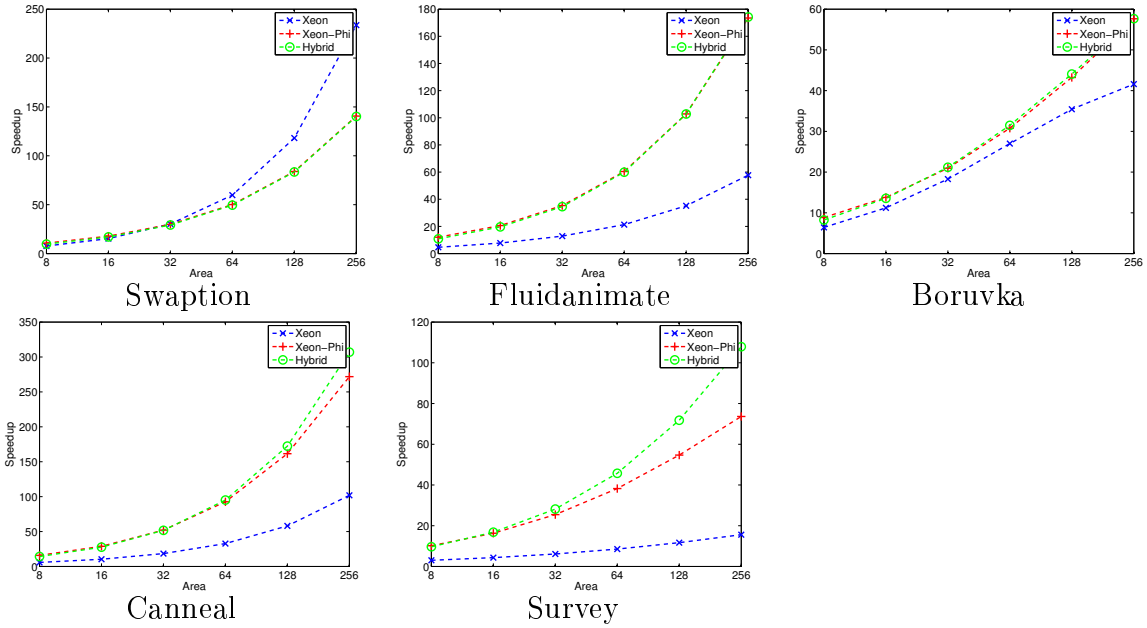


Figure 5.4: Area-Performance graph showing Hybrid architecture has better speedup with serial scaling.

Xeon than Xeon-phi. Fluidanimate, also, does not gain from the large core in hybrid-cores architecture; however, small and hybrid cores perform better as the application scales better in Xeon-phi. Boruvka similarly does not benefit from hybrid cores, and similar to swaption, parallel section scales better in Xeon and hence large-cores scales on par with small-cores.

Conclusion

Conventional wisdom in the processor industry holds that we are in the midst of the multi-core era. Processors for the high end server market transitioned to multi-core in 2001, the mainstream desktop and laptop market in 2005, with the consumer client market making the shift to multi-cores recently. With increasing number of transistors being integrated on a single chip during every new generation of silicon fabrication technology, it is expected that in the near future processors will be able to feature hundreds of cores. In parallel, the industry is also witnessing a shift away from simply replicating homogeneous, either small or large cores, and towards greater heterogeneity.

Heterogeneous Many Cores(HMC) architectures with processors differentiated in terms of core-size and performance help architects to achieve area-performance goals for emerging workloads under fixed energy constriants. They enable a wider spectrum of design trade-offs by matching the performance needs of applications to the power-performance profile of available cores. Single threaded workloads requiring high ILP (instruction level parallelism) are mapped to run on powerful high performance large cores, while parallel workloads demanding high TLP (thread level parallelism) are mapped to power-efficient throughput oriented small cores.

In a HMC, parallel applications that feature many threads will benefit from the large number of cores that support greater TLP. However, legacy single threaded applications and sequential code sections within parallel applications cannot benefit from the increased core count, resulting in a need to improve single threaded performance on HMCs. In addition to traditionally growing/strengthening the large core itself, techniques that exploit additional cores for sequential performance for the main thread are being revisited in the context of HMCs.

In this thesis, we explore the use of small cores in the HMC to run precomputation based helper threads that can generate prefetches for the main thread. Instead of turning off small cores when sequential(main) threads are mapped to

large cores, helper threading allows them to be employed for achieving even higher sequential performance on the main thread.

When porting the helper threading paradigm to the HMC architecture template the following three issues need to be considered

1. Overheads in helper execution

   When using small cores to execute helper threads in a HMC, helper-execution related actions such as thread-spawn and synchronization incur overheads due to inter-processor communication latencies through the shared caches, and also operating system overhead.

2. Suitability of small cores for executing helper threads

   Due to performance disparity between the large and the small cores, it is not clearly known if helpers executing on the small cores can provide sufficient lookahead. i.e generate prefetches in-time, to benefit the main thread running on a much powerful large core.

3. Latency overhead of shared caches

   Since HMC is a shared memory architecture, the helper is limited to prefeching cache lines to reside in the shared last level cache (L3). The main thread incurs an additional latency to access the shared cache even when the prefetch is successful. This inter-core resource independence i.e. lack of shared resources closer to the cores prevents realization of full benefit of helper threading [BWC+02].

In this thesis, we focus on efficient helper threading on small/simple cores in a HMC processor. To address the issue of overheads in helper threading, we propose a hardware/software framework called *core-tethering* which adds new user mode instructions that provide a co-processor like interface to the helper cores in a HMC. It reduces overheads of helper threading on small cores by allowing a large core to directly initiate and control helper execution, and to efficiently transfer application context needed for execution to the helper core.

Second, we evaluate the suitability of small cores to prefetch for the more powerful large core using trace based simulation and memory intensive programs from standard benchmark suites. We find that helper threads running on a moderately sized small cores can significantly accelerate much larger main cores compared to using a hardware prefetcher alone, and that small cores provide a good trade-off against using an equivalent large core to execute the helper. We also find that helper prefetching on small cores when used together with hardware prefetching,

can provide an alternative to growing the core size for achieving higher performance on memory intensive applications.

In summary, despite the latency overheads of accessing prefetched cache lines from the shared L3 cache, helper thread based prefetching on small cores looks as a promising way to improve single thread performance on memory intensive workloads in HMC architectures.

Bibliography

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[ARM] Big.Little processing with Arm Cortex-A15 & Cortex-A7. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf.

[BBSG11] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15, 2011.

[BCD+11] Brad Burgess, Brad Cohen, Marvin Denman, Jim Dundas, David Kaplan, and Jeff Rupley. Bobcat: AMD's low-power x86 processor. *Micro, IEEE*, 31(2):16–25, 2011.

[BG04] Doug Burger and James R Goodman. Billion-transistor architectures: There and back again. *Computer*, 37(3):22–28, 2004.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[BWBJ11] Jeffrey D Brown, Sandra Woodward, Brian M Bass, and Charles L Johnson. Ibm power edge of network processor: A wire-speed system on a chip. *IEEE Micro*, 31(2):76–85, 2011.

[BWC+02] Jeffery A Brown, Hong Wang, George Chrysos, Perry H Wang, and John P Shen. Speculative precomputation on chip multiprocessors. *MTEAC-6*, 2002.

[CE12] George Chrysos and Senior Principal Engineer. Intel xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.

[CEG07] Kevin Casey, M Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):37, 2007.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[CH07] Pat Conway and Bill Hughes. The amd opteron northbridge architecture. *Micro, IEEE*, 27(2):10–21, 2007.

[CO98] Michael K Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded java programs. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 176–184. IEEE, 1998.

[CRDI07] Thomas Chen, Ram Raghavan, Jason N Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation, a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[CSK+99] Robert S Chappell, Jared Stark, Sangwook P Kim, Steven K Reinhardt, and Yale N Patt. Simultaneous subordinate microthreading (ssmt). In *ISCA-26*, 1999.

[CTWS01] Jamison D Collins, Dean M Tullsen, Hong Wang, and John P Shen. Dynamic speculative precomputation. In *MICRO-34*, 2001.

[DS98] Michel Dubois and Y Song. Assisted execution. *University of Southern California CENG Technical Report*, 1998.

[EG01] M Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001 Parallel Processing*, pages 403–413. Springer, 2001.

[EG03] M Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[FCJV97] Keith I Farkas, Paul Chow, Norman P Jouppi, and Zvonko Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *ISCA-24*, 1997.

[FSB+11] Denis Foley, Maurice Steinman, Alex Branover, Greg Smaus, Antonio Asaro, Swamy Punyamurtula, and Ljubisa Bajic. Amd's 'llano' fusion apu. In *Hot Chips*, volume 23, 2011.

[GB05] Ilya Ganusov and Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 350–360. IEEE, 2005.

[GB06] Ilya Ganusov and Martin Burtscher. Efficient emulation of hardware prefetchers via event-driven helper threading. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 144–153. ACM, 2006.

[GH08] Alok Garg and Michael C Huang. A performance-correctness explicitly-decoupled architecture. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 306–317. IEEE, 2008.

[Gle98] Andrew Glew. Mlp yes! ilp no. *ASPLOS Wild and Crazy Idea Session 1998*, 1998.

[GOÖ09] Tom Gruber, In Ling Liu Ontology, and M Tamer Özsu. Encyclopedia of database systems. *Ontology*, 2009.

[GOSS13] Michael K Gschwind, John K O'brien, Valentina Salapura, and Zehra N Sura. Method and apparatus for efficient helper thread state initialization using inter-thread register copy, May 28 2013. US Patent 8,453,161.

[GPH11] Alok Garg, Raj Parihar, and Michael C Huang. Speculative parallelization in decoupled look-ahead. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 413–423. IEEE, 2011.

[Gre04] D Greenley. The Ultrasparc IV+ processor. In *Proceedings of the Fall Processor Forum*, 2004.

[Gre11] Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White Paper*, 2011.

[Gus88] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[GVDB01] Bart Goeman, Hans Vandierendonck, and Koenraad De Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 207–216. IEEE, 2001.

[HBT13] Tomas Hruby, Herbert Bos, and Andrew S Tanenbaum. When slower is faster: On heterogeneous multicores for reliable systems. In *USENIX Annual Technical Conference*, pages 255–266, 2013.

[HFE04] Wessam Hassanein, José Fortes, and Rudolf Eigenmann. Data forwarding through in-memory precomputation threads. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 207–216. ACM, 2004.

[HLSS13] Ronald Hall, Hung Le, Raul Silvera, and Balaram Sinharoy. Hardware assist thread for increasing code parallelism, April 16 2013. US Patent 8,423,750.

[HM08] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[IKKM07] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 186–197. ACM, 2007.

[JG97] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[JL01] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 197–206. IEEE, 2001.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, 2005.

[KBCP09] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *Per-*

formance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pages 65–76. IEEE, 2009.

[KC11] Alain Ketterlin and Philippe Clauss. Efficient memory tracing by program skeletonization. In *ISPASS'11*, 2011.

[KFJ+03] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92. IEEE, 2003.

[KHL+12] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *PACT'12*, 2012.

[KKS+14] David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. B-fetch: Branch prediction directed prefetching for chip-multiprocessors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[KST11] Md Kamruzzaman, Steven Swanson, and Dean M Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. *ASPLOS'11*, 2011.

[KT99] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *Computers, IEEE Transactions on*, 48(9):866–880, 1999.

[LCM+05a] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI'05*, 2005.

[LCM+05b] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[LDH+05] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G Abraham. Dynamic helper threaded prefetching on the Sun UltraSPARC® CMP processor. In *MICRO-38*, 2005.

[LJLS09] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. Prefetching with helper threads for loosely coupled multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 20(9):1309–1324, 2009.

[LMC+11] Eric Lau, Jason E Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. Multicore performance optimization using partner cores. In *3rd USENIX HotPar'11*, 2011.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *Ieee Micro*, 28(2):39–55, 2008.

[LWW+02] Shih-Wei Liao, Perry H. Wang, Hong Wang, John Paul Shen, Gerolf Hoflehner, and Daniel M. Lavery. Post-pass binary adaptation for software-based speculative precomputation. In *PLDI'02*, 2002.

[MCY] Jackson Marusarz, Shannon Cepeda, and Ahmad Yasin. How to tune applications using a top-down characterization of microarchitectural issues. Technical report, Intel.

[MH11] Micheal Mantor and Mike Houston. Amd graphic core next: Low power high performance graphics & parallel compute. In *Symposium on High-Performance Graphics*, 2011.

[MSS10] Pierre Michaud, Yiannakis Sazeides, and André Seznec. Proposition for a sequential accelerator in future general-purpose manycore processors and the problem of migration-induced cache misses. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 237–246. ACM, 2010.

[MWU+08] Jason Mars, Daniel Williams, Dan Upton, Sudeep Ghosh, and Kim Hazelwood. A reactive unobtrusive prefetcher for multicore and manycore architectures. In *Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms (SHCMP)*, 2008.

[NSS14] Surya Narayanan Natarajan, Bharath Narasimha Swamy, and Andre Seznec. Impact of serial scaling of multi-threaded programs in many-core era. In *5th Workshop on Applications for Multi-Core Architectures, 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014)*, 2014.

[Nvia] Nvidia. NVIDIA Tegra K1, A New Era in Mobile Computing. http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-K1-whitepaper.pdf.

[Nvib] Nvidia. Tegra K1 and the automotive industry. http://on-demand.gputechconf.com/gtc/2014/presentations/S4412-tegra-k1-automotive-industry.pdf.

[Nvic] Nvidia. Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf.

[ONH+96] Kunle Olukotun, Basem A Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *ASPLOS'96*, 1996.

[PDG06] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *ICCAD'06*, 2006.

[PJS97] Subbarao Palacharla, Norman P Jouppi, and James E Smith. *Complexity-effective superscalar processors*, volume 25. ACM, 1997.

[PPE+97] Yale N Patt, Sanjay J Patel, Marius Evers, Daniel H Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, 1997.

[PSG+09] Gennady Pekhimenko, Zehra Sura, Yaoqing Gao, Tong Chen, and John K. O'Brien. Assist threads for data prefetching in IBM XL compilers. *8th Workshop on Compiler-Driven Performance, 19th IBM CASCON*, 2009.

[RDC+11] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramirez, and Mateo Valero. Trace-driven simulation of multithreaded applications. In *ISPASS'11*, 2011.

[Roh11] Erven Rohou. Tiptop: Hardware Performance Counters for the Masses. Rapport de recherche RR-7789, INRIA, November 2011.

[RSS15] Erven Rohou, Bharath Narasimha Swamy, and Andre Seznec. Branch prediction and the performance of interpreters - don't trust folklore. In *Code Generation and Optimization (CGO), 2015 15th International Symposium on*, 2015.

[RTM+10] Stefan Rusu, Simon Tam, Harry Muljono, Jason Stinson, David Ayers, Jonathan Chang, Raj Varada, Matt Ratta, Sailesh Kottapalli, and Sujal Vora. A 45 nm 8-core enterprise xeon processor. *Solid-State Circuits, IEEE Journal of*, 45(1):7–14, 2010.

[SCZM05] J Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems (TOCS)*, 23(3):253–300, 2005.

[Sez10] Andre Seznec. Dal: Defying amdahl's law, 2010.

[Sez11] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127. ACM, 2011.

[SKKC09] Seung Woo Son, Mahmut Kandemir, Mustafa Karakoy, and Dhruva Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *PPoPP'09*, 2009.

[SKS+11] Balaram Sinharoy, R Kalla, WJ Starke, HQ Le, R Cargnoni, JA Van Norstrand, BJ Ronchetti, J Stuecheli, J Leenstra, GL Guthrie, et al. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1–1, 2011.

[SKS14] Bharath Narasimha Swamy, Alain Ketterlin, and Andre Seznec. Hardware/software helper thread prefetching on heterogeneous many cores. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 214–221. IEEE, 2014.

[SKT05] Yonghong Song, Spiros Kalogeropulos, and Partha Tirumalai. Design and implementation of a compiler framework for helper threading on multi-core processors. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 99–109. IEEE, 2005.

[SLT02] D Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA-29*, 2002.

[SM06] Andre Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of ILP,vol. 8*, pages 1–23, 2006.

[SMQP09] M Aater Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 253–264. ACM, 2009.

[STR02] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization: a path to complexity-effective wide-

issue superscalar processors. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 383–394, 2002.

[TBS08] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th annual Design Automation Conference*, pages 772–775. ACM, 2008.

[TIL] Tilepro. http://www.tilera.com/products/processors/TILEPro_Family.

[VT14] A. Venkat and D.M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 121–132, 2014.

[WBS+08] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M Aamodt, Jamison D Collins, Perry H Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang. Pangaea: a tightly-coupled ia32 heterogeneous chip multiprocessor. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 52–61. ACM, 2008.

[WDY+06] Shengyue Wang, Xiaoru Dai, Kiran S Yellajyosula, Antonia Zhai, and Pen-Chung Yew. Loop selection for thread-level speculation. In *Languages and Compilers for Parallel Computing*, pages 289–303. Springer, 2006.

[WFKL10] Dong Hyuk Woo, Joshua B Fryman, Allan D Knies, and Hsien-Hsin S Lee. Chameleon: Virtualizing idle acceleration cores of a heterogeneous multicore processor for caching and prefetching. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(1):3, 2010.

[Wil] David Williamson. Arm cortex a8: A high performance processor for low power applications. *ARM White Paper*.

[WL10] Dong Hyuk Woo and Hsien-Hsin S Lee. COMPASS: a programmable data prefetcher using idle GPU shaders. In *ASPLOS'10*, 2010.

[WWW+02] Hong Wang, Perry H Wang, Ross Weldon, Scott Ettinger, Hideki Saito, Milind Girkar, Shih-Wei Liao, and John P Shen. Speculative precomputation: Exploring the use of multithreading for latency. *Intel Technology Journal*, 6(1), 2002.

[YXMZ12] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[Zho05] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 231–242. IEEE, 2005.