Modellering en taakplanning van heterogene architecturen met meerdere rekenkernen

Modeling and Scheduling Heterogeneous Multi-Core Architectures

Kenzo Van Craeynest

Promotor:

prof. dr. ir. Lieven Eeckhout

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. Jan Van Campenhout
Faculteit Ingenieurswetenschappen

Academiejaar 2012-2013

UNIVERSITEIT GENT

Examencommissie

Prof. Rik Van De Walle, voorzitter
Decaan, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. David Black-Schaffer
Uppsala University,
Sweden

Prof. Koen De Bosschere, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Jan Fostier
Vakgroep Informatietechnologie, Faculteit Ingenieurswetenschappen
Universiteit Gent

Dr. Jennifer B. Sartor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Aamer Jaleel
Intel, Hudson, MA
USA

Dr. Ibrahim Hur
Intel, Belgium

Leescommissie

Prof. Jan Fostier
Vakgroep Informatietechnologie, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. David Black-Schaffer
Uppsala University,
Sweden

Dr. Jennifer B. Sartor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Aamer Jaleel
Intel, Hudson, MA
USA

Dr. Ibrahim Hur
Intel, Belgium

Samenvatting

In de laatste decennia zijn processors geëvolueerd van relatief eenvoudige scalaire in-order processors tot meerkernige processors met meerdere superscalaire out-of-order rekenkernen. Hierdoor stijgt de complexiteit van de processors en de nood aan hulpmiddelen om te begrijpen waarom de prestatie van een applicatie op een processor is wat ze is. Het beschikken over adequate hulpmiddelen is cruciaal want meer inzicht in het gedrag van de processor zal uiteindelijk leiden naar betere (snellere en/of energie-efficiëntere) ontwerpen.

Simulatie van Meerkernige Processors

Een simulator is een dergelijk hulpmiddel: er wordt een model van een processor in software geïmplementeerd. Zo kunnen ontwerpen geëvalueerd worden alvorens ze effectief te bouwen en kan er meer inzicht verkregen worden in de prestatie van de verschillende onderdelen van de te ontwerpen processor. Het spreekt voor zich dat simulatoren nauwkeurig het gedrag van de processor die ze modelleren moeten nabootsen, maar het is minstens even belangrijk dat de evaluatie voldoende snel kan gebeuren. Hoe sneller de simulatie verloopt, hoe meer parameters, configuraties, werklasten en ontwerpbeslissingen geëvalueerd kunnen worden. Dit probleem wordt exponentieel groter wanneer we meerkernige processors beschouwen: omdat een meerkernige processor meerdere applicaties tegelijk kan uitvoeren, vormt iedere combinatie van programma's een nieuwe potientiële werklast om te evalueren. Gezien simulatie relatief traag is (vaak meerdere grootteordes trager dan de processors die ze modelleren) is het dan ook volledig onhaalbaar om alle (of zelfs maar een fractie van alle) werklasten te evalueren. Er is dus duidelijk nood aan een fundamenteel andere aanpak voor de prestatie-evaluatie en exploratie van de ontwerpruimte van meerkernige processors.

Analytisch Modelleren van Meerkernige Processors

In dit proefschrift onderzoeken we het gebruik van analytische modellen om de prestatie van meerkernige processors te evalueren. Eén van de

grootste uitdagingen bij de prestatie-evaluatie van meerkernige processors zijn de gedeelde systeembronnen — wij hebben ons voornamelijk geconcentreerd op de gedeelde caches. Er is een complexe terugkoppellus waarbij de prestatie van de individuele applicaties bepaald wordt door de manier waarop de gedeelde cacheruimte dynamisch toegewezen wordt aan de verschillende applicaties die samen uitvoeren op de meerkernige processor. Maar de verdeling van de cache beïnvloedt dan weer de prestatie van de applicaties, waardoor de toegangspatronen naar de cache veranderen, wat op zijn beurt leidt tot een andere verdeling in de cache. Het is duidelijk dat de prestatie van een meerkernige processor bepaald wordt door de prestatie van de individuele rekenkernen én de manier waarop gedeelde systeembronnen toegewezen worden aan de rekenkernen.

Wij stellen een methodologie, het *Multi-Program Performance Model* (MPPM), voor waarbij we enkel (trage) gedetailleerde simulatie gebruiken voor de prestatie-evaluatie van de rekenkernen en waarbij we gebruik maken van een iteratief analytisch model om de impact van de gedeelde caches te schatten. Omdat we per applicatie slechts één maal gedetailleerd moeten simuleren, is het MPPM-raamwerk lineair in tijdscomplexiteit m.b.t. het aantal te evalueren applicaties. De MPPM-methode voor de prestatie-evaluatie van meerkernige processors is daardoor tot vijf grootteordes sneller dan gedetailleerde simulatie, terwijl de door MPPM geschatte prestatie een fout vertoont van slechts 2.3% voor *system throughput* (STP) en 2.9% voor *average normalized turnaround time* (ANTT) in vergelijking met gedetailleerde simulatie. Bovendien hebben we, gebruik makend van het MPPM-raamwerk hebben we aangetoond dat de gangbare praktijk om slechts een beperkt aantal combinaties van applicaties te evalueren kan leiden tot foutieve ontwerpbeslissingen.

Exploratie van de Ontwerpruimte van Heterogene Meerkernige Processors

Traditioneel bestaan meerkernige processors uit meerdere identieke kernen (zogenaamde homogene meerkernige processors). Heterogene meerkernige processors bestaan uit verschillende types rekenkernen. Heterogene meerkernige processors kunnen veel energie-efficiënter zijn dan homogene meerkernige processors door taken op een type processorkern die best geschikt is voor de gegeven taak (beste prestatie voor het minste vermogenverbruik) uit te voeren.

Het MPPM-raamwerk stelt ons in staat een uitgebreide exploratie te doen van de heterogene ontwerpruimte van meerkernige processors, om zo inzicht te verschaffen in heterogeen meerkernig ontwerp en een antwoord te geven op een aantal fundamentele ontwerpkeuze vragen. We hebben alle mogelijke configuraties van vijf verschillende types processorkernen beschouwd, met verschillende cache configuraties, off-chip band-

breedtelimieten en taakplanningsalgoritmes. De belangrijkste conclusies uit deze studie luiden als volgt:

- We hebben aangetoond dat twee verschillende types processorkernen voldoende zijn om het grootste deel van de energie-efficiëntie van heterogeniteit te bereiken.
- Heterogeniteit houdt een fundamentele afweging in tussen de prestatie van individuele applicaties en de prestatie van de volledige processor.
- Een beperkt aantal combinaties van types processorkernen bestrijkt het grootste deel van de ontwerpruimte.
- Een goede plaatsing van de applicaties op de types processorkernen is cruciaal voor de efficiëntie van heterogene meerkernige processors.
- Beperken van off-chip bandbreedte heeft een grote impact op een aantal fundamentele ontwerpkeuzes van heterogene meerkernige processors.

Taakplanning voor Heterogene Meerkernige Processors

Eén van de belangrijkste aspecten van heterogene meerkernige processors is de taakplanning: heterogeniteit kan enkel tot een betere energie-efficiëntie leiden indien er een goede taakplanning is van de applicaties op de processorkernen. Bovendien vertonen programma's vaak tijdsvariërend gedrag, waardoor het cruciaal is dat taakplanningen dynamisch zijn: de optimale planning verandert naarmate het uitvoeringspatronen van de applicaties veranderen.

Wij stellen *Performance Impact Estimation* (PIE) voor, een methode waarbij de prestatie van een applicatie op andere types processorkernen geschat wordt met behulp van een analytisch model. Omdat we de prestatie niet hoeven te bemonsteren op alle types processorkernen bekomen we een schaalbaar systeem, zowel in het aantal processorkernen als in het aantal types processorkernen. Bovendien vereist PIE slechts zeer beperkte hardwareondersteuning nodig. We hebben PIE geëvalueerd voor verschillende hardwareconfiguraties: voor een set van werklasten waarvoor taakplanning zeer belangrijk is presteert PIE gemiddeld 5.5% beter dan de conventionele technieken en 8.7% beter dan bemonstering-gebaseerde taakplanning. Tevens hebben we aangetoond dat de logica van de conventionele manier van taakplanning (geheugenintensieve taken worden prioritair op de minder krachtige processorkernen geplaatst) ernstige gebreken vertoont in situaties waarbij parallellisme niet behouden wordt op de minder krachtige types processorkernen.

Fairness-gebaseerde Taakplanning

Bijna zonder uitzondering wordt er enkel rekening gehouden met *throughput* (of totale prestatie van de processor) wanneer men de taakplanning evalueert voor heterogene meerkernige processors, inclusief de net besproken PIE-techniek. Niettemin is het zeer belangrijk om naast throughput ook fairness in rekening te brengen voor zowel werklasten bestaande uit meerdere onafhankelijke programma's als meerdradige werklasten. Een hoge fairness betekent dat alle draden in het systeem proportioneel dezelfde vooruitgang maken ten opzichte van geïsoleerde uitvoering. Voor werklasten bestaande uit meerdere onafhankelijke programma's is fairness belangrijk als het gaat over eigenschappen zoals quality-of-service. Voor meerdradige werklasten is fairness dan weer belangrijk vanwege de synchronisatie tussen de draden.

Wij hebben twee manieren van fairness-gebaseerde taakplanning voor heterogene meerkernige processors ontwikkeld en geëvalueerd:
equal-time taakplanning en *equal-progress taakplanning*. *Equal-time taakplanning* tracht fairness te optimaliseren door alle draden evenveel uitvoeringstijd te geven op elke type processorkern. Wanneer echter de uitvoeringssnelheid van de verschillende draden sterk verschillend is tijdens het uitvoeren op de meest krachtige rekenkern, leidt dit niet noodzakelijkerwijze tot meer fairness. *Equal-progress taakplanning* gebruikt de effectief gemaakte vooruitgang om de taakplanning aan te sturen, waardoor ook in deze gevallen een hoge fairness kan behaald worden.

Bovendien hebben we aangetoond dat door fairness te optimaliseren, ook de prestatie verbeterd wordt voor meerdradige applicaties. Voor homogene meerdradige applicaties tonen onze resultaten aan dat de prestatie gemiddeld toeneemt met 14% (tot maximaal 25%) ten opzichte van een statische werkverdeling. Voor heterogene meerdradige applicaties neemt de prestatie gemiddeld zelfs toe met 32%.

Summary

Multi-Cores and Simulation

Over the past few decades, processors have evolved from relatively simple scalar in-order processors to complex multi-core processors with superscalar out-of-order cores. With increasing transistor counts and increasing core count due to Moore's Law, also comes an increase in complexity. As this trend continues, there emerges a growing need for tools that help the architect understand why a processor is performing the way it is and how design decisions impact performance. A software simulator is a tool that accurately predicts the behavior and performance of a processor: designs can be evaluated using a software model of the hardware which models the behavior and performance of all, or at least, the most important processor components. Obviously, a simulator needs to accurate, but the speed at which an application's performance can be evaluated (relative to native execution speed) is equally important. If simulation is too slow, this might be incompatible with tight time-to-market demands, forcing the architect to evaluate fewer design options, potentially yielding a less-than-optimal design.

This problem gets further compounded when considering representative multi-core workloads: a multi-core processor can execute multiple independent programs concurrently, therefore, any combination of benchmarks forms a potential multi-program workload. We explore the use of analytical modeling to increase multi-core simulation speed. Instead of using slow detailed simulation to evaluate performance for a large number of multi-program workloads and multi-core designs, we propose a methodology (the Multi-Program Performance Model, MPPM) in which detailed simulation is done for single-program workloads only. We then employ an iterative analytical model to estimate the impact of the sharing effects in the last-level caches when co-running programs on the multi-core processor. MPPM is both fast and accurate: we achieve simulation speedups of up to five orders of magnitude, and report an average performance prediction error of 2.3% and 2.9% for system throughput (STP) and average normalized turnaround time (ANTT), respectively. Additionally, we demonstrate that current practice of randomly picking a limited number of multi-program

workloads, can lead to incorrect design decisions in practical design scenarios,, ultimately resulting in sub-optimal designs.

Heterogeneous Multi-Core Processors

Recently, heterogeneous multi-core architectures have been proposed: which integrate multiple core types on a single chip. The idea is that applications have different resource requirements during their execution and by matching the application's requirements to the core capabilities, a much higher level of energy-efficiency may be achieved.

In this work, we aim at understanding how heterogeneity affects both chip throughput and per-program performance; how heterogeneous architectures compare to homogeneous architectures under said performance metrics; and how fundamental design choices, such as core type, cache size and off-chip bandwidth, affect performance. We use the MPPM framework to explore the large heterogeneous multi-core architecture design space. Because MPPM has linear-time complexity in the number of core types and programs of interest, it offers a unique opportunity for exploring the large space of both homogeneous and heterogeneous multi-core processors in limited time. We considered five different core types (ranging from a small scalar in-order core to a large 4-wide out-of-order core). We considered multiple last-level cache sizes and evaluated scenarios of varying off-chip bandwidth limits. Our analysis provides several interesting insights:

- Two core types provide most of the benefits from heterogeneity.
- Heterogeneity fundamentally trades per-program performance for chip throughput.
- Some homogeneous configurations are optimal for particular throughput versus per-program performance trade-offs.
- Job-to-core mapping is both crucial and challenging for heterogeneous multi-core processors to achieve optimum performance.
- Limited off-chip bandwidth alters some of the fundamental design choices in heterogeneous multi-core architectures.

Scheduling Heterogeneous Multi-cores

There exists a fundamental challenge in the design space of heterogeneous multi-core processors, which is how best to schedule workloads on the most appropriate core type. Heterogeneous multi-cores can enable higher performance and reduced energy consumption (within a given power budget) if and only if workloads are executed on the most appropriate core

type. Because a lot of workloads show time-varying behavior, a good scheduling policy needs to be dynamic. Additionally, because processor die size continues to steadily increase, a scheduling policy needs to be scalable with the number cores and the number of core types.

We propose Performance Impact Estimation (PIE), which collects basic characteristics from the application on the core that it is running on, and uses these as inputs to an analytical model to estimate performance of the application on the other core types. PIE uses a dynamic scheduling approach and is a flexible and scalable solution because it scales easily with the number of cores and core types. We evaluate PIE scheduling and report improvements in system performance by 5.5% on average over memory-dominance scheduling (conventional wisdom) and by 8.7% over sampling-based scheduling for a set of scheduling-sensitive workload mixes. Additionally, using insights gained from analytical modeling, we have shown that the state-of-the-art scheduling policy only works as expected when considering core types that do not differ significantly in their capabilities for exploiting parallelism.

Fairness-Aware Scheduling

Previously, we focused on optimizing system throughput with PIE scheduling with little attention to fairness. Yet, guaranteeing that all threads make equal progress on heterogeneous multi-cores is important for both multi-threaded and multi-program workloads. Achieving high fairness means that all threads in the system are making equal progress proportional to isolated execution.

For multi-program workloads, fairness is important when it comes to system-level priorities and quality-of-service (QoS) because a heterogeneous system then behaves as a homogeneous one. For a multi-threaded workload, fairness is desirable because of thread synchronization: threads running on a powerful core will typically make faster progress than threads running on a small, low-performance core. When there is need for synchronization, the fast running threads will stall at the barrier until all other threads have reached the barrier.

We propose two techniques to achieve fairness-aware scheduling for heterogeneous multi-cores: equal-time scheduling strives at achieving high fairness by running each thread on each core type for an equal fraction of time. When the threads benefit differently from running on a big core however, equal-time scheduling will not necessarily yield high fairness. Alternatively, equal-progress scheduling can guarantee high fairness for these bases as well: by continuously monitoring progress and by changing the thread-to-core mapping to get equal amounts of work done (instead of time spent) for each thread in the system.

Fairness-aware scheduling improves fairness over pinned scheduling

as done by contemporary operating systems, and it also improves system throughput by enabling threads to run on a big core type for some fraction of time. For homogeneous multi-threaded workloads, we report an average 14% (and up to 25%) performance improvement over pinned scheduling through fairness-aware scheduling. For heterogeneous multi-threaded workloads; equal-progress scheduling improves performance by 32% on average.

Contents

List of Tables

List of Figures

List of Abbreviations

ANTT	Average Normalized Turnaround Time
CMP	Chip Multiprocessor
CPU	Central Processing Unit
EPI	Energy Per Instruction
FOA	Frequency Of Access
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
KIPS	Kilo-Instructions Per Second
LLC	Last-Level Cache
MIPS	Million Instructions per Second
MLP	Memory Level Parallelism
PIE	Performance Impact Estimation
QoS	Quality of service
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
RRIP	Re-reference Interval Prediction
STP	System Throughput
SMT	Simultaneous Multithreading
TLP	Thread Level Parallelism

Chapter 1

Introduction

It never gets easier, you just go faster.
Greg Lemond

1.1 Multi-Cores and Simulation

Over the past few decades, processors have evolved from relatively simple scalar in-order processors to multi-core processors with superscalar out-of-order cores containing complex structures such as branch predictors, large reorder buffers, a multitude of functional units, large instruction and data caches, etc. Historically, the focus for increasing the processor performance was on improving single-core performance. Single-core performance has improved by exploiting more instruction-level parallelism (ILP) and by increasing clock frequency. Due to Moore's law [Moore 1965], which states that the number of transistors that can be integrated on a single chip doubles approximately every two years, this could be done by integrating more and more components that aided in the exploitation of ILP. This trend came to an end for three reasons.

Firstly, as single-core processors continued to increase in performance and clock speed, processor power consumption and heat dissipation increased as well, which in its turn, leads to higher costs for thermal packaging, fans and ultimately, electricity. Up to a point where this becomes a limiter of processor design (this is commonly known as the power wall). Secondly, even though there is sufficient ILP available in the instruction stream [Jouppi and Wall 1989], it becomes increasingly difficult to efficiently extract ILP [Wall 1991]. Finally, main memory performance has been

growing at a much slower pace than the processors performance [Wulf and McKee 1995]. As a result, memory has become increasingly slower compared to the processor. Eventually, this also becomes a limiter for performance and is known as the memory wall.

As the rate of clock speed improvements slowed, increased use of parallel computing has been pursued to improve processor performance. Chip-multiprocessors (CMP) or multi-cores offer an elegant solution to this problem. Advances in chip manufacturing (both transistor scaling and manufacturing processes) have significantly increased the number of transistors on a single chip (as predicted by Moore's law), allowing architects to integrate multiple processor cores on a single die, effectively allowing for making a trade-off between ILP and Thread-Level Parallelism (TLP) [Olukotun et al. 1996]. Another advantage of a multi-core design is that existing processor designs (in which considerable effort and money was invested to create, evaluate and produce) can be used as a starting point by duplicating them on a single processor die, making multi-core a logical step in processor design evolution. CMPs reduce the impact of both the power wall and the memory wall by focusing on throughput instead of single-core performance, allowing the cores to be less aggressive or to run at a lower clock frequency.

With increased transistor densities and core counts also comes an increase in complexity. As this trend continues, there emerges a growing need for tools that help the architect understand why a processor is performing the way it is and how design decisions impact performance (and ultimately, guiding the creation of a new design that performs better than the previous one). A software simulator is a tool that accurately predicts the behavior of a processor and is arguably one of the most powerful tools used by processor designers. Instead of having to physically build processors, designs can be evaluated using a software model of the hardware which models the behavior and performance of all, or at least the most important, processor components.

A simulator has two critical qualities: accuracy and speed. Obviously, a simulator needs to accurately reflect the hardware it is configured to simulate. Simulation speed — the speed at which an application's performance can be evaluated (relative to native execution speed) is equally important: the design process involves a very large number of simulations over a wide range of benchmarks and design options in order to define the optimum design point. If simulation is too slow, this might be incompatible with tight time-to-market demands, forcing the architect to evaluate fewer design op-

tions, potentially yielding a less-than-optimal design.

This problem gets further compounded when considering representative multi-core workloads: a multi-core processor can execute multiple independent programs concurrently, therefore, any combination of programs forms a potential multi-program workload. This results in an exponentially growing workload space as the number of cores continues to increase. Given the very large number of possible multi-program workloads and the limited speed of current simulation methods, it is impossible to evaluate all (or even a fraction of all) possible multi-program workloads. Consider a small benchmark suite that contains only 20 benchmarks, the number of two-program workloads is 210, the number of four-program workloads quickly increases to 8,855 and there are more than 2 million possible eight-program workloads to consider. There is a clear need for a fundamentally different approach to explore multi-core designs that does not rely on detailed on detailed simulation of multi-program workloads.

1.2 Multi-Core Analytical Modeling

In this dissertation, we explore the use of analytical modeling to greatly decrease the time it takes to evaluate large numbers of multi-program workloads on multi-core architectures. One of the biggest challenges with multi-core performance modeling are the shared resources (such as a shared last-level cache). There is a complex feedback mechanism where the contention for these resources changes how much progress the individual cores make. This change causes a different interaction pattern in the shared resource, i.e., it affects how the cores contend for shared resources. It is clear that per-program performance in a multi-core processor is the result of both the architecture and the entire workload. Instead of using slow detailed simulation to evaluate performance for a large number of multi-program workloads and multi-core designs, we propose a methodology in which detailed simulation is done for single-program workloads only. We then employ an iterative analytical model to estimate the impact of the sharing effects in the last-level caches when co-running multiple programs in a multi-program workload.

Contribution #1: The Multi-Program Performance Model

We present the Multi-Program Performance Model (MPPM), a method for quickly estimating multi-program multi-core performance based on single-

core simulation runs. MPPM employs an iterative analytical method to model the tight performance entanglement between co-executing programs on a multi-core processor with shared caches. Because MPPM involves analytical modeling, it is very fast, and it estimates multi-core performance for a very large number of multi-program workloads in a reasonable amount of time with good accuracy (in addition, it provides confidence bounds on its performance estimates). We report an average performance prediction error of 2.3% and 2.9% for system throughput (STP) and average normalized turnaround time (ANTT), respectively, while being up to five orders of magnitude faster than detailed simulation. Subsequently, we demonstrate that current practice of randomly picking a limited number of multi-program workloads (which is motivated by how slow detailed simulation is), can lead to incorrect design decisions in practical design and research studies. MPPM addresses this pitfall by quickly evaluating all possible, or at least a very large number of, multi-program workloads through analytical modeling. This work has been presented at the 2011 IEEE International Symposium on Workload Characterization (IISWC):

> K. Van Craeynest and L. Eeckhout. The Multi-Program Performance Model: Debunking Current Practice in Multi-Core Simulation. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pages 26-37, Nov 2011.

1.3 Heterogeneous Multi-Core Processors

Whenever we discussed multi-core processors so far, there was an implicit assumption that the cores are all identical, i.e., a so-called homogeneous multi-core. Recently, non-uniform or heterogeneous multi-core architectures have been proposed [Kumar et al. 2003; 2004, Becchi and Crowley 2008]. These are multi-cores containing multiple core types, where every core type typically has different performance and power levels. For the scope of this thesis, we will assume that all the core types implement the same instruction-set architecture (ISA), i.e., single-ISA heterogeneous multi-cores.

The idea behind heterogeneity is that different applications have different resource requirements. For example, some applications may be compute-bound and benefit from a core that can issue many instructions per cycle (i.e., a wide-issue superscalar core). The same core, however, might be wasting power and energy while running an application that

does not benefit from these resources. By matching the application's requirements to a core's capabilities, a much higher level of energy-efficiency can be achieved through heterogeneity.

Prior work in heterogeneous architectures has mainly focused on how heterogeneity can improve overall system throughput. To what extent heterogeneity affects per-program performance has remained largely unanswered. With this work, we aim at understanding how heterogeneity affects both chip throughput and per-program performance; how heterogeneous architectures compare to homogeneous architectures under said performance metrics; and how fundamental design choices, such as core type, cache size and off-chip bandwidth, affect performance. The design space for heterogeneous multi-cores is even larger than that of homogeneous multi-cores, as a single chip may contain any number of core types and various instances of each of those core types. Hence a very fast modeling methodology is required to gain insight into the design trade-offs. We use the MPPM framework, as described in the previous section, to explore the large heterogeneous multi-core architecture design space. The analytical model has linear-time complexity in the number of core types and programs of interest, and offers a unique opportunity for exploring the large space of both homogeneous and heterogeneous multi-core processors in limited time.

Contribution #2: Understanding Design Trade-offs in Heterogeneous Multi-Cores

Using MPPM, we did an extensive single-ISA heterogeneous multi-core design space exploration. We considered five different core types (ranging from a small scalar in-order core to a large 4-wide out-of-order core). We considered multiple last-level cache sizes and evaluated scenarios of varying off-chip bandwidth limits. Our analysis provides several interesting insights with respect to heterogeneous multi-cores:

- While many core types can be integrated for fine-grained program-to-core mapping, we show that two core types provide most of the benefits from heterogeneity. A larger number of core types does not contribute much.
- Heterogeneity fundamentally trades per-program performance for chip throughput.
- Some homogeneous configurations are optimal for particular through-

put versus per-program performance trade-offs.

- Job-to-core mapping is both crucial and challenging for heterogeneous multi-core processors to achieve optimum performance.
- Limited off-chip bandwidth alters some of the fundamental design choices in heterogeneous multi-core architectures.

This work has been published in ACM Transactions on Architecture and Code Optimization (TACO) and presented at the 2013 International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC):

> K. Van Craeynest and L. Eeckhout. Understanding Fundamental Design Choices in Single-ISA Heterogeneous Multi-core Architectures. ACM Transactions Architecture Code Optimization, Vol. 9, Issue 4, Article 32, Jan 2013

Scheduling Heterogeneous Multi-core Processors

One of the key insights from exploring the heterogeneous architecture design space as just described is, that there exists a fundamental challenge for heterogeneous multi-core processors regarding how to best schedule workloads on the most appropriate core type. Heterogeneous multi-cores can enable higher performance and reduced energy consumption (within a given power budget) if and only if workloads are executed on the most appropriate core type. Making wrong scheduling decisions can lead to suboptimal performance and excess energy/power consumption.

Finding an optimal (or even a good) schedule is non-trivial for a couple reasons. For one, applications tend to behave differently across core types. Certain application characteristics such as limited ILP might be limiters for performance on one core type (in this case, in-order cores) while on other core types (such as out-of-order cores) the impact is much less pronounced. What makes scheduling such a challenging problem is that ultimately, system performance is the result of a complex relationship between an application's characteristics and a core's features. To our knowledge, no single metric has been found that accurately describes this relationship.

In addition, a lot of workloads show time-varying behavior [Sherwood and Calder 1999, Hamerly et al. 2004] (application characteristics, and hence resource requirements, change over time). A good scheduling policy therefore needs to be dynamic: the application's characteristics should

be continuously monitored and the thread-to-core mapping needs to be changed accordingly. Yet another challenging aspect of scheduling is that heterogeneous multi-cores may contain a wide range of core types and any number of instances of those core types. This implies that scalability is a very important property of any scheduling algorithm. This is compounded by the fact that scheduling algorithms tend to work well for certain combinations of core types, but may yield suboptimal results for others. For example, a commonly used scheduling policy is to schedule threads that are memory-intensive on the smaller, less powerful core types because of power-efficiency. However, this strategy only works as expected when considering core types that do not differ significantly in their capabilities for exploiting parallelism. If they do, the loss of parallelism may outweigh power efficiency.

Contribution #3: Performance Impact Estimation Based Scheduling

We propose Performance Impact Estimation (PIE) scheduling which estimates which core type is likely to achieve the best performance. PIE collects basic performance characteristics from the application on the core that it is running on, and uses these as inputs to the analytical model to estimates performance if the application were to run on another core type. PIE adjusts the scheduling during runtime and thereby exploits fine-grained time-varying execution behavior. PIE provides a flexible and scalable solution because it scales easily with the number of cores and core types. Further, PIE incurs limited hardware support. We evaluate PIE scheduling and report improvements in system performance by 5.5% on average over memory-dominance scheduling (conventional wisdom) and by 8.7% over sampling-based scheduling for a set of scheduling-sensitive workload mixes.

This work has been published at the 2012 International Symposium on Computer Architecture (ISCA):

> K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez and J. Emer. Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE). Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA), pages 213-224, June 2012.

1.4 Fairness-Aware Scheduling

We primarily focused on optimizing system throughput while developing and evaluating PIE scheduling and little, not to say no, attention was given to fairness. Yet, guaranteeing that all threads make equal progress on heterogeneous multi-cores is of utmost importance for both multi-threaded and multi-program workloads to improve performance and quality-of-service. Furthermore, modern operating systems pin workloads to cores (pinned scheduling) which dramatically affects fairness on heterogeneous multi-cores.

Fairness is important for multi-program workloads when it comes to system-level priorities and quality-of-service (QoS): achieving high fairness means that all threads in the system are making equal progress proportional to isolated execution. Because a heterogeneous system then behaves as a homogeneous one, the same methods for achieving QoS in a homogeneous system can be utilized.

Fairness is also a desirable property for multi-threaded workloads because of thread synchronization. Threads running on a powerful core will typically make faster progress than threads running on a small, low-performance core. When there is need for synchronization (e.g. due to barriers), the fast running threads will stall at the barrier until all other threads running on the less powerful cores have reached the barrier — yielding no performance benefit from heterogeneity. Guaranteeing fairness, or making sure all threads make equal progress, will lead to all threads reaching the barrier at nearly the same time as they would on a homogeneous multi-core, thereby improving overall application performance.

Contribution #4: Fairness-aware Scheduling For Heterogeneous Multi-core Architectures

We propose fairness-aware scheduling for single-ISA heterogeneous multi-cores, and explore two flavors for doing so.

- Equal-time scheduling strives at achieving high fairness by running each thread or workload on each core type for an equal fraction of time. Equal-time scheduling leads to equal-progress scheduling in case co-executing threads benefit equally from running on the big core, because then, time equals work.
- When threads do not equally benefit from core heterogeneity (i.e.

some threads benefit more than others from running on a big core) , there is no one-to-one mapping of time and progress for the entire system. Therefore, equal-time scheduling will not necessarily yield high fairness. By continuously monitoring progress and by (continuously) changing the thread-to-core mapping to get equal progress for each thread in the system, equal-progress scheduling guarantees higher fairness compared to equal-time scheduling.

Fairness-aware scheduling not only improves fairness over pinned scheduling (as done in current operating systems to improve locality in multi-core processors), it also improves system throughput by enabling threads to run on a big core for some fraction of time. Our experimental results demonstrate an average 14% (and up to 25%) performance improvement over pinned scheduling through fairness-aware scheduling for homogeneous multi-threaded workloads; equal-progress scheduling improves performance by 32% on average for heterogeneous multi-threaded workloads.

This work is submitted to the 2013 International Symposium on Computer Architecture (ISCA):

K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel and L. Eeckhout. Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores. Submitted for the 39th Annual International Symposium on Computer Architecture (ISCA), June 2013.

1.5 Overview

This dissertation is organized as follows. Each contribution has its own dedicated chapter. In Chapter 2, we will explain in much detail the inner workings of the MPPM framework. In Chapter 3, we will expand on this and leverage the MPPM framework for exploring the heterogeneous multi-core design space, providing key insights regarding fundamental design decisions. Chapter 4 focuses on the PIE model and uses it to create a scheduling algorithm that optimizes system throughput. In Chapter 5 we show the importance of fairness in heterogeneous multi-core systems and propose and evaluate fairness-aware scheduling policies. Finally, we conclude and provide hints towards future work in Chapterr̃efchapter:fw.

Chapter 2

The Multi-Program Performance Model

Don't buy upgrades, ride up grades.
Eddy Merckx

In this chapter, we propose the Multi-Program Performance Model (MPPM), a method for quickly estimating multi-program multi-core performance based on single-core simulation runs. MPPM employs an iterative method to model the tight performance entanglement between co-executing programs on a multi-core processor with shared caches.

2.1 Introduction

Simulation and modeling are at the foundation of processor design and computer architecture research, i.e., research and development is driven by the careful evaluation of design alternatives in order to make correct design decisions. A key aspect of experimental evaluation is the workload that serves as input to simulation and modeling. A workload that is unrepresentative of a processor's target workload may lead to a suboptimal design, hence, it is absolutely crucial to have a representative workload.

Building a representative workload is very challenging, especially for multi-core processors. A multi-core processor has multiple hardware thread contexts, and each hardware thread context can execute a different program. As a result, a multi-core processor workload may consist of a mix of multiple independent programs. In fact, multi-program workloads are a very important and significant fraction of today's workloads.

Given the limited amount of multi-threading in current desktop applications [Blake et al. 2010], multi-program workloads are predominant in today's computer systems, including mobile devices, laptops, desktops and even servers.

Evaluating multi-program workloads is non-trivial though. For a given number of programs, the number of multi-program workloads quickly explodes. For N programs and M hardware contexts, there are M combinations with repetition out of N programs, or $\binom{N+M-1}{M}$ multi-program workloads in total. This means there are 435 possible multi-program workload mixes for 29 SPEC CPU2006 benchmarks on a dual-core processor; 35,960 workload mixes for a quad-core processor; and more than 30.2 million workload mixes for an eight-core processor. Assuming detailed cycle-accurate processor simulation at a speed of 300 KIPS and assuming 1 B instruction workloads — as in our setup — this would result in 54 days for simulating all possible two-program workloads. For four- and eight-program workloads, total simulation time would need to be counted in years. Clearly, simulating all possible multi-program workloads using detailed simulation is completely infeasible in practice.

Hence, current practice in computer architecture research and development is to pick a limited number of multi-program workload mixes, typically a dozen or a couple tens that are randomly chosen. Often, architects compose classes of multi-program workload mixes with each class representing a particular set of multi-program workloads. For example, one class may comprise combinations of memory-intensive programs, another class may comprise mixes of compute-intensive and memory-intensive programs, and yet another class may comprise a set of compute-intensive workloads. Within each class, architects then select a number of random multi-program workload mixes. It is unclear though whether a limited number of randomly chosen multi-program workloads is representative for the very large set of multi-program workloads.

In this chapter, we propose the Multi-Program Performance Model (MPPM), a method for quickly estimating multi-program multi-core performance from single-core simulation runs; this allows for quickly estimating multi-core performance for a large number of multi-program workloads in a reasonable amount of time. We collect a profile during single-core simulation that captures a program's memory behavior as well as its phase behavior; this is a one-time cost. We then employ an iterative method that models the performance entanglement between the co-executing programs on a multi-core processor with shared caches: the it-

erative method captures how per-program performance affects the amount of resource sharing, and, vice versa, how resource sharing in its turn affects per-program performance. Since this iterative method involves an analytical model, it is very fast, while being accurate. Using this powerful technique, we demonstrate that current practice of simulating a limited number of multi-program mixes may lead to incorrect design decisions. Instead, we advocate MPPM which allows for evaluating performance for a very large number of multi-program workload mixes in a reasonable amount of time. The method can also be used to identify workload mixes that stress multi-core performance due to excessive conflict behavior in shared resources.

More specifically, we make the following contributions.

- We propose the Multi-Program Performance Model (MPPM) for estimating multi-core processor performance for multi-program workloads. MPPM uses single-core simulation profiles and estimates multi-core processor performance while taking into account resource sharing in shared caches when running multi-program workloads. The performance entanglement between co-executing programs due to resource sharing is solved through an iterative approach that estimates the amount of resource sharing and how it affects per-core performance, and vice versa. We report an average performance prediction error of 2.3% and 2.9% for system throughput (STP) and average normalized turnaround time (ANTT), respectively, compared to detailed simulation across SPEC CPU2006 using the x86 CMP$im simulator [Jaleel et al. 2008a] and up to 16 cores.

- We demonstrate that MPPM can quickly estimate multi-core processor performance from single-core simulation runs, which enables estimating multi-core performance for a large number of multi-program workload mixes in a reasonable amount of time. Indeed, the key feature of the proposed method is that it decouples per-core simulation from multi-core simulation, yielding a multi-core processor simulation and modeling approach for multi-program workloads with only linear time complexity in the number of programs. Our method is shown to be up to five orders of magnitude faster than detailed multi-core processor simulation.

- We demonstrate that current practice of randomly choosing a limited number of multi-program workloads may lead to incorrect design decisions. Instead, a more accurate approach is to use MPPM

for evaluating all, or at least a very large number of, multi-program workload mixes. In addition, MPPM provides confidence bounds on its performance estimates.

- We demonstrate that MPPM can identify multi-program workloads that yield very poor multi-core processor performance due to excessive conflict behavior in shared resources. Architects can then focus on these stress workloads and fine-tune the design to yield better performance.

This chapter is organized as follows. Section 2.2 presents MPPM in great detail, after which we present the experimental setup in Section 2.3. We evaluate MPPM in Section 2.4, debunk current practice in Section 2.5, and use MPPM for identifying multi-program stress workloads in Section 2.6. Finally, we discuss related work and how MPPM differs from prior work in this area in Section 2.7, and we conclude Section 2.8.

2.2 Multi-Program Performance Model

Figure 2.1 gives a general overview of the Multi-Program Performance Model (MPPM). We first perform single-core simulation profiling runs for all the benchmarks in the benchmark suite. This is a one-time cost. Once these single-core profiles are collected, they serve as input to the multi-program performance model. In other words, MPPM estimates multi-program performance from single-core simulation profiling runs. Because the single-core simulation runs are a one-time cost only, and because MPPM is an analytical approach, the proposed method is very effective at estimating multi-program performance on multi-core processors. In fact, estimating multi-program performance for an arbitrary set of benchmarks is done very quickly — typically around a few hundred milliseconds per multi-program workload — hence, MPPM enables predicting multi-core performance for a large set of multi-program workloads in a reasonable amount of time. As an example, MPPM estimates multi-core performance for 5,000 four-program workloads in half an hour.

MPPM assumes a particular multi-core processor architecture of interest for the single-core simulation runs. In other words, if one were to predict performance for a multi-core processor with out-of-order processor cores and a cache hierarchy with three levels of cache, this same or derived processor architecture needs to be considered during the single-core simulations as well. This means that the single-core simulations need

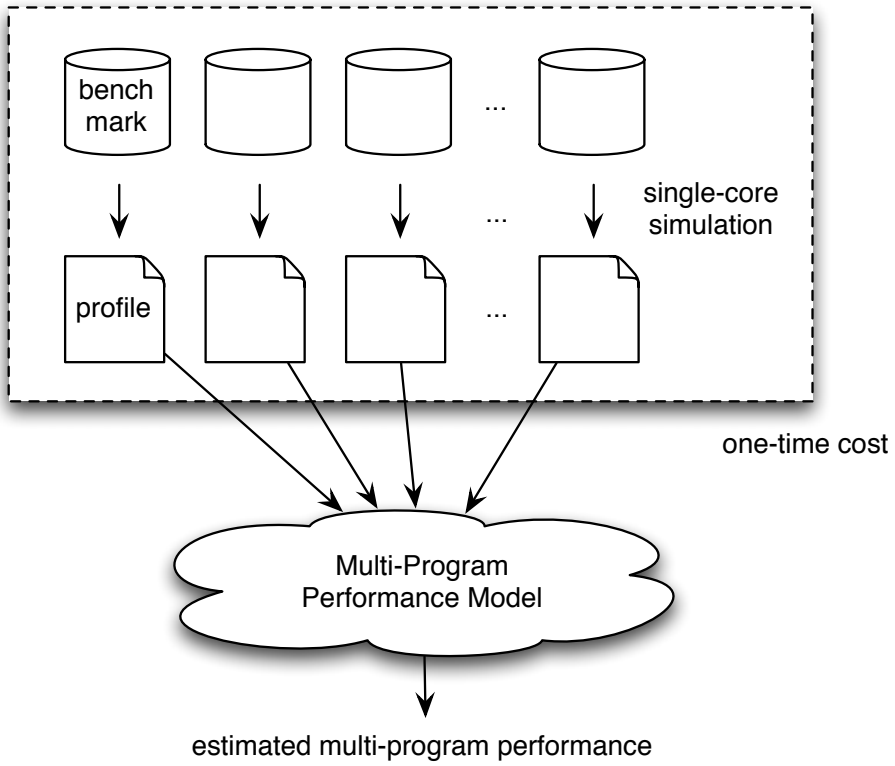


Figure 2.1: General overview of MPPM.

to be performed with the same core architecture and the same or derived cache hierarchy, however, the benchmark is run in isolation, i.e., there are no co-executing programs. (In fact, in our setup, we can run single-core simulations and derive performance models for cache hierarchies with reduced associativity at each level of the hierarchy. For example, we can run single-core simulations and collect the single-core profiles for a 16-way set-associative cache, and derive single-core simulation profiles for an 8-way set-associative cache without having to run additional single-core simulations.) Although this may seem like a limitation of MPPM — single-core simulation runs need to be performed for different multi-core architectures of interest — it is not a major limitation in practice: it requires single-core simulation runs only. There are no time-consuming multi-core simulations required, and the MPPM model makes multi-core performance predictions quickly for a large set of multi-program workloads. Further, once the single-core simulation profiles are obtained, MPPM can estimate performance for a varying number of cores, different cache associativities, and different combinations of co-executing programs very quickly.

We now detail on the two major steps in MPPM: single-core simulation profiling and the performance model itself.

2.2.1 Single-core Simulation Profiling

Single-core simulation profiling collects three characteristics:

- **Single-core CPI** is the number of Cycles Per Instruction (CPI) when running the single-core workload in isolation, i.e., there are no co-executing programs. CPI is easily obtained by dividing cycle count with the number of dynamically executed instructions.
- **Memory CPI** is the fraction of the single-core CPI waiting for memory. There are two ways for computing the memory CPI. One way is to employ the counter architecture proposed by Eyerman et al. [2006] for computing CPI stacks on out-of-order processors; implementing this counter architecture in the simulator enables computing the memory CPI component from a single simulation run. Alternatively, the memory CPI can be computed from two simulation runs: one run with a perfect Last-Level Cache (LLC), i.e., all accesses to the LLC are hits and there are no memory accesses, versus one run with an imperfect LLC, i.e., LLC misses go off to memory. The CPI obtained from the latter minus the CPI obtained from the former then is the memory CPI.

- **Stack Distance Counters (SDCs)** capture a program's temporal memory access pattern in set-associative (or even fully associative) caches [Mattson et al. 1970]. We collect SDCs for each program on the LLC without cache sharing, i.e., by running the program in isolation. An SDC for an A-way set-associative cache involves $A + 1$ counters, $C_1, C_2, \ldots, C_A, C_{>A}$, and is computed as follows. On each access, one of the counters is incremented. If the access is to the ith position in the LRU stack for that set, the ith counter C_i is incremented. If the cache access involves a miss, then the $C_{>A}$ counter is incremented.

Each of these performance characteristics are measured on a per-interval basis. The reason is to be able to model the impact of time-varying phase behavior on resource contention in multi-core processors. In our setup, we measure these characteristics for every interval of 20 million (dynamically executed) instructions. For a 1 billion instruction trace, this implies 50 intervals in total per benchmark with the above characteristics measured for each interval. This is done for each benchmark in the benchmark suite.

2.2.2 MPPM

The single-core performance characteristics as mentioned in the previous section then serve as input to the Multi-Program Performance Model (MPPM). The concept of the MPPM is to initially start from the single-core performance measurements and then iteratively converge on how resource contention in shared resources affects per-core performance in a multi-core processor. The reason for the iterative process is the tight performance entanglement between per-core performance and resource contention, i.e., per-core performance affects the amount of resource contention, and vice versa, resource contention affects per-core performance. In order to model this tight performance entanglement, the model initially estimates the amount of resource contention assuming each program makes progress as per the single-core simulations; however, the amount of resource sharing affects per-core progress, which in its turn affects resource sharing. Hence, in the next iteration, per-core progress is adjusted to incorporate how resource contention affects per-core progress. This, in its turn, may again affect the amount of resource contention seen, which leads to the second iteration, etc. This iterative process continues until convergence.

Figure 2.2 gives a schematic overview of the MPPM model. We define R_p as the slowdown for a program p in the multi-program workload mix

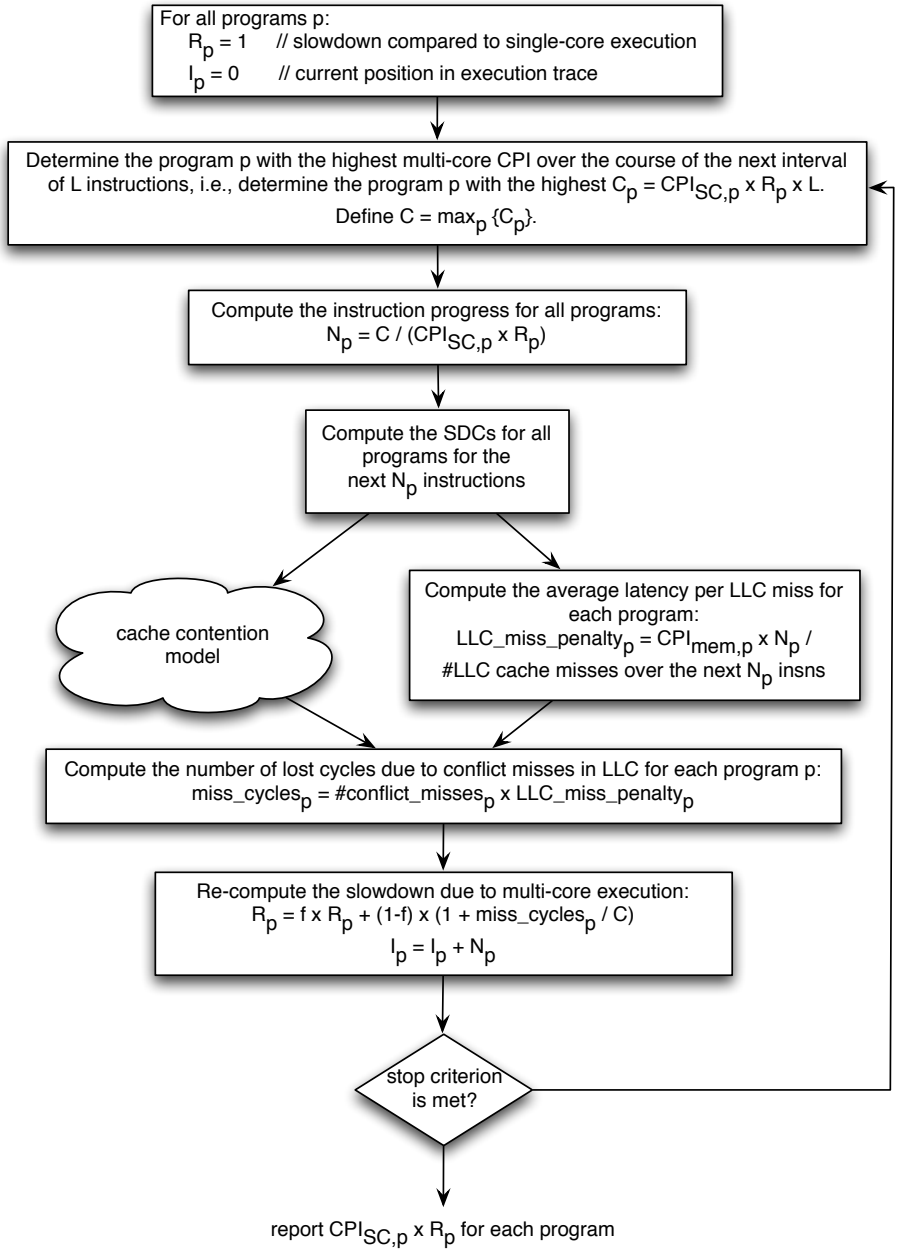


Figure 2.2: The Multi-Program Performance Model.

relative to isolated execution. We assume all programs experience the same relative slowdown of $R_p = 1$ and execute at single-core speed initially. Further, we assume that all programs start at the beginning of the execution trace, i.e., the instruction pointer is set to zero: $I_p = 0$. Once these initial conditions are set, the iterative process starts.

At each step in the iterative process, we first determine the slowest program in the workload mix, or the program in the multi-program workload mix with the highest multi-core CPI over the next L instructions. L is 200 M instructions in our setup. A program's multi-core CPI is computed as the single-core CPI ($CPI_{SC,p}$) multiplied with its relative slowdown R_p. We define C to be the number of cycles it takes for the slowest program to execute L instructions. We then determine how much progress each program in the workload mix can make during the next C cycles. We define N_p as the number of instructions each program p can execute during the next C cycles.

Once we know how many instructions each program will execute during the next time interval of C cycles, we can compute the SDCs for each of the programs over this time interval. This is done by adding the per-interval SDCs. As mentioned in the previous section, the single-core performance characteristics are measured on a per-interval basis of 20 M instructions. Computing the SDCs for the next time interval of C cycles is done by simply adding the per-interval SDCs for the next N_p instructions. The SDCs serve two needs. First, it serves as input to a cache contention model that estimates the additional number of conflict misses due to cache sharing in the LLC. There exist several cache contention models [Chandra et al. 2005, Eklöv et al. 2011, Lee et al. 2008]. We use the Frequency of Access (FOA) model proposed by Chandra et al. [2005] because it is a fairly simple model and we found it to be accurate enough for our needs. Second, the SDCs allow for estimating the average penalty per LLC miss. This is done by dividing the number of cycles lost due to memory accesses with the number of LLC misses; we assume here that the average LLC miss penalty is the same under multi-core execution versus single-core execution. The number of cycles waiting for memory is computed as the memory CPI component ($CPI_{mem,p}$) multiplied with the number of instructions in the next time interval N_p. The number of LLC misses is obtained from the SDC's $C_{>A}$ counter, as mentioned above. We estimate the number of cycles lost due to conflict misses in the LLC by multiplying the number of additional conflict misses due to cache sharing (obtained from the cache contention model) and the average penalty per LLC miss (computed as de-

scribed above).

We can now estimate the (current) relative slowdown for each program in the workload mix due to resource contention. This is done using an exponential moving average of the average slowdown observed so far and the current slowdown according to the above model. The reason for taking an exponential moving average is to include a smoothing effect, which we found to be important for achieving good accuracy, especially for programs with significant time-varying execution behavior. We also compute the current position in the execution for each program. This is done by simply advancing the instruction pointer by N_p instructions for each program.

This iterative process is repeated multiple times until a stop criterion is met. Each iteration involves 200 M instructions for the slowest running program, and the iterative process continues until the slowest running program in the workload mix has executed 5 B instructions in total. Given that our instruction traces are 1 B instructions in size, this means that the slowest program needs to iterate over its entire trace five times. Faster running programs may iterate over their trace more than five times. We found that the performance numbers converged given this stop criterion.

2.2.3 Discussion

We want to emphasize again that MPPM itself does not involve detailed cycle-accurate multi-core simulation. The process as explained above only involves 'analytical' simulation in which we employ analytical models for estimating multi-core performance. This yields a very fast multi-core performance estimation technique: MPPM makes a multi-core performance estimate in less than one second, provided that the single-core simulation runs were done beforehand.

It is also important to stress that MPPM is independent of the cache replacement and/or partitioning strategy employed in the shared cache. In fact, the cache contention model is an integral part of the approach, and if the cache contention model supports multiple cache replacement and/or partitioning strategies, so does MPPM. The cache contention model used in this chapter is the FOA model [Chandra et al. 2005], as mentioned before. FOA assumes that the effective cache space for a program is proportional to its access frequency. The intuition is that a program that has a high access frequency tends to bring in more data into the cache, and hence it effectively occupies a larger fraction of the caches. We found FOA to be accurate enough for our purpose.

ROB	128 entries
pipeline	8-stage, 4-wide
ld/st	max of two loads & one store per cycle
L1 I-cache	32 KB, 4 WSA, LRU, 1 cycle
L1 D-cache	32 KB, 8 WSA, LRU, 1 cycle
L2 cache	private, 256 KB, 8 WSA, 10 cycles
L3 cache	shared, see Table 2.2
memory	200 cycles
branch prediction	perfect

Table 2.1: Baseline processor configuration.

	size	*assoc*	*latency*
config #1	512 KB	8	16
config #2	512 KB	16	20
config #3	1 MB	8	18
config #4	1 MB	16	22
config #5	2 MB	8	20
config #6	2 MB	16	24

Table 2.2: Last-level cache (LLC) configurations.

2.3 Experimental Setup

We use a multi-core processor simulator based on CMP$im [Jaleel et al. 2008a], which is an x86 simulator built on top of Pin [Luk et al. 2005]. CMP$im is a user-level simulator and allows for simulating multi-core processor architectures. The version of the CMP$im simulator that we used for the work done in this chapter is the one available from the Cache Replacement Championship[1]; CMP$sim is not publicly available in any other form. The processor architecture that we simulate is detailed in Tables 2.1 and 2.2. We consider 4-wide out-of-order processor cores with private L1 instruction and data caches. Each core has a private L2 cache. The L3 cache is shared among the cores and is the last-level cache (LLC) in our setup: we apply our method to the L3 cache. All caches implement an LRU replacement policy. We consider different LLC configurations, as shown in Table 2.2. If not explicitly mentioned, we report performance results for configuration #1 which has the smallest LLC; this is to stress our model. This does not limit the generality of the proposed method; our method can be applied to any (shared) level in the cache hierarchy.

We consider all the SPEC CPU2006 benchmarks with their reference inputs. All the benchmarks were compiled with the GNU C compiler version

[1] `http://www.jilp.org/jwac-1/`

4.3.4 and optimization level `-O2`. We use SimPoint [Sherwood et al. 2002] to pick representative simulation points of one billion instructions each.

When quantifying multi-core processor performance we consider two performance metrics, namely system throughput (STP) and average normalized turnaround time (ANTT) [Eyerman and Eeckhout 2008]. STP measures multi-core performance from a system perspective and quantifies the accumulated progress by all the programs in the multi-program workload mix. STP equals weighted speedup by Snavely and Tullsen [2000], and is a higher-is-better metric:

$$STP = \sum_{p=1}^{n} \frac{CPI_{SC,p}}{CPI_{MC,p}}.$$

ANTT focuses on user-perceived performance and quantifies the average slowdown during multi-core execution relative to single-core, isolated execution. ANTT is the reciprocal of the hmean metric proposed by Luo et al. [2001]:

$$ANTT = \frac{1}{n} \sum_{p} \frac{CPI_{MC,p}}{CPI_{SC,p}}.$$

2.4 Model Evaluation

We evaluate MPPM along two criteria: accuracy and speed. However, before doing so, we first quantify performance variability across random sets of multi-program workloads — this will demonstrate that obtaining tight confidence bounds requires a sufficiently large number of workload mixes.

2.4.1 Variability

Figure 2.3 shows the variability in STP and ANTT as a function of the number of random sets of multi-program workloads on a four-core processor. These graphs clearly illustrate that selecting a limited number of random multi-program workloads yields limited confidence in the overall performance measurements. For example, selecting 10 workload mixes yields a 10% and 18% confidence interval for STP and ANTT, respectively. Doubling the number of workload mixes to 20 does not increase confidence dramatically: the confidence intervals are still around 7% and 13% for STP and ANTT, respectively. Such confidence intervals may be too large for many practical research and design studies. Comparing design alternatives that

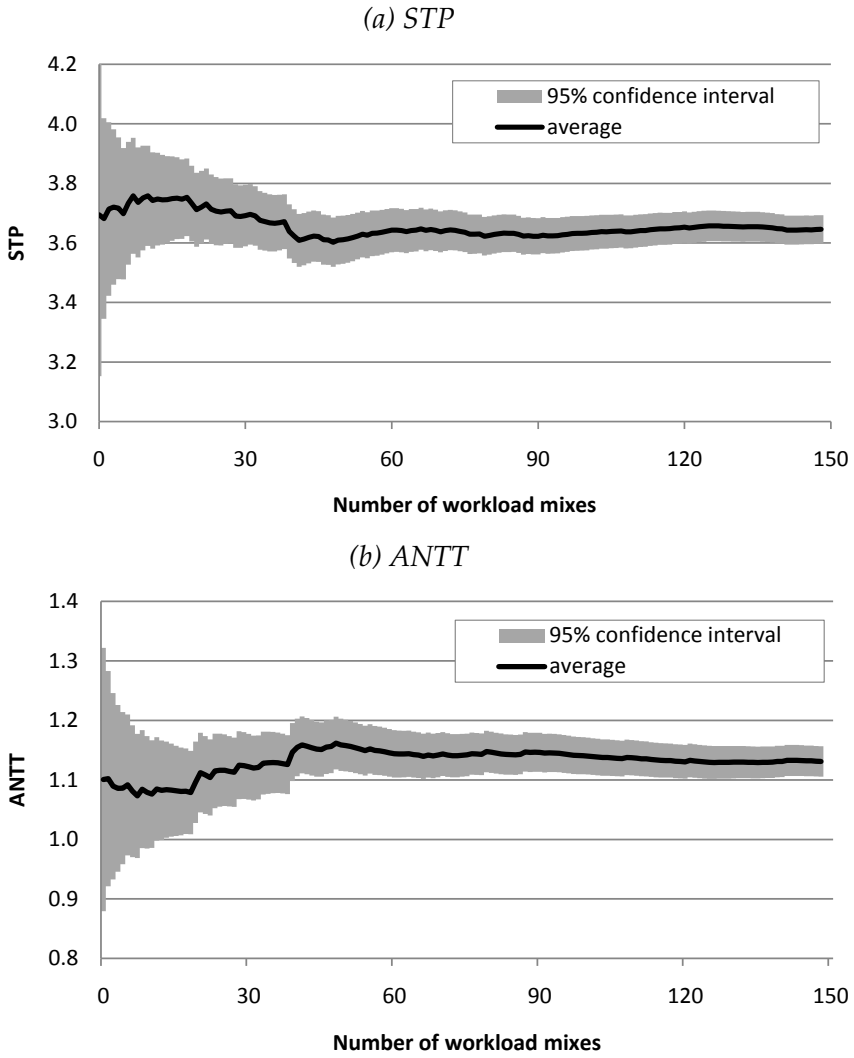


Figure 2.3: Variability in (a) STP and (b) ANTT as a function of the number of multi-program workload mixes.

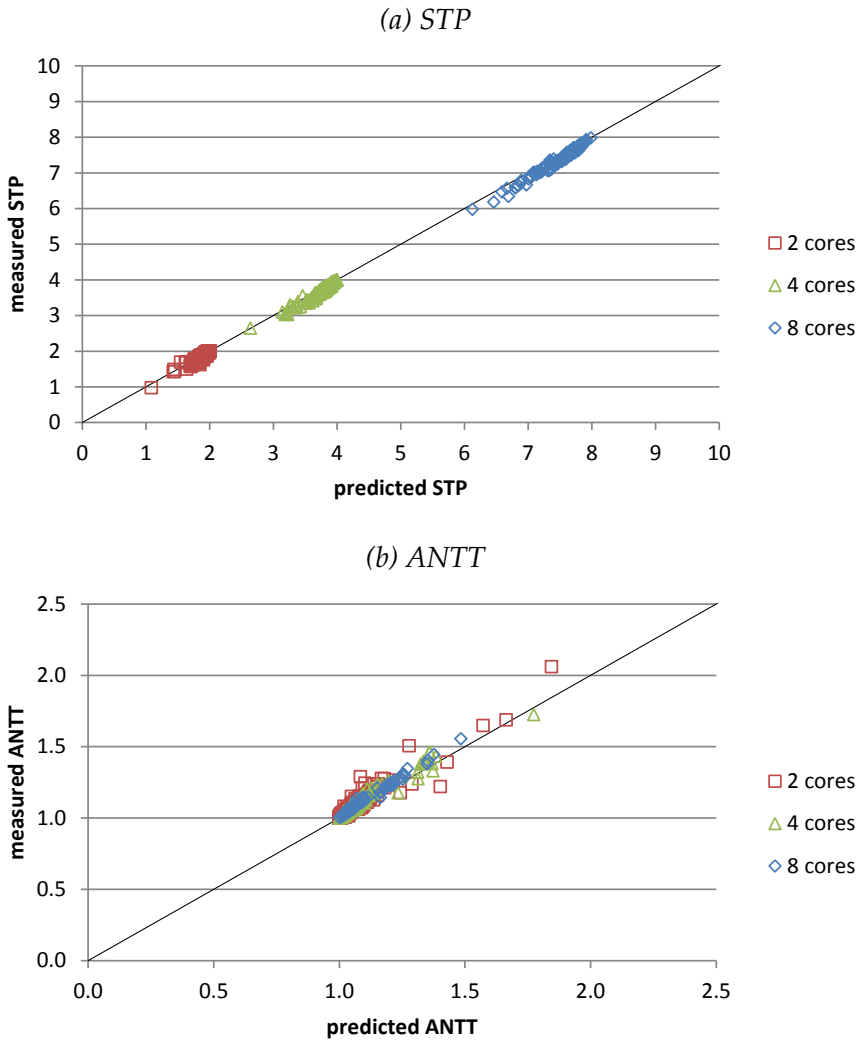


Figure 2.4: Accuracy of MPPM for predicting (a) STP and (b) ANTT; measured STP/ANTT on vertical axis versus predicted STP/ANTT on the horizontal axis.

differ in the percent range with a performance evaluation method with this large confidence intervals may be problematic. At 150 workload mixes, the confidence bounds are down to 2.6% and 4.5% for STP and ANTT, respectively; hence, we report performance numbers for 150 workload mixes in the remainder of the chapter.

2.4.2 Accuracy

Figure 2.4 shows scatter plots for STP and ANTT. These graphs assume the baseline processor configuration with 2, 4 and 8 cores, and 150 multi-program workload mixes. We simulate these workload mixes through detailed simulation using CMP$im; this yields the 'measured' STP and ANTT metrics. We also estimate multi-core performance using MPPM which yields the 'predicted' metrics. The scatter plots show the predicted metrics versus the measured metrics. Each dot represents one of the workload mixes. Perfect prediction would imply all dots to lie on the bisector. We observe a strong correlation between the measured and predicted performance metrics, i.e., all the dots lie around and are close to the bisector. The average error across these workload mixes equals 1.4%, 1.6% and 1.7% for STP and 2, 4 and 8 cores, respectively; and 1.5%, 1.9% and 2.1% for ANTT and 2, 4 and 8 cores, respectively.

We also ran a number of experiments for 16 cores using 25 16-program workload mixes; here, we consider a larger 1 MB LLC (configuration #4). We were unable to run more than 25 16-program workload mixes because of time constraints — the simulations took extremely long, which is exactly the problem we are addressing with MPPM. The average error equals 2.3% and 2.9% for STP and ANTT, respectively.

The fact that MPPM is accurate for predicting STP and ANTT suggests that it is also effective for predicting the relative per-program slowdown, or by how much a program gets slowed down due to multi-program execution on a multi-core processor. Figure 2.5 reports both the measured and the predicted relative slowdown for each program. Correlation is good and the average error equals 7% across the 150 workloads for 2, 4 and 8 cores; for the 25 multi-program workloads on the 16-core processor (not shown in Figure 2.5), we obtain an average error of 4.5%. Figure 2.6 makes this more concrete and shows an example for the 4-program workload with the worst STP; this workload consists of two copies of gamess along with hmmer and soplex. The gamess copies are slowed down substantially through multi-core execution (more than 2×), whereas soplex is slowed down somewhat

only, and hmmer is barely affected by multi-core execution. MPPM predicts these slowdowns accurately.

It is worth noting that the error for predicting per-program performance, although low, is larger than the error for predicting STP and ANTT. The reason is that STP and ANTT measure total system performance and average per-program performance, respectively, and given how STP and ANTT are computed, see Section 2.3, positive and negative errors in predicting per-program performance get smoothed, which leads to more accurate overall STP/ANTT predictions.

2.4.3 Speed

MPPM is substantially faster than detailed simulation. As mentioned before, MPPM requires single-core simulations, but this is a one-time cost only. In our setup with 1 B instruction simulation points this takes around 1 hour of simulation time per benchmark on CMP$im, or slightly more than an entire day for the entire SPEC CPU2006 benchmark suite on a single machine. The MPPM model itself is very fast because it does not involve detailed simulation. Instead, it uses analytical modeling for estimating multi-core performance. MPPM takes less than one second per multi-program workload; typically, a couple tenths of seconds. In contrast, simulating a multi-program workload on a detailed cycle-accurate simulator is extremely time-consuming. For example, simulating one multi-program workload for an eight-core processor takes around 12 hours in our setup. In summary, depending on the scenario, MPPM is up to 5 orders of magnitude faster than detailed simulation. Considering 150 multi-program workloads on an 8-core processor, MPPM (including the cost of single-core simulations) is 62$\times$ faster than detailed simulation. Assuming that the single-core simulations were done beforehand, MPPM is more than 53,000$\times$ faster.

2.5 Debunking Current Practice

Now that we have demonstrated that MPPM is both accurate and fast for estimating multi-program workload performance on multi-core processors, we leverage MPPM to evaluate (and debunk) current practice in simulating multi-program workloads. One of the critical concerns when simulating multi-program workloads is which workloads to pick out of the very large set of possible multi-program workloads. Current practice is to pick a limited number of multi-program workloads. The reason for

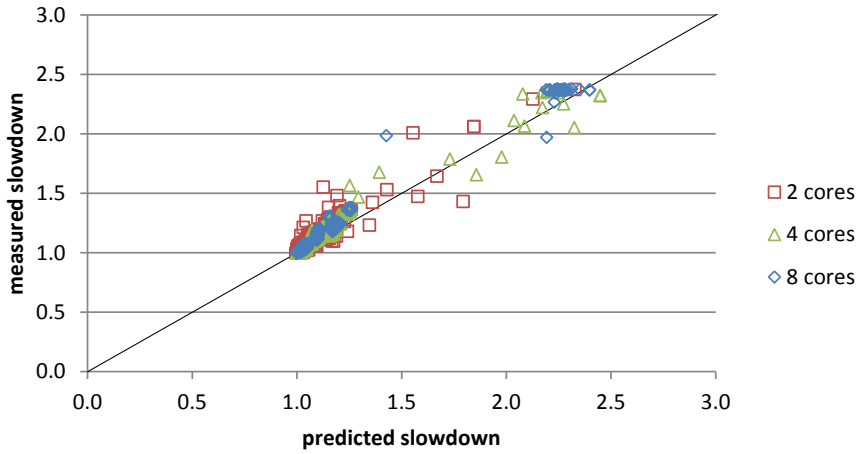


Figure 2.5: Measured versus predicted relative per-program slowdown due to multi-core execution.

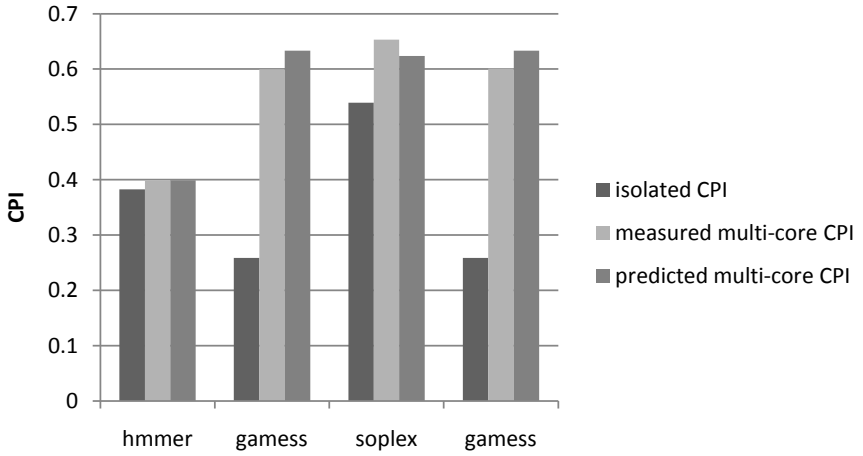
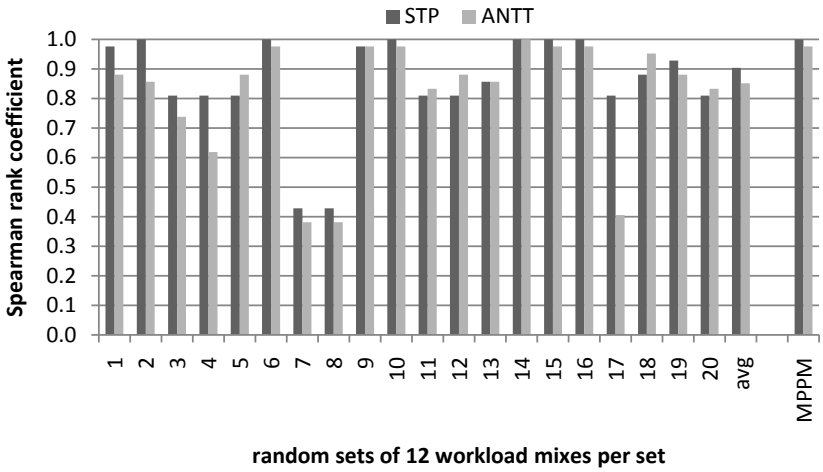


Figure 2.6: Tracking the performance of individual programs in a multi-program workload consisting of two copies of gamess along with hmmer and soplex: isolated execution CPI, measured multi-core execution CPI (through simulation), and predicted multi-core execution CPI (through MPPM).

(a) Random



(b) Random per category

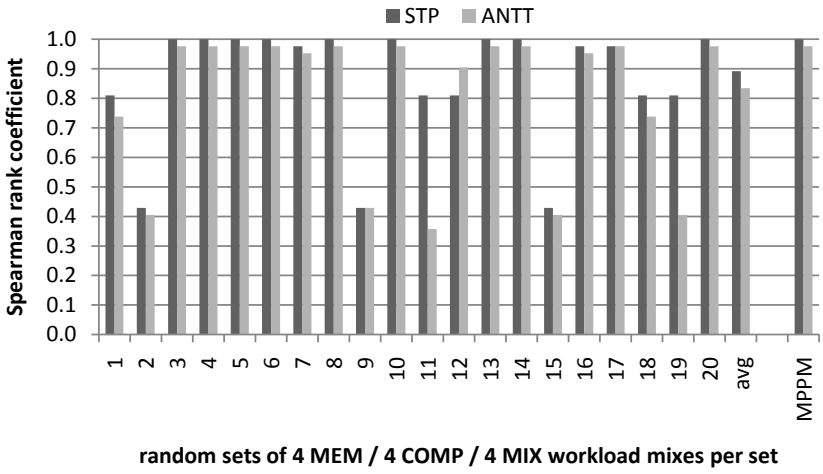


Figure 2.7: Evaluating current practice of selecting random workload mixes: Rank correlation coefficient for 20 sets of 12-program workloads versus MPPM. (a) Random selection of programs; and (b) Random selection of programs within program categories.

limiting the number of workloads is that simulation is extremely time-consuming. Hence, researchers pick a limited number of workloads at random out of the very large set of possible multi-program workloads. Alternatively, researchers often classify their benchmark programs in a number of classes, e.g., compute-intensive versus memory-intensive programs, and then randomly pick multi-program workloads from these classes to form multi-program workload categories. For example, one category may be workload mixes consisting of compute-intensive programs only; another category may be workload mixes with memory-intensive programs only; a third category may contain mixed compute- and memory-intensive programs.

To evaluate whether current practice is adequate, we consider the following setup. We compare six multi-core configurations that differ in their LLC configuration in terms of size, associativity and access time, see Table 2.2. Without detailed experimental evaluation, it is unclear which configuration yields the best overall performance. For example, configuration #2 has a higher associativity than configuration #1, and thus a lower miss rate, but its access latency is also higher. Similar trade-offs are possible between all pairs of configurations. So, it is unclear which configuration yields the best overall performance without detailed analysis. We now evaluate how well current practice and MPPM can rank these six configurations. The results are shown in Figure 2.7. These graphs show the rank correlation coefficients for current practice assuming 12 randomly selected multi-program workloads on a quad-core processor. This experiment is repeated 20 times, hence the 20 bars on the top graph of the two graphs in Figure 2.7. The second but last bars on the bottom graph, labeled 'avg' reports the average correlation coefficient for current practice. We then compare against MPPM while considering 5,000 multi-program workloads, see the right-most bars in Figure 2.7. The Spearman rank correlation coefficient quantifies how well two rankings compare to each other, or more formally, how well the relationship between two rankings can be described using a monotonic function; a Spearman rank correlation coefficient of one means a perfect match in the rankings. Our point of reference is the ranking obtained from detailed simulation with 150 multi-program workloads. MPPM is clearly more accurate than current practice. For some of the randomly picked workload mixes, the rank correlation coefficient is as low as 0.5 and below, see for example mixes 7 and 8 in Figure 2.7(a) and mixes 2, 9 and 15 in Figure 2.7(b). MPPM on the other hand achieves a rank correlation coefficient of 1 and 0.93 for STP and ANTT, respectively.

In order to make it more concrete we now pairwise compare design points, namely configuration #1 versus all the other configurations #2 through #6. Figure 2.8 quantifies how often current practice (assuming multi-program categories) disagrees with MPPM, and, when they disagree, how often MPPM is correct compared to the reference of detailed cycle-accurate simulation of 150 multi-program workloads, and thus by consequence, how often current practice leads to incorrect conclusions. In the most extreme comparison between configuration #1 and #6 we observe that in approximately 40% of the cases current practice disagrees with MPPM, and current practice leads to an incorrect conclusion with respect to which configuration yields the best performance.

These experiments collectively illustrate that current practice of selecting a limited number of multi-program mixes may lead to misleading and incorrect conclusions in practical research and design studies. MPPM on the other hand leads to correct conclusions because it considers a very large number of multi-program mixes. In addition, MPPM does so in only a small fraction of the time needed through current practice.

2.6 Identifying Stress Workloads

As mentioned before, MPPM's unique ability is to quickly estimate multi-program performance on multi-core processors. One important application of the MPPM framework is to identify workload mixes that stress the multi-core processor in the sense that performance is substantially lower for these workloads compared to the other workloads. Once these stress workloads are identified, they can be analyzed in more detail in order to understand why the multi-core processor fails to deliver good performance for these workloads, which may ultimately lead to an improved design. We find MPPM to be accurate in ranking the multi-program workloads with respect to how severe these workloads stress the processor architecture. Figure 2.9 shows sorted STP values obtained through detailed simulation and MPPM; more precisely, the workloads on the horizontal axis are sorted by increasing value of the STP values obtained through detailed simulation. This graph once again illustrates that MPPM is accurate compared to simulation, and in addition, it illustrates that MPPM can identify the worst-case workloads. MPPM is able to identify the top-23 worst-case workloads out of the 25 worst-case workloads obtained through detailed simulation. Further analysis showed that one particular benchmark, namely gamess, is very sensitive to multi-core execution: we found that gamess gets a slow-

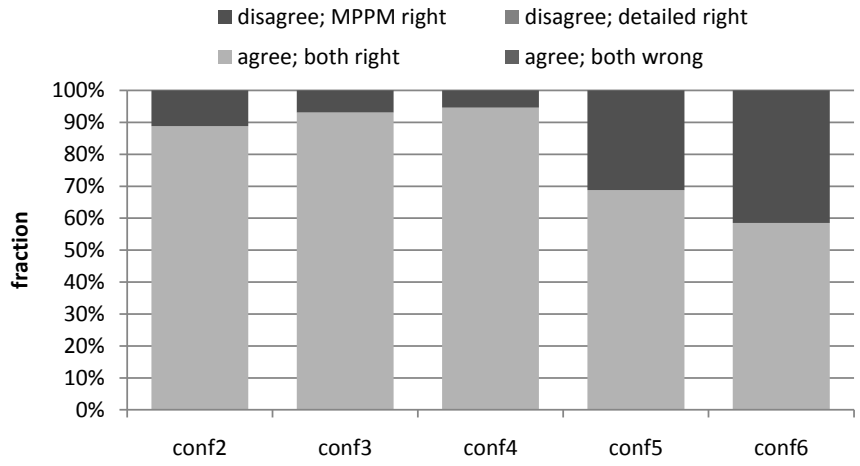


Figure 2.8: Fractions of when current practice agrees or disagrees with MPPM, and when MPPM is correct and current practice is not.

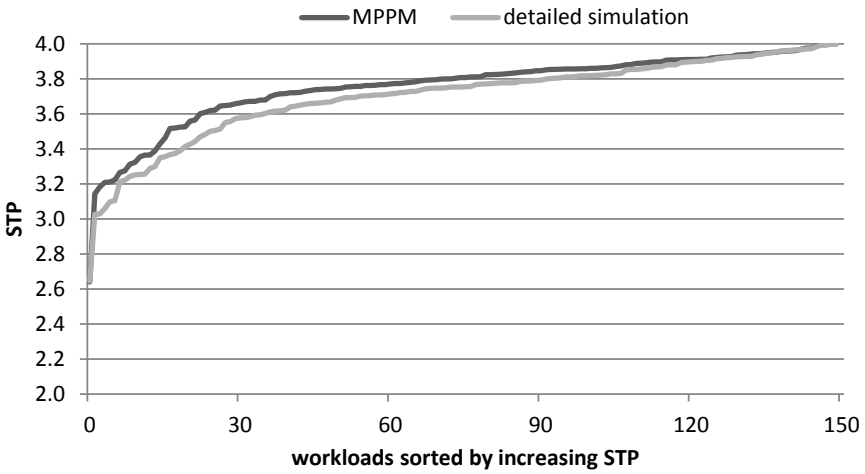


Figure 2.9: Identifying 4-program workloads with the worst STP.

down by a factor 2.2×, whereas the other benchmarks experience a slowdown of at most 1.3× (gokmk) and 1.2× (soplex, omnetpp, h264 and xalan); the remaining benchmarks are less sensitive to cache sharing.

2.7 Related Work

There exists some work in simulating and modeling multi-program workloads on multicore processors. These techniques differ from MPPM in several ways, as decribed below.

The work most similar to ours is the work by Eklöv et al. [2011]. They present StatCC which is a cache contention model that, given the relative CPIs of the co-scheduled programs and their individual reuse distance distributions, estimates the reuse distance distribution of the interleaved access stream to the shared cache. The reuse distance is defined as the number of memory references between two consecutive accesses to the same cache line. Once they have the reuse distance distribution of the interleaved access stream, they leverage their prior StatCache [Berg and Hagersten 2005] and StatStack [Eklöv and Hagersten 2010] work to estimate the cache miss rates for the co-scheduled programs. They use an equation solver to solve the interdependence between the programs' CPIs and cache miss rates. There are a number of key differences between Eklöv et al.'s approach and this work. We use an iterative method for solving the CPI versus cache miss rates interdependence while taking into account time-varying workload behavior, instead of the general-purpose equation solver by Eklöv et al. In addition, we use a stack distance counter distribution instead of a reuse distance distribution. Finally, we evaluate our approach for up to 16 cores; Eklöv et al. limited their evaluation to two cores only. Additional work has been published since the original publication of the MPPM model in IISWC 2011. Most recently, Sandberg et al. [2013] have proposed a methodology where they also used an iterative analytical model to predict the impact on multi-core performance due to cache sharing. Their approach differs from ours in that they collect profile information (input for the analytical model) using actual hardware. Also, they employ their model to quickly evaluate multi-core performance variation due to small differences in thread interleaving.

Lee et al. [2008] use (spline-based) regression modeling to build multicore processor performance models. They build three regression models: a core model, a contention model for the shared resources (i.e., the shared memory hierarchy), and a model that combines the core and contention

models to form an overall multi-core processor performance model. This approach does not address the challenge of having to deal with the explosion in the number of multi-program workload mixes, because the contention model needs to be trained for each workload mix. Although the contention model can be trained through cache simulation, which is much faster than multi-core simulation, it is fundamentally limited by the explosion in the number of multi-program workload mixes. Our method on the other hand, addresses this fundamental challenge through analytical modeling. Further, our method takes into account time-varying execution behavior for determining the amount of contention through shared resources and its impact on performance.

Chandra et al. [2005] propose three cache contention models. The input to the models is the shared cache stack distance distribution or a circular sequence profile for each program. The output is the number of additional cache misses due to cache sharing for each of the threads. The three models proposed by Chandra et al. include the Frequency of Access (FOA) model, the Stack Distance Competition (SDC) model and the Inductive Probability (Prob) model. The key difference between our work and Chandra et al.'s work is that we predict overall multi-core processor performance, in contrast to Chandra et al.'s method which estimates cache miss rates only. Further, they do not take into account time-varying phase behavior, and their evaluation is limited to two-program workload mixes only.

Van Biesbrouck et al. [2004] propose the co-phase matrix as a method to quickly simulate multi-program workloads on multi-threaded architectures. The co-phase matrix takes into account a program's time-varying execution behavior and basically keeps track of the performance for all the co-phases in a two-program workload mix. The key idea here is that each co-phase needs to be simulated only once, its performance is stored in the co-phase matrix, and whenever the same co-phase is encountered again, the relative progress for each of the co-executing programs is simply picked from the co-phase matrix, and does not to be simulated again. This greatly reduces overall simulation time. In follow-on work, Van Biesbrouck et al. [2006] consider multiple starting points for each of the programs in the multi-program workload mix. The co-phase matrix does not address the challenge of having to deal with an explosion in the number of multi-program workloads, i.e., the co-phase matrix needs to computed for each multi-program workload mix of interest.

Van Biesbrouck et al. [2007] propose a method for picking a representative set of multi-program workloads. They profile each program using a

set of micro-architecture independent characteristics, and they then apply statistical analysis techniques such as Principal Component Analysis and Cluster Analysis to pick a limited but representative set of multi-program workload mixes. Similar to our work, they aim at addressing the explosion in the number of multi-program workload mixes, however, their solution differs from ours in a fundamental way. Their approach comes up with a limited set of multi-program workload mixes that need to be simulated in detail, whereas our approach estimates the performance of a multi-program workload mix through analytical modeling. This implies that we can estimate performance for a large number of multi-program workload mixes much more quickly, and in addition, in contrast to Van Biesbrouck et al., we compute confidence bounds on performance by considering a very large number of workload mixes.

Tuck and Tullsen [2003] propose a methodology for quantifying performance on multi-threaded architectures, which is also applicable to multi-core processors. A challenging problem when evaluating multi-threaded processor performance is that the relative progress of independent co-executing programs may differ across processor architectures, and hence, the effective multi-program workload may be different across architectures, leading to biased and incorrect design decisions. Tuck and Tullsen propose to re-iterate the execution of a program in a multi-program workload execution as soon as it reaches the end of its execution. This process is re-iterated until convergence. Whereas Tuck ad Tullsen apply this approach on real hardware experiments, Vera et al. [2007] propose a similar approach for simulation purposes. Both approaches run either real hardware or simulation experiments to evaluate multi-program performance; instead, MPPM employs an analytical model which allows for evaluating a large number of multi-program workload mixes in limited time.

Alameldeen and Wood [2003] study non-determinism when evaluating multi-threaded programs on multi-processor processors. Non-determinism refers to the fact that small timing variations can cause executions that start from the same initial state to follow different execution paths. They propose adding non-determinism in deterministic simulators to model this effect, and they report confidence bounds when simulating multi-threaded programs. Our work is orthogonal and targets multi-program workloads. MPPM quantifies how variability in multi-program workload mixes affects performance; this is done by computing confidence bounds.

2.8 Summary

Current practice in multi-core simulation is to consider a limited number of multi-program workloads. We have shown that tens of multi-program workloads are not representative, and may lead to incorrect decisions in practical design and research questions. Instead, we advocated and proposed the Multi-Program Performance Model (MPPM), which is an analytical approach for estimating multi-program multi-core performance from single-core simulation runs. MPPM incorporates a program's time-varying execution behavior, and accurately estimates the tight performance entanglement between co-executing programs because of resource contention in shared caches. MPPM was shown to be accurate within 2.3% and 2.9% on average for STP and ANTT, respectively, for SPEC CPU2006 and up to 16 cores, while being up to five orders of magnitude faster than detailed simulation. Hence, MPPM estimates multi-program performance for a very large number of multi-program workloads in a reasonable amount of time, while providing confidence bounds on its performance estimates. An appealing use for MPPM is to identify multi-program workloads that yield poor performance due to excessive conflict behavior in shared resources.

Chapter 3

Understanding Fundamental Design Choices in Single-ISA Heterogeneous Multi-Core Architectures

Life is like riding a bicycle. To keep your balance you must keep moving.
Albert Einstein

In this chapter, we aim at understanding how heterogeneity affects both chip throughput and per-program performance; how heterogeneous architectures compare to homogeneous architectures under said performance metrics; and how fundamental design choices, such as core type, cache size and off-chip bandwidth, affect performance. This is done using the MPPM model to quickly explore the large architecture and workload design space.

3.1 Introduction

Heterogeneous multi-core processor architectures have received substantial interest in both academia and industry over the past years. One of the primary drivers for heterogeneous architectures is higher performance for a given power budget, or higher power-efficiency for a given performance target, by dynamically scheduling jobs on the highest performance or most power-efficient core on the chip. By doing so, total chip throughput is maximized while not exceeding the total power budget, or vice versa, power and energy consumption is reduced while maintaining specific per-

formance targets. Prior work has reported substantial power and energy savings through heterogeneity [Kumar et al. 2004]. Industry is actively pursuing the road of heterogeneity: example heterogeneous architectures are the IBM Cell processor with its 8 special-purpose engines and one general-purpose RISC core [Kahle et al. 2005a], as well as recently introduced CPU chips with an integrated GPU such as Intel's Sandy Bridge [Intel 2008], AMD's Fusion [AMD 2008], and NVidia's Tegra [NVidia 2010]. Other commercial offerings integrate different general-purpose CPU core types, see for example NVidia's Kal-El [NVidia 2011] which integrates four performance-tuned cores along with one energy-tuned core, and ARM's big.LITTLE chip [Greenhalgh 2011], which integrates a high-performance big core with a low-energy small core on a single chip. The latter two examples are so-called single-ISA heterogeneous multi-cores, which means that the different core types implement the same instruction-set architecture (ISA).

We aim at exploring the heterogeneous single-ISA multi-core architecture design space, and address some of the fundamental questions related to heterogeneity, such as: What are the performance benefits from heterogeneity over homogeneous architectures, i.e., how does heterogeneity affect chip throughput versus job turnaround time? If heterogeneity yields any performance benefits, what level of heterogeneity should be supported, i.e., how many different core types should be integrated? Do two different core types provide most of the benefit or do we need more core types? And what should these core types look like? Should we go for extreme core types, i.e., aggressive 4-wide out-of-order cores versus scalar in-order processor cores? Or should we deploy middle-of-the road cores along with extreme core types? How does heterogeneity affect off-chip bandwidth requirements? Or, vice versa, how do bandwidth limitations affect some of the fundamental trade-offs in heterogeneous architectures? In spite of the substantial amount of prior work done in this area, a comprehensive study exploring these heterogeneous processor trade-offs and design choices has not been published before, to the best of our knowledge.

We use the MPPM model for exploring the heterogeneous multi-core design space. MPPM enables exploring the heterogeneous multi-core design space from single-core runs (see Chapter 2), i.e., the model has linear-time complexity in the number of core types while enabling performance predictions for arbitrary compositions of heterogeneous architectures and workloads. Moreover, it allows for quantifying heterogeneous architecture performance for a large number (hundreds) of possible job mixes in

a reasonable amount of time. Performing the same study using architectural simulation would have been completely infeasible because of its time complexity: simulating and exploring a large heterogeneous architecture design space for a very large number of job mixes is impossible in a reasonable amount of time.

Our methodology uses analytical modeling to estimate performance for an arbitrary heterogeneous processor architecture and arbitrary job mixes. In contrast to prior work which focused on total chip throughput (also called weighted speedup), we quantify performance along two dimensions: we measure both overall system throughput and per-program performance (average job turnaround time). Although throughput and per-program performance are not independent, we find it very insightful to analyze multicore processor in terms of these performance axes. In particular, we determine the frontier of Pareto-optimal processor architectures that provide the optimum trade-off between system throughput versus average job turnaround time. Pareto-optimality implies that there exist no design points that outperform the Pareto-optimal frontier on all objectives (both system throughput *and* job turnaround time) at the same time. In other words, one cannot say whether one Pareto-optimal configuration outperforms another Pareto-optimal configuration — instead, Pareto-optimal configurations represent different trade-offs. We apply our methodology to SPEC CPU2006 workload mixes and we consider heterogeneous multi-core processor configurations with up to five core types ranging from simple single-issue in-order cores to aggressive four-wide out-of-order cores.

This analysis leads to several interesting and insightful observations.

- While it is true that, as reported by prior work, replacing an aggressive out-of-order core in a homogeneous architecture with several simple in-order cores improves system throughput (while assuming a fixed chip area), it also decreases average per-program performance. Conversely, trading a number of simple in-order cores for an aggressive out-of-order core improves per-program performance, but it also decreases total system throughput. So, fundamentally, heterogeneity trades job turnaround time for total system throughput.

- Homogeneous architectures cover a broad range of the performance spectrum in terms of throughput versus job turnaround time. For a fixed chip area budget, a limited number of aggressive out-of-order cores yield short job turnaround time with limited system through-

put; a large number of simple in-order cores on the other hand yield high system throughput at the cost of longer job turnaround times. Mediocre cores yield intermediate design trade-offs. Heterogeneity on the other hand allows for designing multi-core processors with more fine-grained trade-offs in system throughput versus job turnaround time. Interestingly though, although there exist heterogeneous design points that outperform homogeneous designs both in terms of throughput and per-program performance, some homogeneous design points appear on the heterogeneous architecture Pareto frontier. In other words, some homogeneous configurations are optimal for particular throughput versus job turnaround time trade-offs.

- We find that two core types offer most of the performance benefits from heterogeneity, i.e., going to a larger number of core types does not contribute much. However, performance is greatly affected by which core types are chosen and different compositions lead to different performance trade-offs. Further, some compositions of core types do not yield Pareto-optimal configurations. For other compositions, the number of cores of each core type determines whether the heterogeneous multi-core processor is globally optimal.

- Limited off-chip bandwidth has a significant impact on the fundamental design choices in heterogeneous architectures. When limiting off-chip bandwidth, increasing system throughput comes at the cost of a proportionally larger degradation in per-program performance. Further, although a homogeneous design with many small cores yields the highest throughput assuming infinite bandwidth, only heterogeneous designs can achieve the highest throughput under limited off-chip bandwidth. Finally, architectures designed for high-throughput should employ large LLCs in order to reduce off-chip bandwidth pressure.

- We also find that the effectiveness of heterogeneous architectures heavily depends on how jobs are mapped on the different core types. Simple heuristics based on CPI or miss rates to discern compute- versus memory-intensive jobs do not achieve optimum performance. Instead, more accurate estimates that compare relative performance across core types are needed for effective job-to-core mapping.

The key contributions are to comprehensively explore the heterogeneous multi-core design space and provide insight in some of the fundamental trade-offs and design choices. We use analytical modeling to

do so which enables exploring many more machine configurations and workloads than what is possible to consider with cycle-accurate simulation. Further, one of the key trade-offs that we study relates to chip throughput versus per-program performance. Prior work on the other hand focused on throughput only, for the most part; and prior work that did consider both chip throughput and per-program performance assumed different workload conditions. Kahle et al. [2005b] and Annavaram et al. [2005] considered multi-threaded workloads and advocated spending more energy per instruction during serial phases (e.g., run serial phases on big cores or at higher frequency/voltage in a heterogeneous multi-core); Kumar et al. [2004] focus on throughput when assuming a fixed number of independent programs in a multi-program workload, and focus on per-program performance when considering variable active thread counts in multi-program workloads. In this work, we consider abundant numbers of active thread counts, i.e., at least as many independent programs as there are cores, and we find that heterogeneous multicores provide a trade-off between chip throughput and per-program performance, even under such workload conditions.

The remainder of this chapter is organized as follows. We first describe the analytical model that we use in our exploration (Section 3.2). Subsequently, we present our methodology for exploring the heterogeneous processor architecture design space (Section 3.3). After detailing our experimental setup (Section 3.4), we then present our results and findings (Section 3.5). Finally, we describe related work (Section 3.6) and conclude (Section 3.7).

3.2 Multi-core Performance Modeling

The analytical model used in this chapter is called the Multi-Program Performance Model (MPPM) which we discussed in great detail in the previous chapter. Recall that MPPM collects a profile during single-core simulation that captures a program's memory behavior as well as its phase behavior, and then employs an iterative method to model the performance entanglement between co-executing programs on a multi-core processor with shared caches: the iterative method captures how per-program performance affects the amount of resource sharing, and, vice versa, how resource sharing in its turn affects per-program performance. We refer to Chapter 2 for a detailed description and evaluation of MPPM.

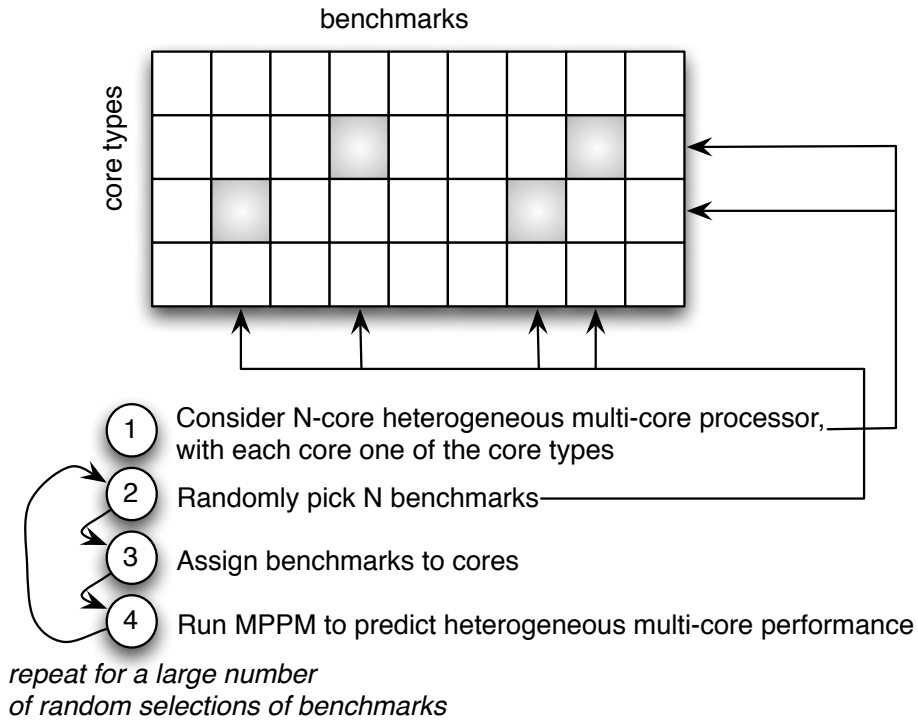


Figure 3.1: Using MPPM for exploring the heterogeneous multi-core design space.

3.3 Exploring the Heterogeneous Multi-core Design Space

Because MPPM provides us with a way for estimating multi-core performance based on single-core simulation results, it is very fast and therefore the ideal tool to explore the very large multi-core design space. Heterogeneous multi-cores are gaining a lot of momentum in the computer architecture community in the last few years because of their potential energy efficiency. However, the design space for heterogeneous multi-cores is huge. In fact, it is so large that without tools like the MPPM framework, it is impossible to do enough experiments to have meaningful data. The problem is even more acute than for homogeneous multi-cores (Chapter 2) because every core can theoretically be individually tuned (there are many more parameters than can be changed). We leverage the MPPM framework to efficiently explore the heterogeneous multi-core design space. This is done as follows, see also Figure 3.1.

We first perform single-core simulation runs for all of the benchmarks

and all of the core types of interest. In this chapter, we consider five core types, and the SPEC CPU2006 benchmarks as our workloads. These single-core simulations need to be done only once, and enable us to explore the heterogeneous design space for arbitrary combinations of number of cores and core types, and for arbitrary job mixes. This matrix of single-core simulation results serves as input for our design space exploration.

For estimating multi-core performance, we consider the single-core simulation results for the core types of interest; the core types could be diverse in case of a heterogeneous design or the same in case of a homogeneous design (step #1 in Figure 3.1). We then randomly pick N benchmarks, with N the number of cores (step #2), and we assign benchmarks to cores (step #3). As we will observe later in the chapter, benchmark-to-core mapping has an important impact on overall performance. In this chapter, unless mentioned otherwise, we map the job that benefits the most to the most aggressive core, and so forth until all jobs are mapped to cores. (We provide more details on the job-to-core mapping approach later in the chapter, and we find this heuristic to be close to optimal, see Section 3.5.6). We then use MPPM to estimate multi-core performance (step #4). This whole process (steps #2 through #4) is iterated for a large number of randomly chosen benchmark mixes (500 in total per experiment). The key feature of MPPM is that it is based on an analytical model that can be evaluated very quickly. One experiment takes approximately one day to complete using MPPM; this includes the one-time cost of single-core simulations for all workloads and core types, as well as computing the MPPM model. The traditional methodology using detailed architectural simulation would take more than 80 days for performing the same experiment. Put differently, MPPM enables considering a large set of job mixes in limited time, which increases confidence in the results compared to detailed simulation, which would limit the number of possible job mixes to just a few examples because of simulation time constraints.

3.3.1 Heterogeneous multi-core design space

We consider five core types in our design space exploration: 4-wide and 2-wide out-of-order cores, and 4-wide, 2-wide and scalar in-order cores. For the out-of-order cores, we assume a 128-entry and 32-entry reorder buffer for the 4-wide and 2-wide core, respectively. We assume a core to have private L1 instruction and data caches, as well as a private L2 cache. The L1 caches are 32 KB in size; the L2 caches are 256 KB in size and are 8-

	#BCEs
scalar in-order core	1
2-wide in-order core	2
4-wide in-order core	3
2-wide out-of-order core	4
4-wide out-of-order core	8
512 KB LLC slice	1

Table 3.1: Chip area cost model.

way set-associative. The L3 cache is shared among the cores and is the last-level cache (LLC) in our setup; we vary the LLC size between 1 MB, 2 MB and 4 MB in our experiments, and we assume the LLC to be 16-way set-associative. All caches implement an LRU replacement policy.

The area cost models for each core type are derived from chip die photos from Intel Quad-Core Nehalem and Intel Atom processors, see also Table 3.1. Nehalem implements 4-wide out-of-order cores, whereas Intel Atom implements 2-wide in-order cores. We empirically observed a 1 to 4 ratio in chip area between these core architectures (in the same chip technology). We also observed that one slice of 512 KB LLC corresponds roughly to half an Intel Atom core or one eighth of an Intel Nehalem core. Hence, we assign one Base Core Equivalent (BCE) [Hill and Marty 2008] to a 512 KB LLC slice; 2 BCEs to a 2-wide in-order core (alike Intel Atom) and 8 BCEs to a 4-wide out-of-order core (alike Intel Nehalem). We extrapolated towards scalar in-order, 4-wide in-order and 2-wide out-of-order cores as shown in Table 3.1.

We consider 40 BCEs in total in our experiments. This corresponds to the chip area of the Intel Quad-Core Nehalem processor, which includes four 4-wide out-of-order cores along with a 4 MB LLC. In the experiments to follow we vary the configuration of the multi-core processor architecture and we consider both homogeneous and heterogeneous designs, while bounding total chip area to 40 BCEs.

Unless mentioned otherwise, we assume unlimited off-chip bandwidth; however, in the results section, we do study how limited off-chip bandwidth affects heterogeneous multi-core performance.

3.3.2 Multi-core performance

An important distinction between this work and prior work in heterogeneous multi-core architectures is that we focus on two metrics for quan-

tifying multi-core performance from two complementary perspectives; prior work primarily focused on a single metric namely weighted speedup which quantifies performance from a system perspective only and does not take into account per-program performance. By using two metrics we quantify multi-core performance when running multi-program workloads from both a system's and a user's perspective. We consider system throughput (STP) as a metric to quantify system performance, along with average normalized turnaround time (ANTT) to quantify user-perceived per-program performance. The original definition by Eyerman and Eeckhout [2008] introduced STP and ANTT assuming homogeneous multi-core architectures. However, these definitions are inappropriate for heterogeneous designs. The next paragraph describes the original definitions of STP and ANTT, followed by a discussion on how we extended these metrics for heterogeneous designs.

STP and ANTT for homogeneous multi-cores

STP measures multi-core performance from a system's perspective and quantifies the accumulated progress by all the programs in the multi-program workload mix. ANTT focuses on user-perceived performance and quantifies the average slowdown during multi-core execution relative to single-core, isolated execution. ANTT is the reciprocal of the hmean metric proposed by Luo et al. [2001] For more details on how STP and ANTT are calculated, we refer to Chapter 2, Section 2.3.

STP and ANTT for heterogeneous multi-cores

STP and ANTT as defined above for homogeneous multi-cores have no meaning when used for heterogeneous multi-cores. The reason is that single-core CPI ($CPI_{SC,p}$) is not well defined in a heterogeneous multi-core because there are different core types, and hence, the question is which one to measure single-core CPI for. Picking different core types to measure single-core CPI for the different jobs would preclude comparing heterogeneous designs against each other. Hence, we need to agree on a single core type on which to measure single-core CPI ($CPI_{SC,p}$). In this work we arbitrarily consider the 4-wide out-of-order core as our baseline core to measure $CPI_{SC,p}$; the baseline core is assumed to have the entire cache hierarchy (including the shared LLC) to its disposal. Both STP and ANTT are then computed relative to the single-core CPI on this baseline core. In other words, STP now quantifies system throughput achieved

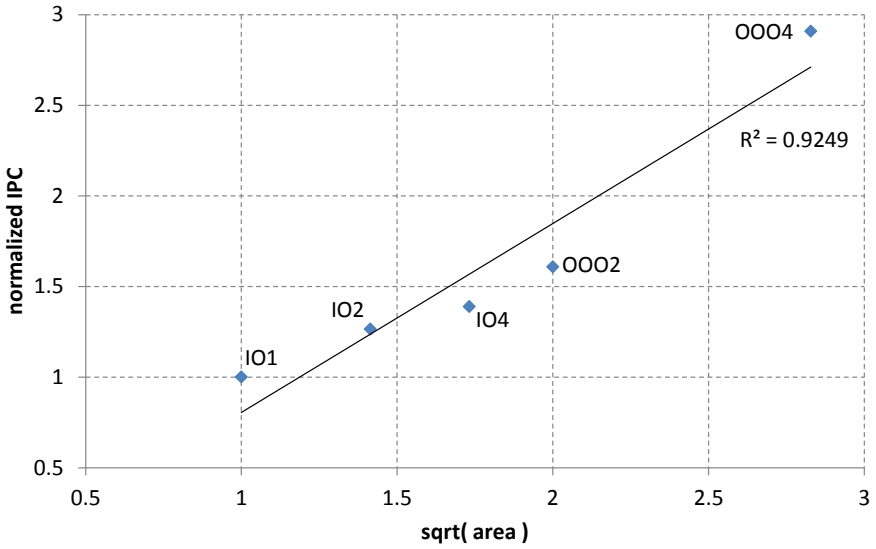


Figure 3.2: Average normalized IPC for the five core configurations considered in this study as a function of the square root of the area counted in BCEs.

over a single 4-wide out-of-order core, e.g., an STP of 8 means that this design achieves an 8× higher total system throughput compared to a single baseline 4-wide out-of-order core. Likewise, ANTT quantifies the average normalized turnaround time relative to a single 4-wide out-of-order core, e.g., an ANTT of 4 means that this design yields an average per-program slowdown of a factor 4× relative to a single 4-wide out-of-order core.

3.4 Experimental Setup

For obtaining the single-core simulation results that serve as input to the performance model, we use a multi-core processor simulator based on CMP$im [Jaleel et al. 2008a], which is an x86 simulator built on top of Pin [Luk et al. 2005]. CMP$im is a user-level simulator and allows for simulating both single-core and multi-core processor architectures. This is the same setup that we described in more detail in Chapter 2

We consider all the SPEC CPU2006 benchmarks with their reference inputs. All the benchmarks were compiled with the GNU C compiler version 4.3.4 and optimization level `-O2`. We use SimPoint [Sherwood et al. 2002] to pick representative simulation points of one billion instructions each.

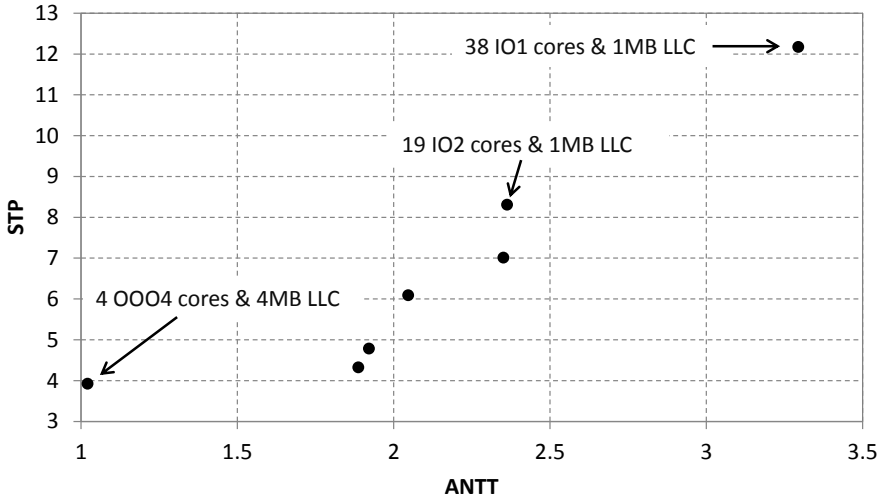


Figure 3.3: Pareto-optimal homogeneous multi-core configurations as a function of STP (vertical axis) and ANTT (horizontal axis).

Figure 3.2 shows performance in terms of average normalized IPC across all benchmarks for the five core types considered in this study as a function of the square root of the area (counted in the number of BCEs, see also Table 3.1). This data complies with Pollack's Law [Borkar 2007] which states that core performance is proportional to the square root of the chip area.

3.5 Results

We now explore the heterogeneous multi-core design space using the methodology just described. This is done in a number of steps. We start by exploring the homogeneous versus heterogeneous multi-core design spaces, and we explore how the number of core types affects heterogeneous multi-core performance. Subsequently, we study the importance of job-to-core mapping, and how heterogeneous multi-core performance is affected by off-chip bandwidth. Finally, we evaluate how sensitive heterogeneous multi-core performance is with respect to LLC size, and which core types should be employed in a heterogeneous design.

3.5.1 Homogeneous multi-core processors

Before exploring the heterogeneous multi-core processor design space, we start with exploring the homogeneous multi-core design space. For this experiment, we consider all possible homogeneous multi-core design points with all possible LLC cache sizes that fit in 40 BCEs; further, we assume unlimited off-chip bandwidth for now. Out of this set of possible homogeneous multi-core designs, we determine the Pareto-optimal ones in terms of system throughput versus average job turnaround time. Figure 3.3 shows the Pareto-optimal homogeneous multi-core design points as a function of STP (vertical axis) and ANTT (horizontal axis). The design points vary from a multi-core design with four 4-wide out-of-order cores and a 4 MB LLC — alike Intel Quad-Core Nehalem — with an STP of 3.92 and an ANTT of 1.02, to 38 scalar in-order cores and a 1MB LLC which achieves an STP of 12.17 and an ANTT of 3.29. In other words, the aggressive out-of-order core design yields excellent per-program performance, yet, total chip throughput is limited. Conversely, the simple in-order core design yields more than $3\times$ higher system throughput, yet, per-program performance is also more than $3\times$ lower.

Finding #0: *Changing core types in a homogeneous multi-core architecture yields different trade-offs in system throughput and per-program performance.* In essence, simple in-order cores trade per-program performance for throughput, and conversely, aggressive out-of-order cores trade throughput for per-program performance. Mediocre cores lead to multi-core design points in the spectrum between aggressive out-of-order cores and simple in-order cores. These trade-offs are well known, but it is important to restate them here in light of the exploration for heterogeneous architectures.

3.5.2 Pareto-optimal heterogeneous multi-cores

Now that we have a good understanding of the homogeneous multi-core design space, we move to heterogeneous architectures. We consider all possible heterogeneous multi-core configurations with at most two different core types. Again, we assume all possible LLC cache sizes, a total chip area of 40 BCEs, and off-chip bandwidth being unlimited. (We consider the impact of limited off-chip bandwidth on heterogeneous multicore design considerations in the next section; the reason for considering unlimited off-chip bandwidth here is to solely focus on the impact of core types initially and study the fundamental impact of core types on heterogeneous multi-core performance.) We determine the Pareto frontier as a function of STP

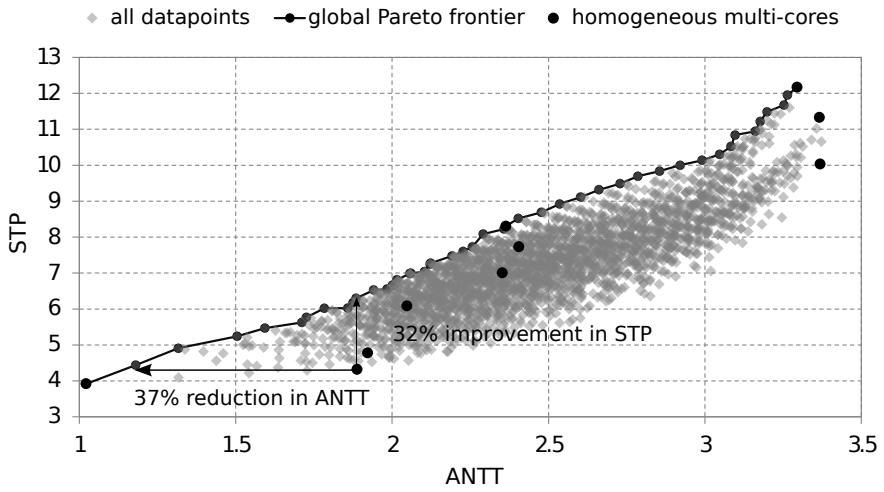


Figure 3.4: Pareto frontier of multi-core configurations, along with all processor configurations explored including the homogeneous design points.

and ANTT out of this set of heterogeneous multi-core configurations. Figure 3.4 shows all the design points, along with the Pareto frontier and the homogeneous multi-core designs (as points of reference).

Finding #1: *Heterogeneity poses a trade-off between system throughput and per-program performance.* Prior work motivated heterogeneity as a way for increasing system throughput within a given power budget [Kumar et al. 2004]. While our results confirm this finding, Figure 3.4 shows that heterogeneity also increases average job turnaround time, or in other words, average per-program performance degrades — compare the heterogeneous multi-core configurations on the Pareto frontier against the homogeneous design point at the bottom left on the Pareto frontier (consisting of 4 4-wide out-of-order cores). Similarly, heterogeneity improves per-program performance compared to a homogeneous design that achieves the highest throughput at the top right on the Pareto frontier (consisting of 38 single-issue in-order cores). Hence, fundamentally, heterogeneity trades per-program performance for system throughput, and vice versa.

Finding #2: *Heterogeneity enables making more fine-grained performance trade-offs.* Although changing the core types in a homogeneous multi-core design enables trading off system throughput against per-program performance, as discussed in the previous section, heterogeneity enables more fine-grained performance trade-offs to be made. Some trade-off points in system throughput versus per-program performance can only be achieved

through heterogeneity.

Finding #3: *Some heterogeneous multi-core configurations outperform specific homogeneous architectures in both system throughput and per-program performance.* Heterogeneity yields a number of configurations with significantly better performance compared to homogeneous designs. For example, see also Figure 3.4, through heterogeneity, one can achieve a 37% reduction in ANTT while achieving similar STP, or conversely, one can achieve a 32% STP increase at the same ANTT. The reason is that heterogeneity allows for mapping jobs to cores that are most appropriate for the job at hand.

Finding #4: *Some homogeneous design points yield optimal performance trade-offs.* Another interesting observation is that some homogeneous designs also appear on the Pareto frontier for the heterogeneous designs: the three homogeneous configurations labeled in Figure 3.3 with scalar in-order cores, dual-issue in-order cores and aggressive 4-wide out-of-order cores, respectively, also appear on the Pareto frontier in Figure 3.4. In other words, heterogeneity does not provide a range of architectures that outperform homogeneous architectures over the entire STP vs. ANTT range. Instead, heterogeneity provides a broader range of STP vs. ANTT trade-offs, and there are many more design points on the Pareto frontier that can be obtained through heterogeneity than what can be achieved through homogeneous designs. However, particular trade-offs are best achieved through a homogeneous design. This, we believe, is an interesting and novel insight: heterogeneity, fundamentally, trades per-program performance for system throughput, and while it is true that heterogeneous architectures can outperform homogeneous architectures for some throughput versus per-program trade-off points, heterogeneous designs do not always outperform homogeneous designs, and some throughput versus per-program performance trade-offs are best achieved through homogeneous designs.

Finding #5: *Two core types provide most of the benefits from heterogeneity.* In the previous experiment, we considered at most two core types. An interesting question is whether adding additional core types improves heterogeneous multi-core architecture performance above two core types. Figure 3.5 shows the Pareto frontier for at most two, three, four and five core types. Interestingly, adding more than two core types does not improve performance much. The highest improvement observed in throughput and turnaround time is no larger than 6.6% and 7.9%, respectively, going from two to three core types; beyond three core types, the improvement is less than 0.3%. Hence, we conclude that two core types provide most of the

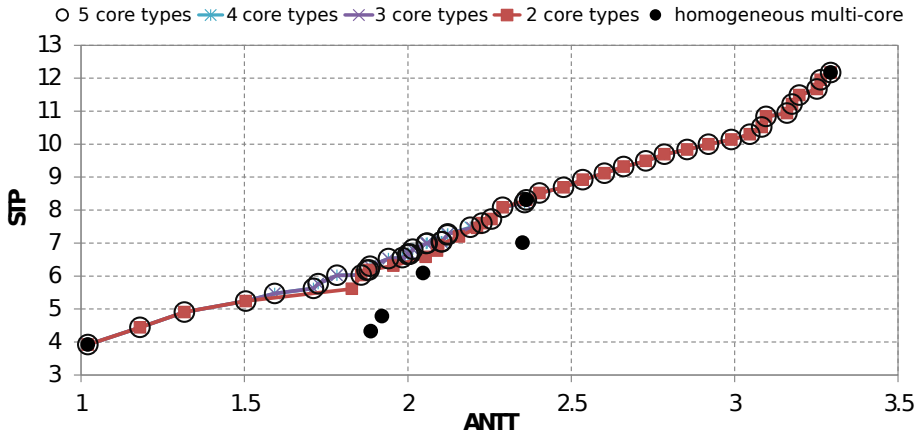


Figure 3.5: Pareto frontier for heterogeneous multi-core architectures with a varying number of core types.

benefits through heterogeneity, and three or more core types does not contribute much.

3.5.3 Limiting off-chip bandwidth

So far, we assumed that off-chip bandwidth is unlimited. We now study the impact of limited off-chip bandwidth on heterogeneous multi-core processor design. We consider a simple bandwidth model to this end. We compute the average off-chip bandwidth requirements for each program in the job mix by multiplying the number of LLC misses per instruction with the achieved per-program IPC, clock frequency, and the machine's LLC cache line size (64 bytes). The sum of the per-program off-chip bandwidth requirements then yields the total off-chip bandwidth requirements. If the aggregate off-chip bandwidth demands exceed the maximum off-chip bandwidth, we discard the design point and we consider it to be invalid.

Figure 3.6 shows the Pareto frontier for unlimited off-chip bandwidth as well as for limited bandwidth at 30 GB/s, 20 GB/s and 15 GB/s. As expected, limited off-chip bandwidth puts a limit on the maximum achievable system throughput, e.g., compare the unlimited bandwidth curve versus the 30 GB/s curve: the maximum achievable STP goes down from 12.17 to 10.03; ANTT varies across a similar range. When limiting off-chip bandwidth even further to 20 GB/s and 15 GB/s, we observe a decrease in achievable STP and ANTT. Further, limiting off-chip bandwidth puts a

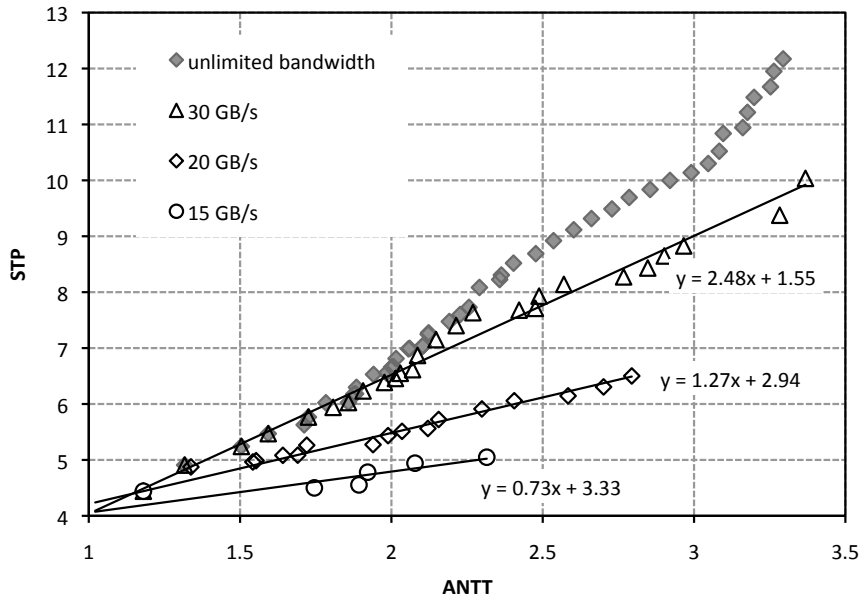


Figure 3.6: Evaluating how off-chip bandwidth limitations affect heterogeneous multi-core performance.

limit on how per-program performance can be traded for throughput, i.e., the range of possible design points is reduced.

Finding #6: *When limiting off-chip bandwidth, increasing system throughput comes at the cost of a proportionally larger degradation in per-program performance.* Interestingly, we observe an almost linear relationship between STP and ANTT for the heterogeneous design points on the Pareto frontier under limited bandwidth constraints; the linear fits are shown in Figure 3.6. Note that the slope decreases with decreasing off-chip bandwidth. This implies that if a processor designer aims at increasing system throughput, limitations in off-chip bandwidth will force the designer to tolerate increasingly larger job turnaround times. In other words, if the goal is to improve system throughput by a given percentage, per-program performance will degrade by an increasingly larger percentage at lower off-chip bandwidths.

Finding #7: *Highest throughput can only be achieved through heterogeneity under off-chip bandwidth constraints.* It is interesting to study how homogeneous multi-cores fare under limited off-chip bandwidth constraints. Figure 3.7 shows the Pareto frontier for heterogeneous and homogeneous designs at 20 GB/s of off-chip bandwidth. The key observation here is that the highest throughput cannot be achieved through homogeneity under this particular off-chip bandwidth constraint; only heterogeneity can

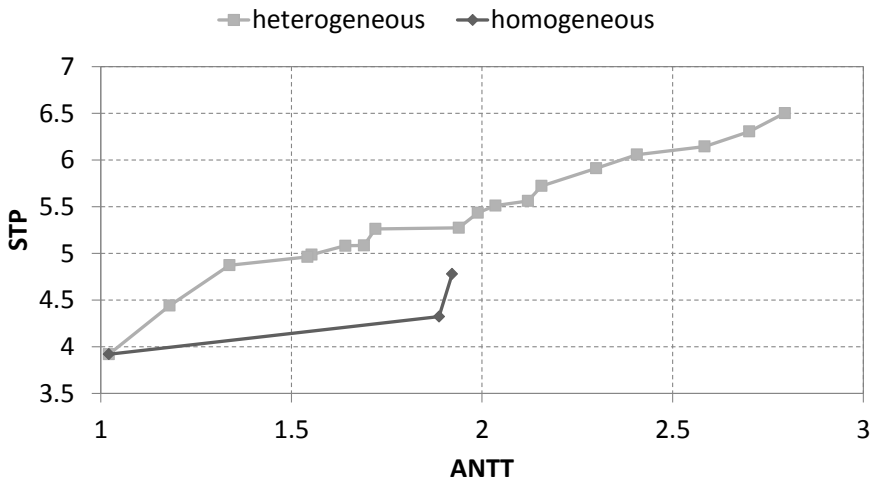


Figure 3.7: Pareto frontier for heterogeneous and homogeneous multi-core designs under 20 GB/s off-chip bandwidth constraints.

achieve these high levels of throughput. The reason is that a large number of small cores imposes large LLC to satisfy bandwidth constraints, which leads to suboptimal performance compared to a heterogeneous design with a few slightly more aggressive cores and a smaller LLC.

3.5.4 Impact of LLC size

As observed in the previous section, off-chip bandwidth has significant impact on (heterogeneous) multi-core performance. Caches are effective at reducing off-chip bandwidth pressure: cache hits do not need to go off chip, thereby saving off-chip traffic. Figure 3.8(a) shows the heterogeneous multi-core Pareto frontier for different cache sizes while assuming infinite off-chip bandwidth. Unsurprisingly perhaps, the smallest LLC (1 MB) configuration yields the highest throughput. In other words, unlimited off-chip bandwidth leads to integrating more cores and not larger caches for optimum performance.

Finding #8: *Large LLCs yield highest throughput under off-chip bandwidth constraints.* Figure 3.8(b) shows the heterogeneous multi-core Pareto frontier with off-chip bandwidth limited to 20 GB/s. We observe a very different result under limited off-chip bandwidth. Counter-intuitively and surprisingly, the highest system throughput is achieved for the largest LLC (see righthand side in Figure 3.8(b)). The reason is that a large LLC reduces the off-chip bandwidth pressure imposed by employing many small

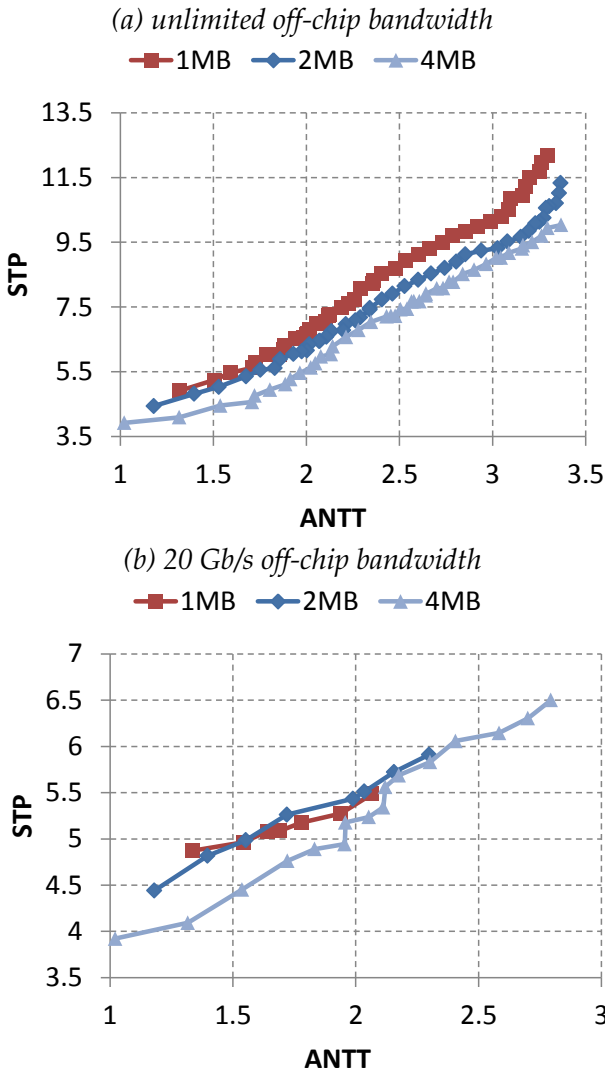


Figure 3.8: Evaluating how LLC size affects Pareto-optimal heterogeneous multi-core performance under different off-chip bandwidth constraints.

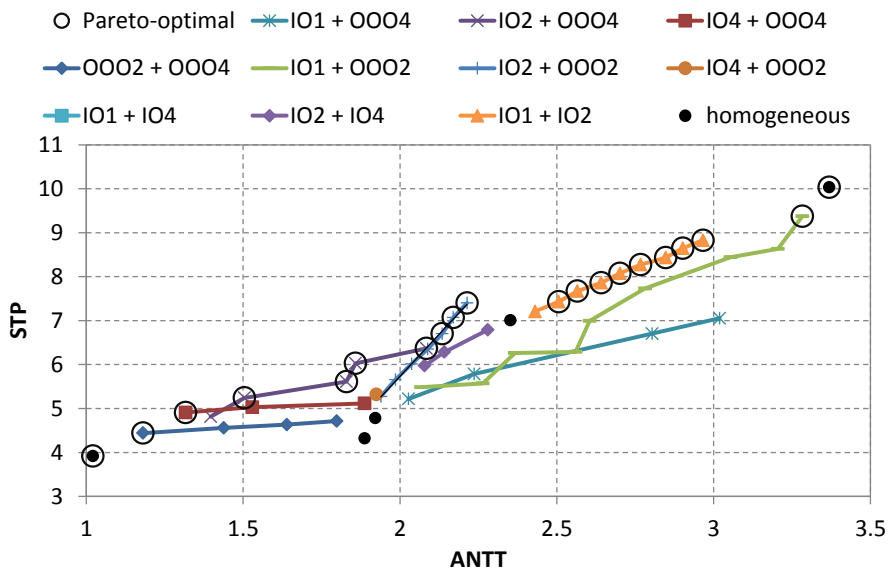


Figure 3.9: Pareto frontiers for heterogeneous multi-cores with two core types, assuming 30 GB/s off-chip bandwidth.

cores to achieve high throughput. In other words, the high-throughput designs on the righthand side of Figure 3.8(b) are bandwidth-constrained, hence, they benefit from a larger LLC to reduce bandwidth pressure. Balanced system throughput and per-program performance (middle part in Figure 3.8(b)) is achieved by employing more (or at least a couple) aggressive big cores which impose less off-chip bandwidth traffic, and hence, a smaller LLC is optimal. However, for the designs on the lefthand side of Figure 3.8(a) and (b) which are optimized for per-program performance, a large LLC is optimal again. The reason is that a large LLC (along with aggressive out-of-order cores) yields the highest per-program performance. In other words, these designs are not bandwidth-constrained but optimized for per-program performance, and hence also benefit from a large LLC.

3.5.5 Which core types to employ in a heterogeneous design?

As mentioned throughout this chapter, there is a clear performance benefit to be achieved from heterogeneity for particular performance trade-offs. An open question though is what the core types should be for optimum performance. Figure 3.9 shows the Pareto frontier for all possible combinations of two core types; we assume off-chip bandwidth is limited to

30 GB/s. The Pareto-optimal points across all Pareto frontiers per two core types obviously corresponds to the global Pareto frontier shown in Figure 3.6.

Finding #9: *Particular compositions for heterogeneity yield particular performance trade-offs, and some compositions do not yield Pareto-optimal performance.* The interesting observation from Figure 3.9 is that the composition of a Pareto-optimal heterogeneous multi-core varies along the Pareto frontier. At high throughput, a Pareto-optimal heterogeneous multi-core is to be composed of single-issue and dual-issue in-order cores. This is in line with prior work which advocated simple cores for server-type throughput applications in the datacenter [Kongetira et al. 2005, Kgil et al. 2006, Lim et al. 2008, Reddi et al. 2010, Mudge and Hölzle 2010]: highest throughput is achieved using many small cores. For high per-program performance, Pareto-optimal heterogeneous multi-cores should employ at least one out-of-order core type. Interestingly, a heterogeneous multi-core composition with two particular core types that is Pareto-optimal locally is not necessarily Pareto-optimal globally, i.e., the exact number of cores of a given type is important and determines whether the heterogeneous multi-core is globally optimal. If not, there exist compositions with other core types that yield better throughput and per-program performance.

Note also that some heterogeneous multi-core compositions do not yield Pareto-optimal performance, see for example heterogeneous designs with four-issue in-order and two-issue out-of-order cores, as well as heterogeneous designs with two-issue and four-issue in-order cores. This can be understood intuitively given the relatively small performance and chip area differences between these core types, see also Section 3.4. However, heterogeneous architectures with single-issue in-order and four-issue out-of-order cores also fall in this category. This is a surprising result because these two core types are the most extreme core types in the mix. The reason is that single-issue in-order cores call for a larger LLC to meet the off-chip bandwhich constraints. Two-issue in-order cores on the other hand put less aggregate pressure on off-chip bandwidth (because one two-issue core generates less off-chip traffic than two single-issue cores for the same chip area). As a result, single-issue in-order cores demand for a larger LLC which is suboptimal compared to having fewer mediocre (dual-issue in-order) cores and a slightly smaller LLC.

3.5.6 Job-to-core mapping

An important challenge with heterogeneous multi-core architectures is how to schedule or map the jobs across the different core types in order to maximize performance. We consider four job-to-core mapping strategies.

- Optimal mapping maps jobs to cores so that overall performance are optimized. One could either optimize throughput or optimize turnaround time; here we maximize throughput (STP). The optimal mapping is obtained by exhaustively trying out all possible job-to-core mapping and picking the best one. This is an oracle and cannot be achieved in practice.

- Cache miss rate based mapping maps the job with the highest LLC miss rate to the lowest-end core, the job with the second highest LLC miss rate to the second lowest-end core, etc. In other words, we map compute-intensive jobs to the high-end cores and the memory-intensive jobs to the low-end cores. The intuition is to map jobs to the core type where they would presumably benefit the most. Several previously proposed scheduling algorithms for heterogeneous architectures are based on this heuristic, see for example [Kumar et al. 2004, Koufaty et al. 2010]. We explored IPC-based mapping strategies as well, following several other prior proposals [Becchi and Crowley 2008], but obtained similar results as for cache miss rate based mapping, hence, the IPC-based mapping results are omitted.

- A relative slowdown mapping assumes that it knows the relative performance for each job on each of the core types. This mapper iteratively picks the job in the job mix that would experience the largest relative slowdown from not being scheduled on the highest-end core, and maps that job to the highest-end core; the job is removed from the job mix and the core is no longer schedulable, after which the mapper picks the next job.

- Random mapping, as it says, performs a random mapping of jobs to cores.

Figure 3.10 compares these job-to-core mapping strategies for six heterogeneous multi-core processors with two core types, namely 4-wide out-of-order and 2-wide in-order cores. Again, we consider 500 randomly chosen multi-program workload mixes.

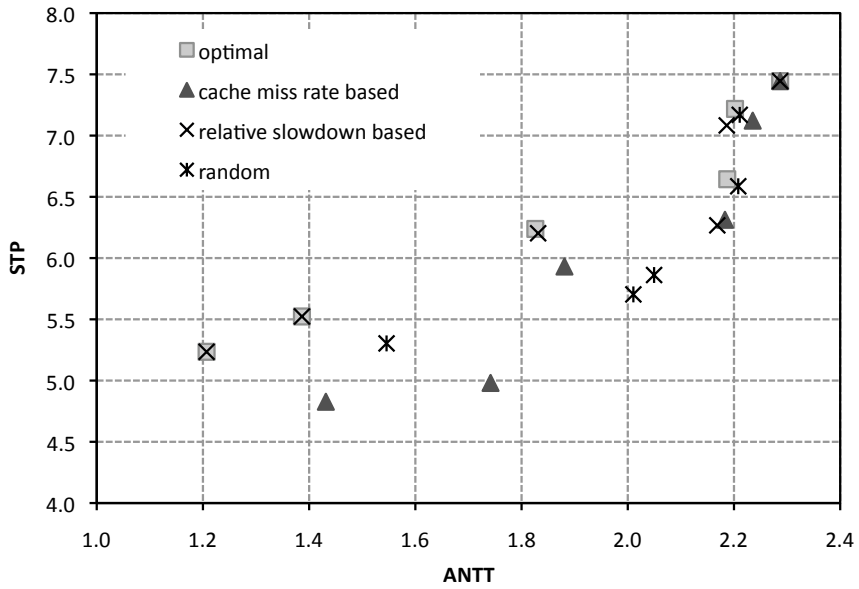


Figure 3.10: Evaluating how job-to-core mapping affects heterogeneous multi-core performance.

Finding #10: *Job-to-core mapping is both important and challenging for achieving optimum performance on heterogeneous multi-core architectures.* Clearly, random mapping as well as a simple heuristic such as cache miss rate based mapping are far from optimal. Random mapping is oblivious to the fact that the underlying hardware is heterogeneous and, as a result, it is not surprising that random mapping does not yield optimum performance. The cache miss rate based mapping strategy apparently does not have enough information about how to optimally map jobs to cores. The reason is that cache miss rate based mapping in unaware of memory-level parallelism and how misses translate into overall performance. Relative slowdown based mapping holds enough information for making a (close to) optimal mapping. Note though that the information needed by relative slowdown mapping is substantial as it requires the knowledge of how well jobs perform on the different core types; this may require extensive profiling which may be not be achievable in practice. Hence, we can consider this approach as an idealized mapping approach. (Note we used the relative slowdown mapping) We conclude from this experiment that job-to-core mapping on heterogeneous multi-core systems is a non-trivial problem, that simple heuristics as presented in the literature are suboptimal, and that solving the mapping problem can yield substantial performance benefits.

3.5.7 Workloads

Any experimental study is bound to the workloads used in the study. In other words, some of the conclusions may be somewhat biased by the set of workloads considered. However, we expect the more general conclusions to still hold true for other types of workloads.

Finding #11: *Although the SPEC CPU benchmark suite comprises a fairly broad set of workloads, as for any experimental study, some of the conclusions reached may be bound by the set of chosen workloads, however, we expect the overall insights to hold true across workload domains.* In particular, we identified specific heterogeneous multicore configurations to be optimal. We expect this to be workload dependent, i.e., another set of workloads is likely to yield a different set of Pareto-optimal architecture configurations. Nevertheless, the more general findings, we believe, are likely to hold true for other workloads as well.

3.6 Related Work

The design space of heterogeneous multi-core architectures is huge. The weakest form of heterogeneity involves different cores only varying in clock frequency (microarchitecture and ISA is the same across the cores); this form of heterogeneity may stem from process variation which may cause cores on the same chip differing substantially in the amount of power that they consume and in the maximum frequency that they can support [Teodorescu and Torrellas 2008]. Per-core DVFS is employed in the AMD Opteron Quad-Core processor [Dorsey et al. 2007] and the Intel Montecito [McGowen et al. 2006] and enables heterogeneity by varying clock frequency per core. A stronger form of heterogeneity are the so-called single-ISA heterogeneous multi-cores: all the cores implement the same ISA but differ in their microarchitecture [Kumar et al. 2003]. Commercial examples are the NVidia Kal-El [NVidia 2011] and the ARM big.LITTLE chip [Greenhalgh 2011], as mentioned in the introduction. Overlapping-ISA heterogeneous multi-cores feature different cores with overlapping ISAs, i.e., all cores implement the same ISA except for a small set of instructions that is unique to each core type [Li et al. 2010a]. The strongest form of heterogeneity involves different cores with different ISAs and microarchitectures. Examples are CPU/GPU integration, such as Intel's Sandy Bridge [Intel 2008], AMD's Fusion [AMD 2008], and NVidia's Tegra [NVidia 2010], or accelerator-based architectures such as the IBM

Cell [Kahle et al. 2005a]. The remainder of this related work section focuses on single-ISA heterogeneous architectures.

Kumar et al. [2008] were the first to propose single-ISA heterogeneous multi-cores. They propose thread migration during run time and powering down unused cores to exploit the time-varying behavior of applications to maximize performance and power-efficiency. In their follow-on work, Kumar et al. [2004] proposed scheduling different programs to different core types in single-ISA heterogeneous multi-core architectures. They showed that scheduling programs to the most power-efficient core in a heterogeneous multi-core processor can lead to substantial improvements in system throughput (weighted speedup) for static workloads and substantial reductions in job response times for dynamic workloads in which jobs come and go as they complete. In contrast to our work, Kumar et al. did not make the observation that heterogeneous architectures fundamentally trade per-program performance for throughput. Kumar et al. [2006] explore principles for designing single-ISA heterogeneous multi-core architectures. They consider in-order as well as out-of-order cores, and they vary core configurations (pipeline width, number of functional units, number of rename registers, reorder buffer size, etc.) as well as cache size and associativity, while considering both area and power cost. In contrast to our work, they limit the design space to four cores only and assume no interactions among cores (no shared LLC). Further, they focus on system throughput only, and do not consider per-program performance.

Kahle et al. [2005b] study the trade-off in per-program performance versus chip throughput in a power-constrained environment. They make the fundamental observation that in order to achieve both high per-program performance and high throughput, a processor needs the ability to dynamically vary the amount of energy expended per instruction according to the amount of parallelism available in software — a technique called Energy Per Instruction (EPI) throttling. They survey four architectural techniques to do so, voltage/frequency scaling, heterogeneity, variable-size cores and speculation control, and conclude that heterogeneous multi-cores along with voltage/frequency scaling are most promising to dynamically achieve high per-program performance when few threads are active and high throughput when many threads are active. Annavaram et al. [2005] experimentally evaluate the idea of EPI throttling using prototype hardware by applying clock throttling to cores in a homogeneous shared-memory processor, effectively creating a heterogeneous multi-core system. They report substantial performance improvements compared to

a homogeneous multi-core within a given power budget while running multi-threaded applications. These papers did not evaluate design trade-offs in a heterogeneous multi-core processor and how these trade-offs affect per-program performance versus throughput. Furthermore, these papers argue workload phases with limited thread-level parallelism should be sped up by consuming more energy per instruction, thereby achieving high per-program performance; in this work, we find that, even under abundant numbers of independent programs and threads, heterogeneous multi-cores provide a trade-off between per-thread performance and chip throughput.

A substantial body of work has been done on scheduling for heterogeneous multi-core processors. Several proposals propose static or offline scheduling based on program characteristics [Chen and John 2009, Shelepov et al. 2009]. An obvious limitation is that static or offline scheduling does not allow for taking advantage of time-varying workload execution behavior. Other proposals employ sampling-based scheduling [Kumar et al. 2004, Becchi and Crowley 2008], i.e., a program is executed on different core types for a short amount of time and the system then dynamically maps the program on the most performance/power-efficient core dynamically. Yet other proposals use heuristics such as schedule memory-intensive programs on small cores and compute-intensive programs on more aggressive cores, see for example [Ghiasi et al. 2005, Shelepov et al. 2009, Koufaty et al. 2010, Li et al. 2010a]. Patsilaras et al. [2010, 2012] study how to best integrate an MLP technique (such as runahead execution [Mutlu et al. 2003]) into a heterogeneous multi-core processor. We refer to the next chapter for more in-depth analysis on scheduling for single-ISA heterogeneous multi-core processors and the proposal of IE scheduling, an accurate and scalable scheduling paradigm which outperforms the state-of-the-art in scheduling by a significant margin.

3.7 Summary

The single-ISA heterogeneous multi-core design space is huge and there are many fundamental design choices to be made. Hence, getting insight in the design space is far from trivial. The core types may vary from simple in-order to complex out-of-order cores, and there are many possible compositions of core types and number of cores. In addition, the design is constrained by chip area limitations as well as limits in off-chip bandwidth, which leads to interesting design trade-offs while considering core types,

number of cores and LLC size. Understanding these design trade-offs is further complicated by the methodology — which is likely the reason why no such a study has not been published before, to the best of our knowledge: heterogeneous multi-core design exploration is complicated by the huge design space, complex interactions through shared resources such as the LLC, the very large number of possible workload mixes, and the sensitivity of the exploration to job-to-core mapping. Clearly, detailed simulation is too slow to be a useful tool for such exploratory analyses.

We used analytical modeling to explore the heterogeneous design space: analytical modeling is fast and allows for exploring many design trade-offs in limited time. The input to the analytical model is obtained in linear time in the number of core types and workloads of interest; all possible combinations of number of cores, core types and workload mixes can be quickly evaluated from this initial profile while taking into account interactions in the shared LLC. Further, analytical modeling facilitates focusing on the major performance trends and insights. In contrast to prior work, we also focus on both system throughput and per-program performance — prior work work focused on system throughput only — and we explore Pareto-optimal configurations.

This analysis provides a number of interesting insights. (1) While it is true that heterogeneity can improve system throughput, it fundamentally trades per-program performance for chip throughput. (2) Some homogeneous multi-core configurations yield optimal performance trade-offs, however, heterogeneity enables making more fine-grained design choices, and yields better throughput and per-program performance than homogeneous designs for particular performance targets. (3) Two core types provide most of the benefits from heterogeneity and a larger number of core types does not contribute much, however, the choice of core types is critical for optimum performance and for achieving particular performance targets. (4) Limited off-chip bandwidth changes some of the fundamental design choices in heterogeneous architectures, such as the need for large on-chip caches for achieving high throughput, and per-program performance degrading more relative to throughput under constrained off-chip bandwidth. Further, while a homogeneous design with many small cores achieves highest throughput assuming infinite bandwidth, only heterogeneous designs can achieve the highest possible throughput under bandwidth constraints. We have shown that job-to-core mapping is both important and challenging for heterogeneous multi-core processors to achieve optimum performance. In the remainder of this dissertation, we will fur-

ther explore the difficulties of scheduling for heterogeneous multi-core processors.

Chapter 4

Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)

Nothing compares to the simple pleasure of a bike ride.
John F. Kennedy

In this chapter, we propose Performance Impact Estimation (PIE) as a mechanism to schedule workloads on a single-ISA heterogeneous multi-cores. PIE collects CPI stack, MLP and ILP profile information at runtime, and estimates the performance impact if the workload were to run on a different core type.

4.1 Introduction

A fundamental problem in the design space of single-ISA heterogeneous multi-core processors is how to best schedule workloads on the most appropriate core type, as clearly shown in the previous chapter. Making wrong scheduling decisions can lead to suboptimal performance and excess energy/power consumption. To address this scheduling problem, recent proposals use workload memory intensity as an indicator to guide application scheduling [Becchi and Crowley 2008, Chen and John 2009, Ghiasi et al. 2005, Koufaty et al. 2010, Li et al. 2010b, Shelepov et al. 2009]. Such proposals tend to schedule memory-intensive workloads on a small core and compute-intensive workloads on a big core. We show that such an ap-

proach causes suboptimal scheduling when memory intensity alone is not a good indicator for workload-to-core mapping.

In general, small (e.g., in-order) cores provide good performance for compute-intensive workloads whose subsequent instructions in the dynamic instruction stream are mostly independent (i.e., high levels of inherent ILP). On the other hand, big (e.g., out-of-order) cores provide good performance for workloads where the ILP must be extracted dynamically or the workload exhibits a large amount of MLP. Therefore, scheduling decisions on heterogeneous multi-cores can be significantly improved by taking into account how well a small or big core can exploit the ILP and MLP characteristics of a workload.

We propose Performance Impact Estimation (PIE) as a mechanism to select the appropriate workload-to-core mapping in a heterogeneous multi-core processor. The key idea of PIE is to estimate the expected performance for each core type for a given workload. In particular, PIE collects CPI stack, MLP and ILP profile information during runtime on any one core type, and estimates performance if the workload were to run on another core type. In essence, PIE estimates how a core type affects exploitable MLP and ILP, and uses the CPI stacks to estimate the impact on overall performance. Dynamic PIE scheduling collects profile information on a per-interval basis (e.g., 2.5 ms) and dynamically adjusts the workload-to-core mapping, thereby exploiting time-varying execution behavior. We show that dynamically collecting profile information requires minimal hardware support: five 10-bit counters and 64 bits of storage.

We evaluate PIE scheduling using a large number of multi-programmed SPEC CPU2006 workload mixes. For a set of scheduling-sensitive workload mixes on a heterogeneous multi-core consisting of one big (out-of-order) and one small (in-order) core, we report an average performance improvement of 5.5% over recent state-of-the-art scheduling proposals. We also evaluate PIE scheduling and demonstrate its scalability across a range of heterogeneous multi-core configurations, including private and shared last-level caches (LLCs). Finally, we show that PIE outperforms a sampling-based scheduling by an average of 8.7%.

4.2 Motivation

Efficient use of single-ISA heterogeneous multi-cores is dependent on the underlying workload scheduling policy. A number of recent proposals use

memory intensity as an indicator to guide workload scheduling [Becchi and Crowley 2008, Chen and John 2009, Ghiasi et al. 2005, Koufaty et al. 2010, Li et al. 2010b, Shelepov et al. 2009]. This policy is based on the intuition that compute-intensive workloads benefit more from the high computational capabilities of a big core while memory-intensive workloads execute more energy-efficiently on a small core while waiting for memory.

To correlate whether memory intensity is a good indicator to guide workload scheduling, Figure 4.1 compares the slowdown for SPEC CPU2006 workloads on a small core relative to a big core (left y-axis), to the normalized CPI stack [Emma 1997] on a big core (right y-axis). The normalized CPI stack indicates whether a workload is memory-intensive or compute-intensive. If the normalized CPI stack is memory dominant, then the workload is memory-intensive (e.g., mcf), else the workload is compute-intensive (e.g., tonto).

The figure illustrates workloads grouped into three categories on the x-axis: workloads that have reasonable slowdown ($<1.75\times$) on the small core (type-I workloads), workloads that have significant slowdown ($>2.25\times$) on the small core (type-III), and the remaining workloads are labeled as type-II. Making correct scheduling decisions in the presence of type-I and III workloads is most critical: making an incorrect scheduling decision, i.e., executing a type-III workload on a small core instead of a type-I workload, leads to poor overall performance, hence we label type-I and III as *scheduling-sensitive* workloads.

The figure shows that while memory intensity alone can provide a good indicator for scheduling some memory-intensive workloads (e.g., mcf) onto a small core, such practice can significantly slowdown other memory-intensive workloads (e.g., soplex). Similarly, some compute-intensive workloads (e.g., astar.r) observe a significant slowdown on a small core while other compute-intensive workloads (e.g., calculix) have reasonable slowdown when executing on a small core. This behavior illustrates that memory intensity (or compute intensity) alone is not a good indicator to guide application scheduling on heterogeneous multi-cores.

The performance behavior of workloads on small and big cores (Figure 4.1) can be explained by the design characteristics of each core. Big cores are particularly suitable for workloads that require ILP to be extracted dynamically or have a large amount of MLP. On the other hand, small cores are suitable for workloads that have a large amount of inherent ILP. This implies that performance on different core types can be directly correlated to the amount of MLP and ILP prevalent in the workload. For example,

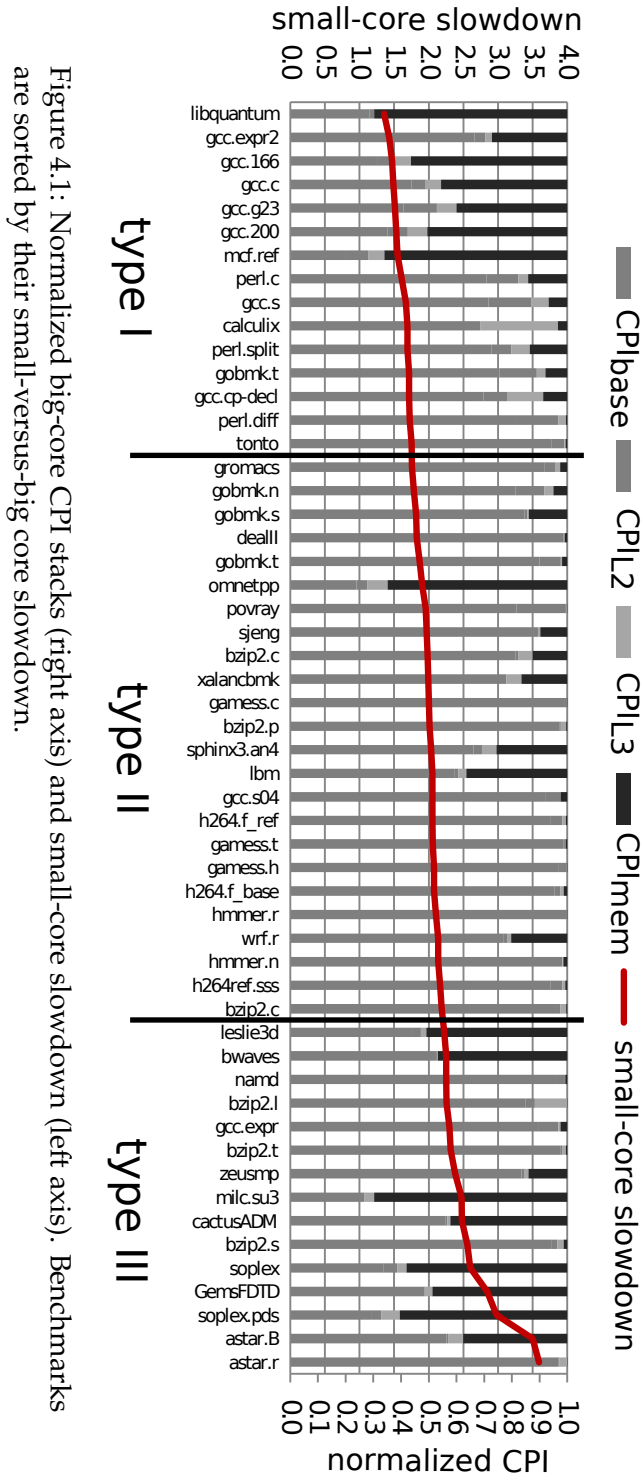


Figure 4.1: Normalized big-core CPI stacks (right axis) and small-core slowdown (left axis). Benchmarks are sorted by their small-versus-big core slowdown.

consider a memory-intensive workload that has a large amount of MLP. Executing such a memory-intensive workload on a small core can result in significant slowdown if the small core does not expose the MLP. On the other hand, a compute-intensive workload with large amounts of ILP may have a reasonable slowdown on a small core and need not require the big core.

To quantify this, Figure 4.2 illustrates slowdown and the loss in MLP (or ILP) when scheduling a workload on a small core instead of a big core. The workloads are sorted left-to-right based on memory intensity (inferred from the normalized CPI stack). We use *MLP ratio* to quantify MLP loss and *ILP ratio* to quantify ILP loss. MLP and ILP ratios are defined as follows:

$$MLP_{ratio} = MLP_{big}/MLP_{small} \tag{4.1}$$

$$ILP_{ratio} = CPI_{base\_big}/CPI_{base\_small} \tag{4.2}$$

with MLP defined as the average number of outstanding memory requests if at least one is outstanding [Chou et al. 2004], and CPI_{base} as the base (non-miss) component of the CPI stack. The key observation from Figure 4.2 is that MLP ratio correlates with slowdown for memory-intensive applications (righthand side of the graph). Similarly, ILP ratio correlates with slowdown for compute-intensive workloads (lefthand side of the graph).

In summary, Figures 4.1 and 4.2 indicate that memory intensity alone is not a good indicator for scheduling workloads on a heterogeneous multi-core. Instead, scheduling policies on heterogeneous multi-cores must take into account the amount of MLP and ILP that can be exploited by the different core types. Furthermore, the slowdowns (or speedups) when moving between different core types can directly be correlated to the amount of MLP and ILP realized on a target core. This suggests that the performance on a target core type can be estimated by predicting the MLP and ILP on that core.

4.3 Performance Impact Estimation (PIE)

A direct approach to determine the best scheduling policy on a heterogeneous multi-core is to apply sampling-based scheduling [Becchi and Crowley 2008, Kumar et al. 2003; 2004]. Sampling-based scheduling dynamically

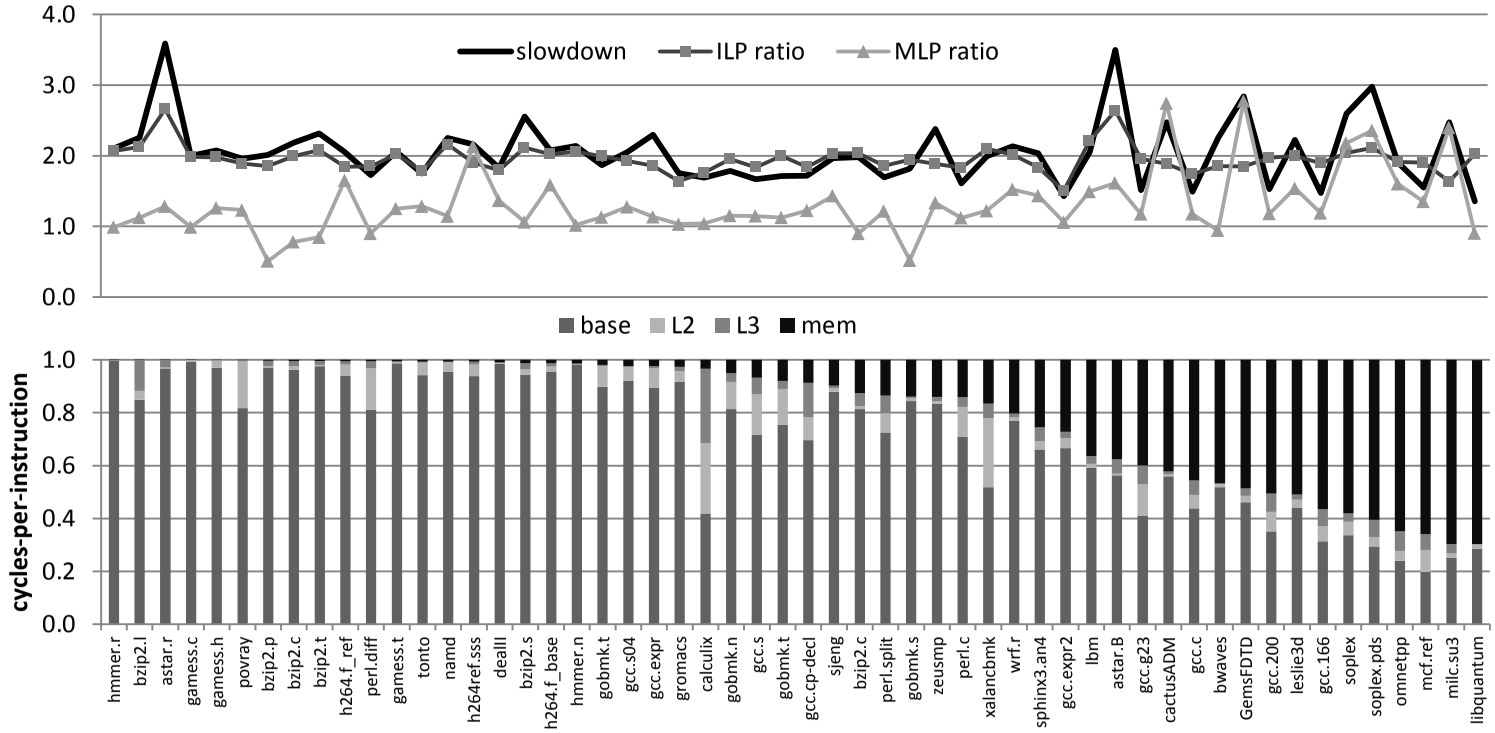


Figure 4.2: Correlating small-core slowdown to the MLP ratio for memory-intensive workloads (right-hand side in the graph) and to the ILP ratio for the compute-intensive workloads (lefthand side in the graph). Workloads are sorted by their normalized memory CPI component (bottom graph).

samples different workload-to-core mappings at runtime and then selects the best performing mapping. While such an approach can perform well, it introduces performance overhead due to periodically migrating workloads between different core types. Furthermore, these overheads increase with the number of cores (and core types). To address these drawbacks, we propose *Performance Impact Estimation (PIE)*.

The key idea behind PIE is to estimate (not sample) workload performance on a different core type. PIE accomplishes this by using CPI stacks. We concentrate on two major components in the CPI stack: the base component and the memory component; the former lumps together all non-memory related components:

$$CPI = CPI_{base} + CPI_{mem}. \tag{4.3}$$

Figure 4.2 illustrated that MLP and ILP ratios provide good indicators on the performance difference between big and small cores. Therefore, we use MLP, ILP, and CPI stack information to develop our PIE model (see Figure 4.3). Specifically, we estimate the performance on a small core while executing on a big core in the following manner:

$$\begin{aligned} CPI_{small} &= \widetilde{CPI}_{base\_small} + \widetilde{CPI}_{mem\_small} \\ &= \widetilde{CPI}_{base\_small} + CPI_{mem\_big} \times MLP_{ratio}. \end{aligned} \tag{4.4}$$

Similarly, we estimate the performance on a big core while executing on a small core as follows:

$$\begin{aligned} CPI_{big} &= \widetilde{CPI}_{base\_big} + \widetilde{CPI}_{mem\_big} \\ &= \widetilde{CPI}_{base\_big} + CPI_{mem\_small}/MLP_{ratio}. \end{aligned} \tag{4.5}$$

In the above formulas, $\widetilde{CPI}_{base\_big}$ refers to the base CPI component on the big core estimated from the execution on the small core; $\widetilde{CPI}_{base\_small}$ is defined similarly. The memory CPI component on the big (small) core is computed by dividing (multiplying) the memory CPI component measured on the small (big) core with the MLP ratio. The remainder of this section details on how we predict the base CPI components as well as the MLP ratio, followed by an evaluation of the PIE model. Section 4.4 then presents dynamic PIE scheduling, including how we collect the inputs to

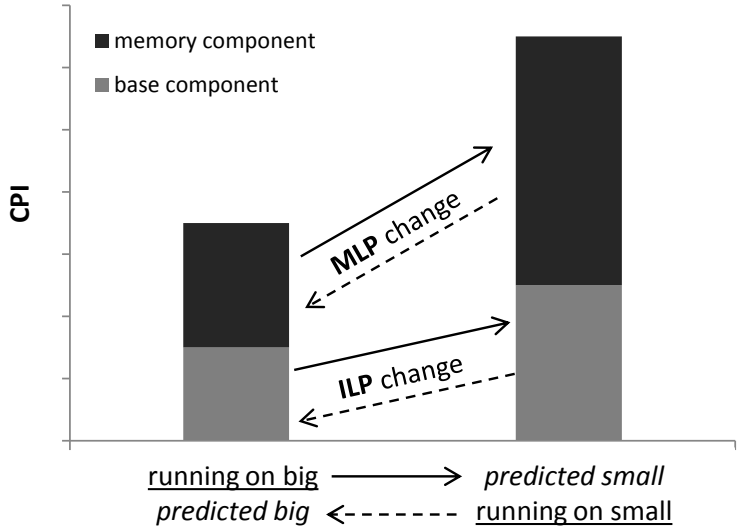


Figure 4.3: Illustration of the PIE model.

the PIE model during runtime by introducing performance counters.

4.3.1 Predicting MLP

The memory CPI component essentially consists of three contributors: the number of misses, the latency per (isolated) miss, and the number of simultaneously outstanding misses (MLP). We assume that the big and small cores have the same cache hierarchy, i.e., the same number of cache levels and the same cache sizes at each level. In other words, we assume that the number of misses and the latency per miss is constant across core types[1]. However, MLP varies across core types as big cores and small cores vary in the amount of MLP that they can exploit. We now describe how we estimate MLP on the big core while running on the small core; and vice versa, we estimate MLP on the small core while running on the big core. Combining these MLP estimates with measured MLP numbers on the current core type enables predicting the MLP ratio using Formula 4.1, which in its turn enables estimating the memory CPI components on the other core type, using Formulas 4.4 and 4.5.

[1]If the cache hierarchy is different, then techniques described in [Jaleel et al. 2012] can be used to estimate misses for a different cache size.

Predicting big-core MLP on small core

Big out-of-order cores implement a reorder buffer, non-blocking caches, MSHRs, etc., which enables issuing independent memory accesses in parallel. The maximum MLP that a big core can exploit is bound by the reorder buffer size, i.e., a necessary condition for independent long-latency load misses to be issued to memory simultaneously is that they reside in the reorder buffer at the same time. We therefore estimate the big-core MLP as the average number of memory accesses in the big-core reorder buffer. Quantitatively, we do so by calculating the average number of LLC misses per instruction observed on the small core (MPI_{small}) multiplied by the big-core reorder buffer size:

$$MLP_{big} = MPI_{small} \times ROB\_size. \tag{4.6}$$

Note that the above estimate does not make a distinction between independent versus dependent LLC misses; we count all LLC misses. A more accurate estimate would be to count independent LLC misses only, however, in order to simplify the design, we simply count all LLC misses.

Predicting small-core MLP on big core

Small in-order cores exploit less MLP than big cores. A stall-on-miss core stalls on a cache miss, and hence, it does not exploit MLP at all — MLP equals one. A stall-on-use core can exploit some level of MLP: independent loads between a long-latency load and its first consumer can be issued to memory simultaneously. MLP for a stall-on-use core thus equals the average number of memory accesses between a long-latency load and its consumer. Hence, we estimate the MLP of a stall-on-use core as the average number of LLC misses per instruction on the big core multiplied by the average dependency distance D between an LLC miss and its consumer. (Dependency distance is defined as the number of dynamically executed instructions between a producer and its consumer.)

$$MLP_{small} = MPI_{big} \times D. \tag{4.7}$$

Again, in order to simplify the design, we approximate D as the dependency distance between any producer (not just an LLC miss) and its consumer. We describe how we measure the dependency distance D in Section 4.4.3.

4.3.2 Predicting ILP

The second CPI component predicted by the PIE model is the base CPI component.

Predicting big-core ILP on small core

We estimate the base CPI component for the big core as one over the issue width W_{big} of the big core:

$$\widetilde{CPI}_{base\_big} = 1/W_{big}. \tag{4.8}$$

A balanced big (out-of-order) core should be able to dispatch approximately W_{big} instructions per cycle in the absence of miss events. A balanced core design can be achieved by making the reorder buffer and related structures such as issue queues, rename register file, etc., sufficiently large to enable the core to issue instructions at a rate near the designed width [Eyerman et al. 2009].

Predicting small-core ILP on big core

Estimating the base CPI component for a small (in-order) core while running on a big core is more complicated. For ease of reasoning, we estimate the average IPC and take the reciprocal of the estimated IPC to yield the estimated CPI. We estimate the average base IPC on the small core with width W_{small} as follows:

$$\widetilde{IPC}_{base\_small} = \sum_{i=1}^{W_{small}} i \times P[IPC = i]. \tag{4.9}$$

We use simple probability theory to estimate the probability of executing i instructions in a given cycle. The probability of executing only one instruction in a given cycle equals the probability that an instruction produces a value that is consumed by the next instruction in the dynamic instruction stream (dependency distance of one):

$$P[IPC = 1] = P[D = 1]. \tag{4.10}$$

Likewise, the probability of executing two instructions in a a given cycle equals the probability that the second instruction does not depend on the

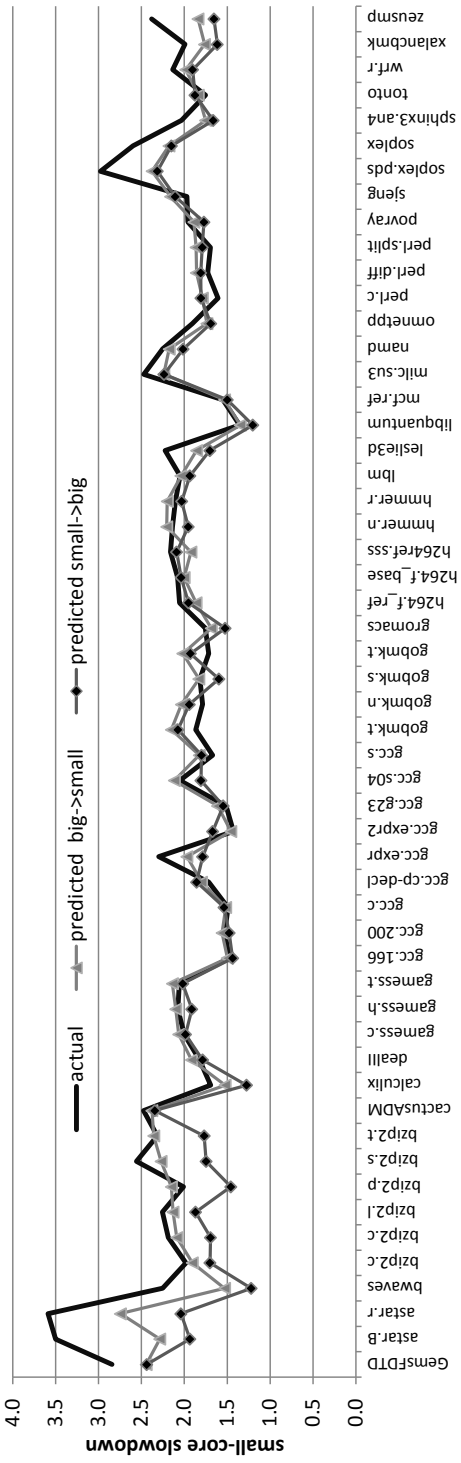


Figure 4.4: Evaluating the accuracy of the PIE model.

first, and the third depends on either the first or the second:

$$P[IPC = 2] = (1 - P[D = 1]) \times (P[D = 1] + P[D = 2]) . \tag{4.11}$$

This generalizes to three instructions per cycle as well:

$$P[IPC = 3] = (1 - P[D = 1]) \times (1 - P[D = 1] - P[D = 2]) \times (P[D = 1] + P[D = 2] + P[D = 3]) . \tag{4.12}$$

Finally, assuming a 4-wide in-order core, the probability of executing four instructions per cycle equals the probability that none of the instructions depend on a previous instruction in a group of four instructions:

$$P[IPC = 4] = (1 - P[D = 1]) \times (1 - P[D = 1] - P[D = 2]) \times (1 - P[D = 1] - P[D = 2] - P[D = 3]) . \tag{4.13}$$

Note that the above formulas do not take non-unit instruction execution latencies into account. Again, we used this approximation to simplify the design, and we found this approximation to be accurate enough for our purpose.

4.3.3 Evaluating the PIE Model

Figure 4.4 evaluates the accuracy of our PIE model. This is done in two ways: we estimate big-core performance while executing the workload on a small core, and vice versa, we estimate small-core performance while executing the workload on a big core. We compare both of these to the actual slowdown. (We will describe the experimental setup in Section 4.5.) The figure shows that we achieve an average absolute prediction error of 9% and a maximum error of 35% when predicting speedup (predicting big-core performance on the small core). The average absolute prediction error for the slowdown (predicting small-core performance on the big core) equals 13% with a maximum error of 47%. More importantly, PIE accurately predicts the relative performance differences between the big and small cores.

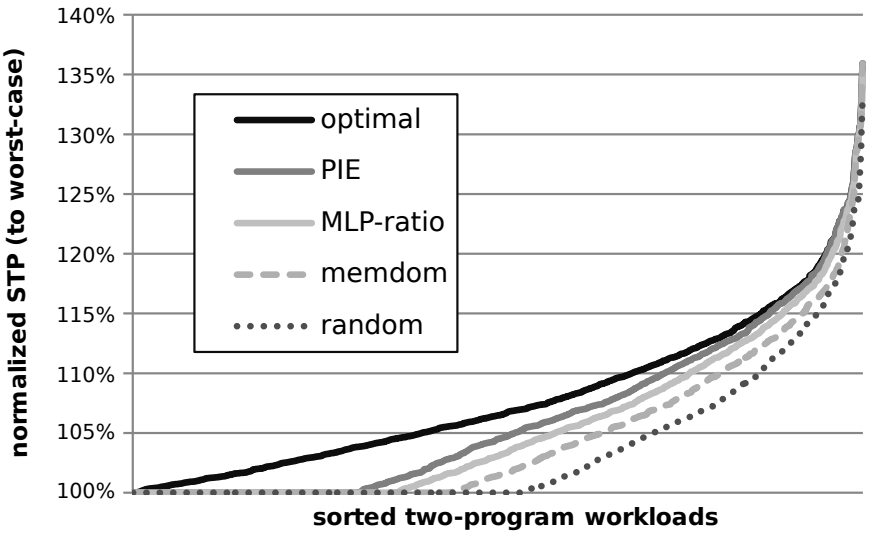


Figure 4.5: Comparing scheduling policies on a two-core heterogeneous multi-core.

This is in line with our goal of using PIE for driving runtime scheduling decisions.

As a second step in evaluating our PIE model, we consider a heterogeneous multi-core and use PIE to determine the workload-to-core mapping. We consider all possible two-core multi-programmed workload mixes of SPEC CPU2006 applications and a two-core system with one big core and one small core and private LLCs. Further, benchmarks are scheduled on a given core and stay there for the remainder of the execution (static scheduling).

Figure 4.5 reports performance (system throughput or weighted speedup) relative to worst-case scheduling for all workload mixes; we compare PIE scheduling against random and memory-dominance (memdom) scheduling. Memory-dominance scheduling refers to the conventional practice of always scheduling memory-intensive workloads on the small core.

PIE scheduling chooses the workload-to-core mapping by selecting the schedule that yields the highest (estimated) system throughput across both cores. PIE scheduling outperforms both random and memory-dominance scheduling over the entire range of workload mixes. Figure 4.6 provides more detailed results for workload mixes with type-I and type-III workloads. PIE outperforms worst-case scheduling by 14.2%, compared to random (8.5%) and memory-dominance scheduling (9.2%). Put differently,

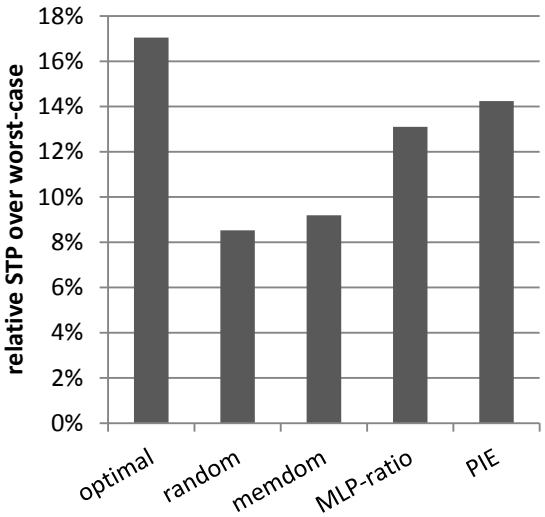


Figure 4.6: Comparing different scheduling algorithms for type-I and type-III workload mixes assuming a static setup.

PIE scheduling achieves 84% of optimal scheduling, compared to 54% for memory-dominance and 50% for random scheduling.

The PIE model takes into account both ILP and MLP. We also evaluated a version of PIE that only takes MLP into account, i.e., ILP is not accounted for and is assumed to be the same on the big and small cores. We refer to this as MLP-ratio scheduling. Figures 4.5 and 4.6 illustrate the importance of taking both MLP and ILP into account. MLP-ratio scheduling improves worst-case scheduling by 12.7% for type-I and III workloads, compared to 14.2% for PIE. This illustrates that accounting for MLP is more important than ILP in PIE.

So far, we evaluated PIE for a heterogeneous multi-core with one big and one small core (e.g., ARM's big.LITTLE design [Greenhalgh 2011]). We now evaluate PIE scheduling for heterogeneous multi-cores with one big core and multiple small cores, as well as several big cores and one small core (e.g., NVidia's Kal-El [NVidia 2011]); we assume all cores are active all the time. Figure 4.7 shows that PIE outperforms memory-dominance scheduling by a bigger margin even for these heterogeneous multi-core design points than for the one-big, one-small multi-core system.

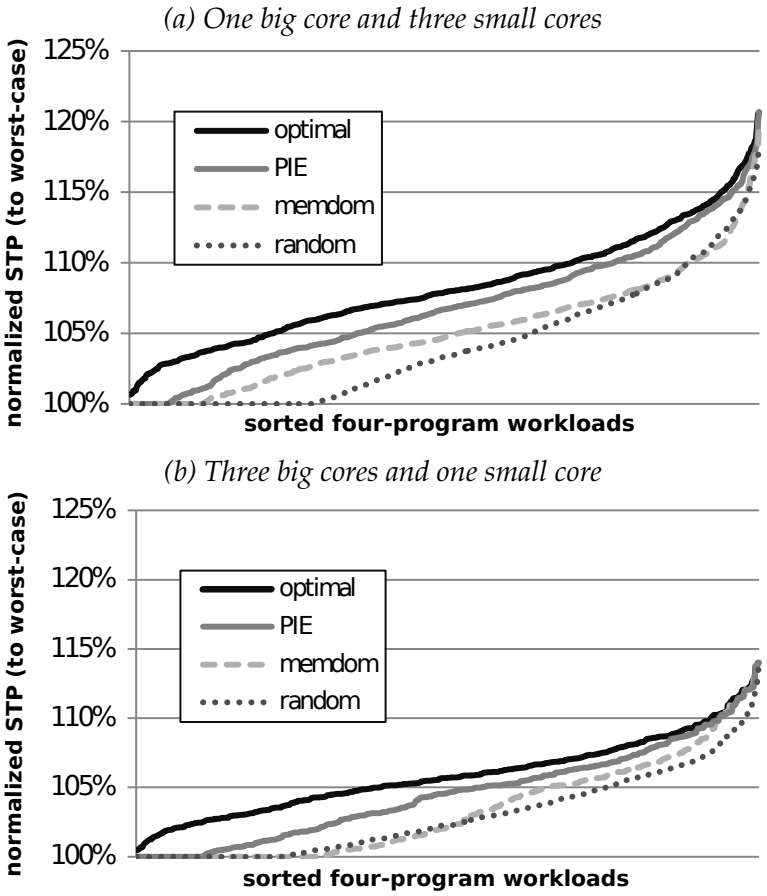


Figure 4.7: Evaluating PIE for heterogeneous multi-core with one big and three small cores (top graph), and three big cores and one small core (bottom graph).

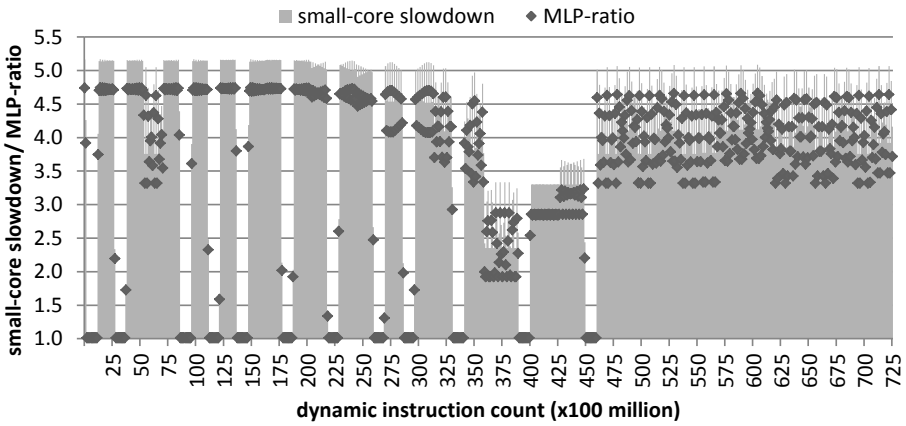


Figure 4.8: Dynamic execution profile of libquantum.

4.4 Dynamic Scheduling

So far, PIE scheduling was evaluated in a static setting, i.e., a workload is scheduled on a given core for its entire execution. There is opportunity to further improve PIE scheduling by dynamically adapting to workload phase behavior. To illustrate this, Figure 4.8 shows big-core and small-core CPI and MLP as a function of time for libquantum from SPEC CPU2006. The key observation here is that, although the average slowdown is high for the small core compared to the big core, the small core achieves comparable performance to the big core for some execution phases. For libquantum, approximately 10% of the instructions can be executed on the small core without significantly affecting overall performance. However, the time-scale granularity is relatively fine-grained (few milliseconds) and much smaller than a typical OS time slice (e.g., 10 ms). This suggests that dynamic hardware scheduling might be beneficial provided that rescheduling (i.e., migration) overhead is low.

4.4.1 Quantifying migration overhead

Dynamic scheduling incurs overhead for migrating workloads between different cores. Not only does migration incur a context switch, it also incurs overhead for warming hardware state, especially the cache hierarchy. A context switch incurs a fixed cost for restoring architecture state. To better understand the overhead due to cache warming, we consider a number of scenarios to gain insight on cache hierarchy designs for low migration overheads at fine-grained dynamic scheduling.

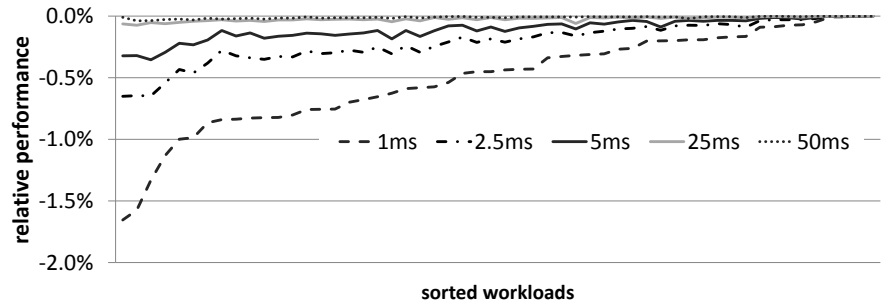


Figure 4.9: Migration overhead for a shared LLC.

Shared LLC. Figure 4.9 quantifies the performance overhead of migrating a workload every x milliseconds, with x varying from 1 ms to 50 ms. Migration overhead is measured by configuring two identical cores to share a 4MB LLC. Workloads are rescheduled to a different core every x ms. Interestingly, for a 2.5 ms migration frequency, the performance overhead due to migration is small, less than 0.6% for all benchmarks. The (small) performance overhead are due to (private) L1 and L2 cache warmup effects.

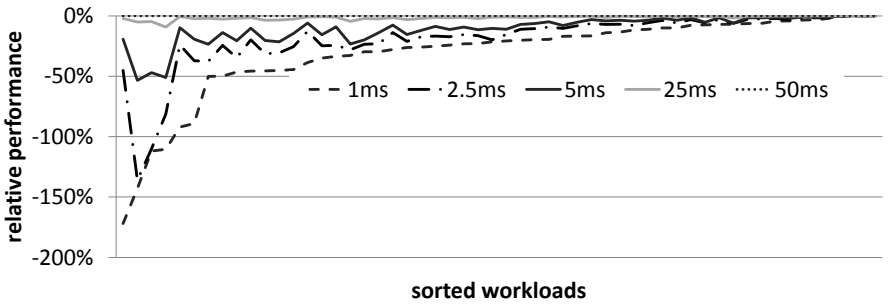


Figure 4.10: Migration overhead for private powered-off LLCs.

Private powered-off LLCs. The situation is very different in case of a private LLC that is powered off when migrating a workload. Powering off a private LLC makes sense in case one wants to power down an entire core and its private cache hierarchy in order to conserve power. If the migration frequency is high (e.g., 2.5 ms), Figure 4.10 reports severe performance overhead for some workloads when the private cache hierarchy is powered off upon migration. The huge performance overheads are because the cache looses its data when powered off, and hence the new core must refetch the data from main memory.

Private powered-on LLCs. Instead of turning off private LLCs, an alternative is to keep the private LLCs powered on and retain the data in the cache. In doing so, Figure 4.11 shows that performance overhead from frequent migrations is much smaller and in fact even leads to substantial performance benefits for a significant fraction of the benchmarks. The performance benefit comes from having a larger effective LLC: upon a miss in the new core's private LLC, the data is likely to be found in the old core's private LLC, and hence the data can be obtained more quickly from the old core's LLC through cache coherency than by fetching the data from main memory.

4.4.2 Dynamic PIE Scheduling

Having described the PIE model, we now describe Dynamic PIE scheduling. PIE scheduling is applicable to any number of cores of any core type. However, to simplify the discussion, we assume one core of each type. We assume as many workloads as there are cores, and that workloads are initially randomly scheduled onto each core. Furthermore, we assume that workload scheduling decisions can be made every x milliseconds.

To strive towards an optimal schedule, PIE scheduling requires hardware support for collecting CPI stacks on each core, the number of misses, the number of dynamically executed instructions, and finally the inter-instruction dependency distance distribution on the big core. We discuss the necessary hardware support in the next section.

During every time interval of x milliseconds, for each workload in the system, PIE uses the hardware support to compute CPI stacks, MLP and ILP on the current core type, and also predicts the MLP and ILP for the same workload on the other core type. These predictions are then fed into the PIE model to estimate the performance of each workload on the other core type. For a given performance metric, PIE scheduling uses these estimates to determine whether another scheduling decision would potentially improve overall system performance as compared to the current schedule. If so, workloads are rescheduled to the predicted core type. If not, the workload schedule remains intact and the process is repeated the next time interval.

Note that PIE scheduling can be done both in hardware and software. If the time interval of scheduling workloads to cores coincides with a time slice, then PIE scheduling can be applied in software, i.e., the hardware would collect the event counts and the software (e.g., OS or hypervisor)

Figure 4.11: Migration overhead for private powered-on LLCs.

would make scheduling decisions. If scheduling decisions would need to be made at smaller time scale granularities, hardware can also make the scheduling decisions, transparent to the software [Greenhalgh 2011].

4.4.3 Hardware support

PIE scheduling requires hardware support for collecting CPI stacks. Collecting CPI stacks on in-order cores is fairly straightforward and is implemented in commercial systems, see for example Intel Atom [Halfhill 2008]. Collecting CPI stacks on out-of-order cores is more complicated because of various overlap effects between miss events, e.g., a long-latency load may hide the latency of another independent long-latency load miss or mispredicted branch, etc. Recent commercial processors such as IBM Power5 [Mericas 2006] and Intel Sandy Bridge [Intel 2008] however provide support for computing memory stall components. PIE scheduling also requires the number of LLC misses and the number of dynamically executed instructions, which can be measured using existing hardware performance counters. In other words, most of the profile information needed by PIE can be readily measured on existing hardware.

PIE scheduling requires some profile information that cannot be collected on existing hardware. For example, while running on a big core, PIE requires the ability to measure the inter-instruction dependency distance distribution for estimating small-core MLP and ILP. The PIE model requires the dependency distance distribution for a maximum dependency distance of W_{small} only (where W_{small} is the width of the small core). For a 4-wide core, this involves four plus one counters: four counters for computing the dependency distance distribution up to four instructions, and one counter for computing the average distance.

The PIE model requires that the average dependency distance D be computed over the dynamic instruction stream. This can be done by requiring a table with as many rows as there are architectural registers. The table keeps track of which instruction last wrote to an architectural register. The delta in dynamic instruction count between a register write and subsequent read then is the dependency distance. Note that the table counters do not need to be wide, because the dependency distance tends to be short [Franklin and Sohi 1992]; e.g., four bits per counter can capture 90% of the distances correctly. In summary, the total hardware cost to track the dependency distance distribution is roughly 15 bytes of storage: 4 bits times the number of architectural registers (64 bits for x86-64), plus five 10-bit counters.

4.5 Experimental Setup

We use CMP$im [Jaleel et al. 2008a] to conduct the simulation experiments in this chapter[2]. We configure our simulator to model heterogeneous multi-core processors with big and small cores. The big core is a 4-wide out-of-order processor core; the small core is a 4-wide (stall-on-use) in-order processor core. We also ran experiments with a 2-wide in-order processor and found the performance for the 2-wide in-order processor to be within 10% of the 4-wide in-order processor, which is a very small compared to the 200%+ performance difference between in-order versus out-of-order processor performance. Hence, we believe that our conclusions hold true irrespective of the width of the in-order processor. We assume both cores run at a 2 GHz clock frequency. Further, we assume a cache hierarchy consisting of three levels of cache, separate 32 KB L1 instruction and data caches, a 256 KB L2 cache and a 4 MB last-level L3 cache (LLC). We assume the L1 and L2 caches to be private per core for all the configurations evaluated in this chapter. We evaluate both shared and private LLC configurations. We consider the LRU replacement policy in all of the caches unless mentioned otherwise; we also consider a state-of-the-art RRIP shared cache replacement policy [Jaleel et al. 2010]. Finally, we assume an aggressive stream-based hardware prefetcher; we experimentally evaluated that hardware prefetching improves performance by 47% and 25% on average for the small and big cores, respectively.

[2]The version we use during this chapter is different from the one used in the previous chapters. This work was done during an internship at Intel, Hudson. While at Intel, we had access to the internal version of CMP$im.

We further assume that the time interval for dynamic scheduling is 2.5 ms; this is small enough to benefit from fine-grained exploitation of time-varying execution behavior while keeping migration overhead small. The overhead for migrating a workload from one core to another (storing and restoring the architecture state) is set to 300 cycles; in addition, we do account for the migration overhead due to cache effects.

We consider all 26 SPEC CPU2006 programs and all of their reference inputs, leading to 54 benchmarks in total. We select representative simulation points of 500 million instructions each using PinPoints [Patil et al. 2004]. When simulating a multi-program workload we stop the simulation when the slowest workload has executed 500 million instructions. Faster running workloads are reiterated from the beginning of the simulation point when they reach the end. We report system throughput (STP) [Eyerman and Eeckhout 2008] (also called weighted speedup [Snavely and Tullsen 2000]) which quantifies system-level performance or aggregate throughput achieved by the system.

4.6 Results and Analysis

We evaluate dynamic PIE scheduling on private and shared LLCs with LRU, and a shared LLC with RRIP replacement. We compare PIE scheduling to a sampling-based strategy [Becchi and Crowley 2008, Kumar et al. 2003; 2004] that assumes running a workload for one time interval on one core and for the next time interval on the other core. The workload-core schedule that yields the highest performance is then maintained for the next 10 time intervals, after which the sampling phase is reinitiated.

4.6.1 Private LLCs

We first assume that each core has its private LLC. Figure 4.12 quantifies the relative performance over random scheduling for sampling-based, memory-dominance and PIE scheduling. PIE scheduling clearly outperforms the other scheduling strategies by a significant margin. Across the type-I and III workload mixes, we report an average 5.5% and 8.7% improvement in performance over memory-dominance and sampling-based scheduling, respectively. The improvement over memory-dominance scheduling comes from two sources: PIE is able to more accurately determine the better workload-to-core mapping, and in addition, PIE can exploit fine-grain phase behavior, unlike memory-dominance scheduling.

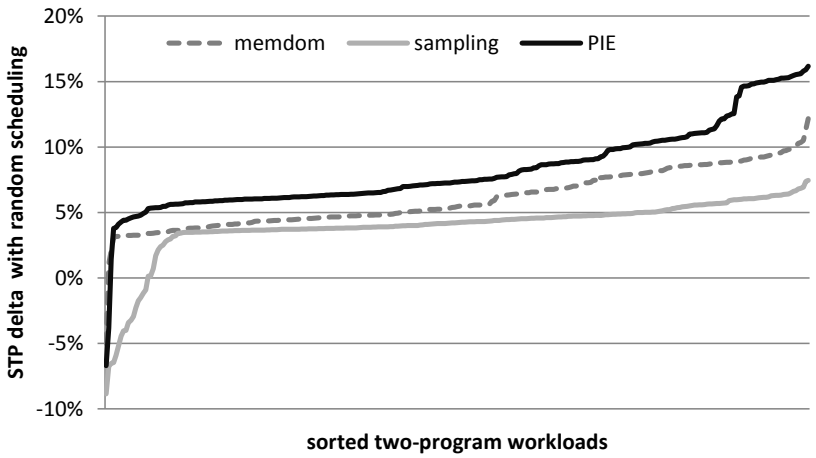


Figure 4.12: Relative performance (STP) delta over random scheduling for sampling-based, memory-dominance and PIE scheduling, assuming private LLCs.

PIE also improves upon sampling-based scheduling, because PIE does not incur any overhead from sampling because it (accurately) estimates the performance impact of a workload reschedule, and hence, it can more quickly and better adapt to fine-grain phase behavior.

4.6.2 Shared LLC

With shared LLCs, Figure 4.13 shows similar conclusions to private LLCs: PIE outperforms random, sampling-based and memory-dominance scheduling. For the type-I and III workload mixes, we obtain an average 3.7% and 6.4% improvement in performance over memory-dominance and sampling-based scheduling, respectively. The performance improvement is slightly lower for private LLCs though. The reason is that none of the scheduling strategies anticipate conflict behavior in the shared LLC, and, as a result, some of the scheduling decisions may be partly offset by negative conflict behavior in the shared LLC. Further, in the case of sampling-based scheduling, LLC performance changes when switching between core types (as a result of sampling) because the access patterns change, which in turn changes overall performance; in other words, sampling is particularly ineffective in case of a shared LLC.

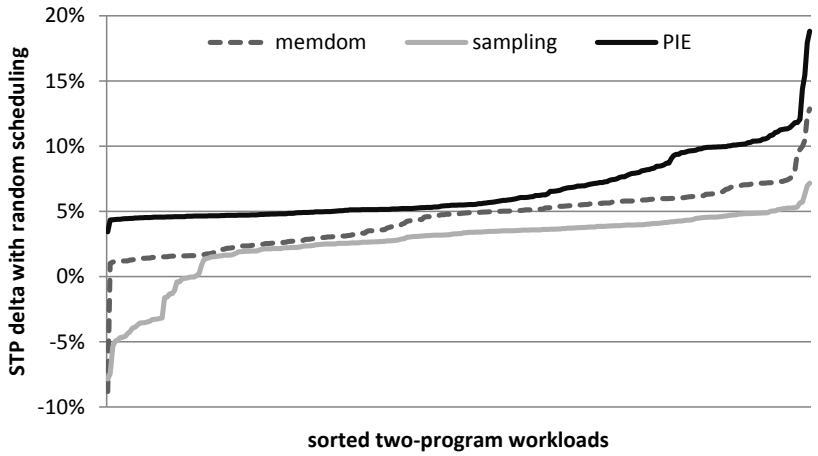


Figure 4.13: Relative performance (STP) delta over random scheduling for sampling-based, memory-dominance and PIE scheduling, assuming an LRU-managed shared LLC.

4.6.3 RRIP-managed shared LLC

So far, we assumed an LRU cache replacement policy. However, it has been shown that LRU is not the most effective shared cache management policy; a state-of-the-art shared cache replacement policy is RRIP [Jaleel et al. 2010] which significantly improves LLC performance by predicting the re-reference behavior of cache blocks. The results for PIE scheduling applied to an RRIP-managed LLC are shown in Figure 4.14. For the type-I and III workload mixes, PIE scheduling improves performance by 2.4% and 7.8% over memory-dominance and sampling-based scheduling, respectively.

An interesting observation to make from Figure 4.14 is that an intelligent shared cache management policy such as RRIP is able to reduce the performance hit observed for some of the workloads due to scheduling. A large fraction of the workloads observe a significant performance hit under sampling-based scheduling (and a handful workloads under memory-dominance scheduling) for an LRU-managed shared LLC, see bottom left in Figure 4.13; these performance hits are removed through RRIP, see Figure 4.14. In other words, a scheduling policy can benefit from an intelligent cache replacement policy: incorrect decisions by the scheduling policy can be alleviated (to some extent) by the cache management policy.

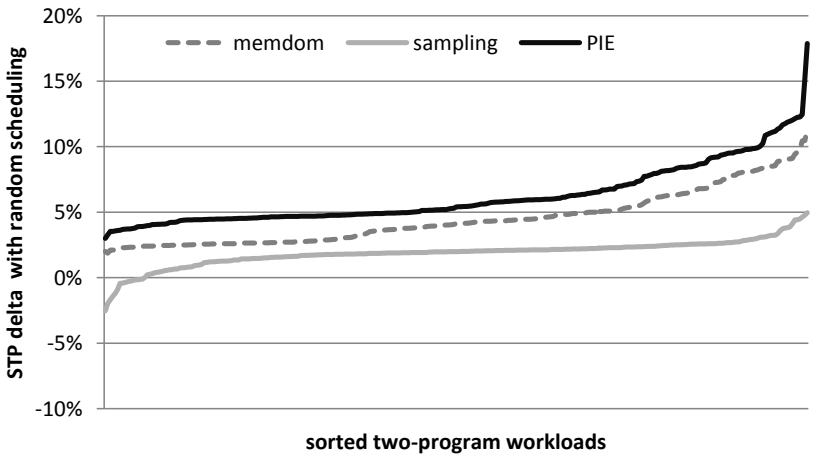


Figure 4.14: Relative performance (STP) delta over random scheduling for sampling-based, memory-dominance and PIE scheduling, assuming an RRIP-managed shared LLC.

4.7 Related Work

Heterogeneous multi-cores desings vary from single-ISA cores only varying in clock frequency, to single-ISA cores differing in microarchitecture, to cores with non-identical ISAs. Since we focus on single-ISA heterogeneous multi-cores, we only discuss this class of heterogeneity.

Kumar et al. [2003] made the case for heterogeneous single-ISA multi-core processors when running a single application: they demonstrate that scheduling an application across core types based on its time-varying execution behavior can yield substantial energy savings. They evaluate both static and dynamic scheduling policies. In their follow-on work, Kumar et al. [2004] study scheduling on heterogeneous multi-cores while running multi-program workloads. The dynamic scheduling policies explored in these studies use sampling to gauge the most energy-efficient core. Becchi and Crowley [2008] also explore sample-based scheduling. Unfortunately, sample-based scheduling, in contrast to PIE, does not scale well with increasing core count: an infrequent core type (e.g., a big core in a one-big, multiple-small core configuration) quickly becomes a bottleneck.

Bias scheduling [Koufaty et al. 2010] is very similar to memory-dominance scheduling. It schedules programs that exhibit frequent memory and other resource stalls on the small core, and programs that are dominated by execution cycles (and hence low fraction of stalls) on the big core. Thresholds are used to determine a program's bias towards a big versus small core

based on these stall counts.

HASS [Shelepov et al. 2009] is a static scheduling policy, the key motivation being scalability. Chen and John [2009] leverage offline program profiling. An obvious limitation of static/offline scheduling is that it does not enable exploiting time-varying execution behavior. PIE on the other hand is a dynamic scheduling algorithm that, in addition, is scalable.

Several studies [Ghiasi et al. 2005, Shelepov et al. 2009] explore scheduling in heterogeneous systems by changing clock frequency across cores; the core microarchitecture does not change though. Such studies do not face the difficulty of having to deal with differences in MLP and ILP across core types. Hence, memory-dominance based scheduling is likely to work well for such architectures.

Age-based scheduling [Lakshminarayana et al. 2009] predicts the remaining execution time of a thread in a multi-threaded program and schedules the oldest thread on the big core. Li et al. [2007] evaluate the idea of scheduling programs on the big core first, before scheduling programs on the small cores, in order to make sure the big power-hungry core is fully utilized.

Chou et al. [2004] explored how microarchitecture techniques affect MLP. They found that out-of-order processors can better exploit MLP compared to in-order processors. We show that MLP and ILP are important criteria to take into account when scheduling on heterogeneous multi-cores, and we propose the PIE method for doing so.

Patsilaras et al. [2010, 2012] study how to best integrate an MLP technique (such as runahead execution [Mutlu et al. 2003]) into an asymmetric multi-core processor, i.e., should one integrate the MLP technique into the small or big core, or both? They found that if the small core runs at a higher frequency and implements an MLP technique, the small core might become more beneficial for exploiting MLP-intensive workloads. Further, they propose a hardware mechanism to dynamically schedule threads to core types based on the amount of MLP in the dynamic instruction stream, which they estimate by counting the number of LLC misses in the last 10 K instructions interval. No currently shipping commercial processor employs runahead execution; also, running the small core at a high frequency might not be possible given current power concerns. We therefore take a different approach: we consider a heterogeneous multi-core system as a given — we do not propose changing the architecture nor the frequency of either core type — and we schedule tasks onto the most appropriate core type to improve overall performance while taking both MLP and ILP into account as

a criterion for scheduling. After our publication, additional work has been published by Lukefahr et al. [2012] who propose composite cores: a heterogeneous multi-core architecture that consists of tightly coupled big and little cores. This allows for even lower overhead context switching, thus for more fine-grained power and performance trade-offs. Much like our PIE scheduler, Lukefahr et al. advocate the use of a model-based scheduling algorithm to enable scalability.

4.8 Summary

Single-ISA heterogeneous multi-cores are typically composed of small (e.g., in-order) cores and big (e.g., out-of-order) cores. Using different core types on a single die has the potential to improve energy-effiency without sacrificing significant performance. However, the success of heterogeneous multi-cores is directly dependent on how well a scheduling policy maps workloads to the best core type (big or small). Incorrect scheduling decisions can unnecessarily degrade performance and waste energy/power. In this chapter we make the following contributions:

- We show that using memory intensity alone as an indictator to guide workload scheduling decisions can lead to suboptimal performance. Instead, scheduling policies must take into account how a core type can exploit the ILP and MLP characteristics of a workload.
- We propose the Performance Impact Estimation (PIE) model to guide workload scheduling. The PIE model uses CPI stack, ILP and MLP information of a workload on a given core type to estimate the performance on a different core type. We propose PIE models for both small (in-order) and big (out-of-order) cores.
- Using the PIE model, we propose dynamic PIE scheduling. Dynamic PIE collects CPI stack, ILP and MLP information at run time to guide workload scheduling decisions.
- We show that the use of shared LLCs can enable high frequency, low-overhead, fine-grained scheduling to exploit time-varying execution behavior. We also show that the use of private LLCs can provide similar capability as long as the caches are not flushed on core migrations.

We evaluate PIE for a variety of systems with varying core counts and cache configurations. Across a large number of scheduling-sensitive work-

loads, we show that PIE scheduling is scalable to any core count and outperforms prior work by a significant margin.

In this chapter, we focused on using PIE scheduling to improve the weighted speedup metric for a heterogeneous multi-core systems and the evaluations were primarily done for multi-programmed workload mixes. In the next chapter, we will explore scheduling for heterogeneous multi-cores even further. We will show that PIE can be applied to multi-threaded applications as well and that scheduling can be done for other (system) metrics than weighted speedup and still provide good throughput.

Chapter 5

Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores

Training is realizing that pain is temporary and that quiting is forever.
Lance Armstrong

In this chapter, we propose fairness-aware scheduling for single-ISA heterogeneous multi-cores, and explore two flavors for doing so. Equal-time scheduling runs each thread on each core type for an equal fraction of the time, whereas equal-progress scheduling strives at getting equal amounts of work done on each core type. Our experimental results demonstrate an average 14% (and up to 25%) performance improvement over pinned scheduling through fairness-aware scheduling for homogeneous multi-threaded workloads; equal-progress scheduling improves performance by 32% on average for heterogeneous multi-threaded workloads. Further, we report dramatic improvements in fairness over prior scheduling proposals for multi-program workloads, while achieving comparable levels of system throughput compared to throughput-optimized scheduling, and an average 21% improvement in throughput over pinned scheduling.

5.1 Introduction

The previous chapter focused on optimizing total system throughput and, to the best of our knowledge, none of the prior work considered fairness as an optimization target. Yet, fairness, or guaranteeing that all threads and/or programs make equal progress, is of great importance. In a multi-

threaded workload, a thread that gets to run on a big core will just wait stalling on a barrier until all other threads running on the small cores have reached the barrier — yielding no performance benefit from heterogeneity. Guaranteeing fairness, or making sure all threads make equal progress, will lead to all threads reaching the barrier at nearly the same time, thereby improving overall application performance. For multi-program workloads, fairness is of utmost importance when it comes to system-level priorities and quality-of-service (QoS). In particular, system software (e.g., the operating system or the virtual machine monitor) essentially assumes all threads (or programs) make equal progress when run on the hardware. Yet, a thread/program that runs on a big core gets more work done than when run on a small core.

Leveraging existing multi-core schedulers on heterogeneous multi-cores does not provide fairness either. Schedulers in modern operating systems affinitize or pin threads or jobs to cores in order to minimize overhead of context switching and increase data locality [Jones 2006]. As a result, a thread or a job that ends up being pinned to a big core will make faster progress compared to threads or jobs that are pinned to small cores, leading to poor fairness. We will refer to this scheduling policy as pinned scheduling throughout the chapter.

In this chapter, we propose fairness-aware scheduling for single-ISA heterogeneous multi-cores. We consider a number of mechanisms for achieving fairness. *Equal-time* scheduling strives at scheduling all threads onto a big core for an equal amount of time. Because equal time does not necessarily lead to equal progress, especially for heterogeneous workloads in which threads exhibit different execution behavior, we also propose *equal-progress* scheduling which strives at getting all threads to make equal progress. We consider three different ways for estimating progress: sampling, history and model-based estimations. Finally, we also explore tunable scheduling policies that trade off system throughput versus fairness. All of these scheduling strategies monitor a thread's progress or time during run-time, and dynamically reschedule threads to improve fairness. Fairness-aware scheduling not only improves fairness over pinned scheduling, it also improves system throughput by enabling threads to run on a big core type for some fraction of time. Further, it achieves a level of system throughput that is comparable to throughput-optimized scheduling as proposed in prior work, while dramatically improving fairness.

Our experimental evaluation includes both multi-threaded and multi-program workloads across a range of heterogeneous multi-core architec-

tures. We report average performance improvements of 14% (and up to 25%) for the multi-threaded workloads through fairness-aware scheduling. Equal-progress scheduling improves performance by 32% on average for heterogeneous multi-threaded workloads; equal-time and equal-progress scheduling perform equally well on homogeneous multi-threaded workloads in which all threads run the same code. For multi-program workloads on a heterogeneous multi-core with one big and three small cores, fairness-aware scheduling achieves an average fairness level of 86%, a significant improvement over pinned and throughput-optimized scheduling with fairness levels of 56% and 64%, respectively. Moreover, fairness-aware scheduling improves system throughput by 21% on average over pinned scheduling, while being within 3.6% on average compared to throughput-optimized scheduling. Scheduling that trades off fairness for throughput enables reducing the maximum throughput reduction compared to throughput-optimized scheduling while achieving similar levels of fairness compared to fairness-aware scheduling. Overall, these results demonstrate that fairness-aware scheduling is key to optimizing performance on single-ISA heterogeneous multi-cores for both multi-threaded and multi-program workloads.

5.2 Motivation

Before elaborating on fairness-aware scheduling, we now motivate the need for fairness for both multi-threaded and multi-program workloads on heterogeneous multi-cores.

5.2.1 Fairness

We first define fairness for heterogeneous multi-cores, along the lines of prior definitions of fairness in multi-threaded and multi-core systems [Eyerman and Eeckhout 2008, Gabor et al. 2007]. We denote T_{het} as the number of cycles to execute a thread on the heterogeneous multi-core when run simultaneously with other threads or applications; T_{big} is defined as the time it takes to execute on the big core (of the same heterogeneous multi-core) when run in isolation. The slowdown of thread i on the heterogeneous multi-core is then defined as the slowdown when running on the heterogeneous multi-core compared to running on the big core is isolation:

$$S_i = \frac{T_{het,i}}{T_{big,i}}. \tag{5.1}$$

We define a schedule to be fair if the slowdowns of all (equal-priority) threads running simultaneously on the heterogeneous multi-core are the same, similarly to prior work on fairness [Eyerman and Eeckhout 2008, Gabor et al. 2007]. A frequently used metric for fairness is to compute the ratio of the minimum versus maximum slowdowns among all simultaneously running threads. One problem with this metric is that it only considers the outlier threads, and does not take into account the 'average' thread. We therefore propose and use a different metric in our work, which is based on the statistically well-founded coefficient of variation:

$$fairness = 1 - \frac{\sigma_S}{\mu_S}. \tag{5.2}$$

μ_S is the average slowdown across all threads, and σ_S is the standard deviation across all slowdowns of all threads. The fraction σ_S/μ_S is the so-called coefficient of variation and measures the variability in slowdown in relation to the mean slowdown — hence, it is a measure for the unfairness, i.e., the larger the variability in slowdown, the more unfair the execution is. One minus the coefficient of variation then is a measure for fairness. Fairness is a higher-is-better metric, and a fairness of one means that all threads make equal progress, relative to running on the big core in isolation.

5.2.2 Multi-threaded workloads

In order to illustrate the importance of achieving fairness, we first consider a number of multi-threaded workloads from the Phoenix [Ranger et al. 2007] and PARSEC [Bienia et al. 2008] benchmark suites, and run these workloads on a heterogeneous multi-core with one big and three small cores (1B3S); the last-level cache is shared among all four cores. (We refer to later for a detailed description of the experimental setup.) We pin each thread to a core, and compare heterogeneous multi-core performance against homogeneous multi-cores with four big (4B) versus four small cores (4S), see Figure 5.1. A homogeneous multi-core with big cores achieves a speedup ranging between 1.25× to 2.5× (run-time reduction by 20% to 60%) compared to a homogeneous multi-core with small cores. This is to be expected given the relative performance difference between big versus small cores.

The more interesting result is that a heterogeneous multi-core achieves no speedup over a homogeneous multi-core with all small cores for most of the benchmarks. The reason is that a thread that is pinned onto a small

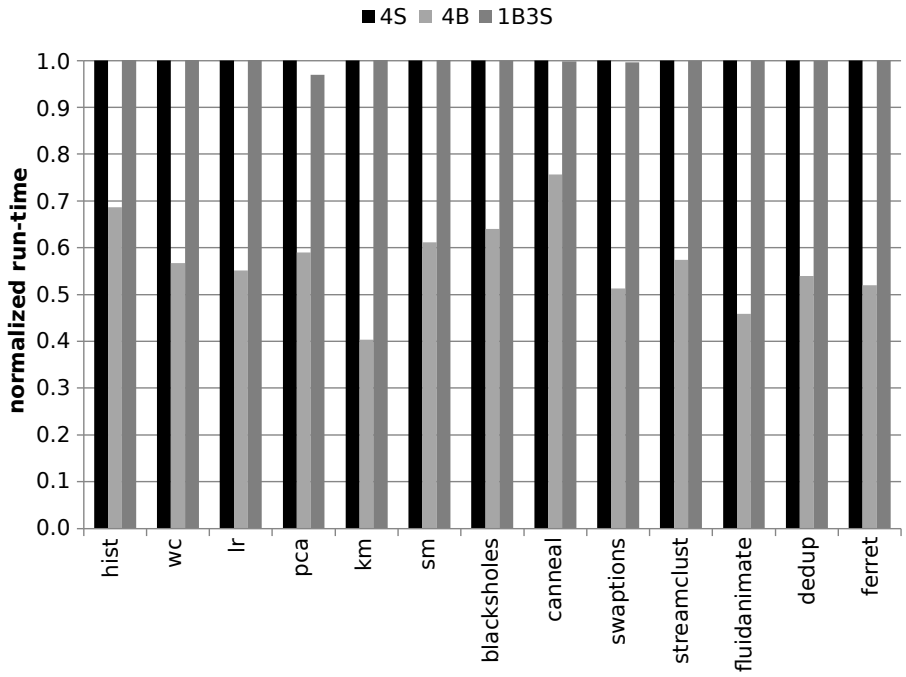


Figure 5.1: Normalized run-time on a homogeneous multi-core with 4 small cores (4S), 4 big cores (4B), and a heterogeneous multi-core with one big and three small cores (1B3S) while keeping threads pinned to cores.

core determines overall application performance, i.e., all threads have to wait for the threads running on the small cores because of synchronization (i.e., barriers). Or, in other words, the thread that runs on the big core makes faster progress compared to the threads running on the small cores, yet it does not contribute to performance. As we will later see in this chapter, by guaranteeing fairness, or making sure all threads get an equal amount of work done or time spent on the big core, all threads will reach the barriers at the same time, improving overall performance.

5.2.3 Multi-program workloads

Figure 5.2 quantifies fairness for a heterogeneous multi-core with one big and one small core (1B1S) for both pinned scheduling and throughput-optimized scheduling while running multi-program workloads composed of random mixes of SPEC CPU2006 benchmarks. Pinned scheduling leads to some programs to make poor progress, i.e., the program that runs on the small core makes less progress than the program that runs on the big

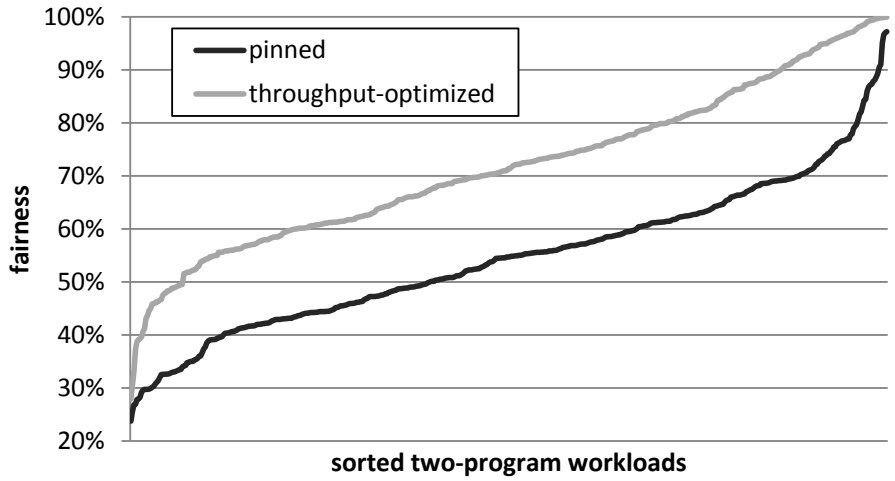


Figure 5.2: Fairness for a 1B1S system for pinned versus throughput-optimized scheduling using PIE for 500 randomly chosen two-job mixes.

core. Overall fairness through pinned scheduling equals 55% on average and is as low as 24% for some workload mixes (see bottom left of the curve in Figure 5.2). State-of-the-art throughput-optimized scheduling using PIE [Van Craeynest et al. 2012], see also previous chapter, not only improves system throughput by 26.6% on average, it also improves fairness by a significant margin from 55% to 72% on average. The reason is that throughput-optimized scheduling reschedules programs during run-time in order to improve throughput, and by dynamically migrating programs between big and small cores in response to time-varying execution behavior, it also improves fairness. Yet, fairness is fairly low (72% on average), and some workload mixes experience significant levels of fairness equal to 27% (see bottom left in Figure 5.2). In other words, both pinned scheduling and throughput-optimized scheduling are largely unfair which may compromise quality-of-service.

5.3 Fairness-Aware Scheduling

Having motivated the importance of fairness as an optimization criterion on single-ISA heterogeneous multi-cores, we now propose fairness-aware scheduling. The key idea of fairness-aware scheduling is to make sure all threads get to run on both the big and small cores for an equal share. We consider two different ways for guaranteeing equal shares, which we describe next.

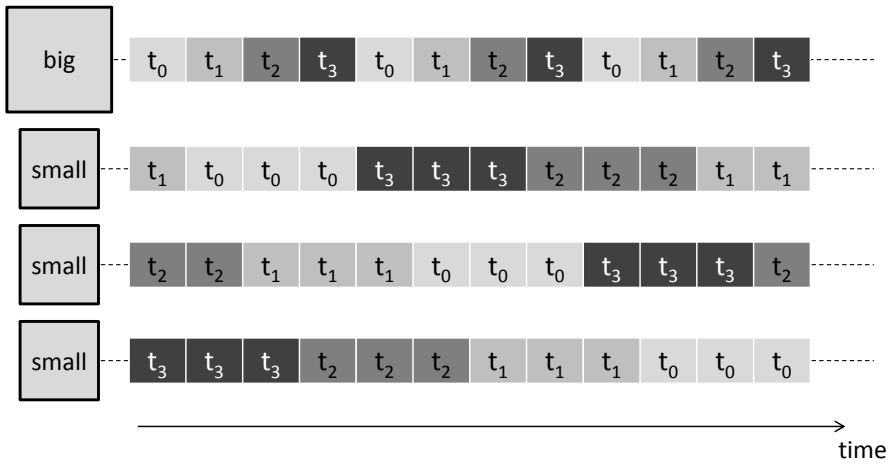


Figure 5.3: Equal-time scheduling on a 1B3S heterogeneous multi-core.

5.3.1 Equal-time scheduling

Equal-time scheduling strives at achieving fairness by running each thread on each core type for an equal amount of time. This is done by keeping track of how often (for how many time slices) a thread has run on all core types, and reschedule if necessary to make sure all threads have run on either core type for an equal number of time slices. Figure 5.3 illustrates this for a heterogeneous multi-core with one big and three small cores (1B3S): each thread gets to run on the big core for one-fourth of the time. Round-robin or random selection of a thread that runs on a small core to next run on the big core is an implementation of equal-time scheduling. Note we do not migrate threads among cores of the same type in order to preserve data locality.

A pitfall with equal-time scheduling is that spending equal time on either core type does not necessarily imply fairness. Some threads experience a larger slowdown from running on a small core than others — these threads get proportionally less work done when scheduled on a small core. Hence, although all threads spend equal time on either core type, threads that experience higher slowdowns on the small cores, will make proportionally less progress. This leads to an unfair system. We therefore propose equal-progress scheduling in the next section which strives at getting equal work done on either core type, and by consequence achieve equal progress.

Note that when all threads exhibit the same (or similar) execution behavior — a so-called homogeneous workload — equal-progress scheduling is in fact identical to equal-time scheduling. Because of the one-to-one rela-

tionship between time and work done, i.e., equal time leads to equal work, scheduling all threads on either core type for equal amounts of time leads to equal amounts of work done on either core type. This is not the case for heterogeneous workloads in which threads execute different codes (and are therefore heterogeneous by design), as we will demonstrate later in this chapter. Similarly, homogeneous-by-design workloads for which different threads end up processing different parts of the input data may exhibit heterogeneous behavior, and may therefore benefit from equal-progress scheduling over equal-time scheduling.

5.3.2 Equal-progress scheduling

Equal-progress scheduling strives at getting all threads to make equal progress. Or, in other words, it strives at making sure all threads experience equal slowdown, per the definition of fairness (Equation 5.2). Equal-progress scheduling continuously monitors fairness and dynamically adjusts the scheduling to achieve fairness. This involves computing the slowdowns for all threads and scheduling the thread with the currently highest slowdown on the big core. (If there are multiple big cores in the system, the threads with the top-n highest slowdowns are scheduled on a big core.) Scheduling the thread with the currently highest slowdown on a big core will reduce its slowdown compared to the other threads (which are scheduled on a small core). As a result, the threads' slowdowns will converge and fairness is achieved.

Computing slowdowns for all threads is where the key challenge lies for equal-progress scheduling. In order to compute a thread's slowdown, we need to know the total execution time on the heterogeneous multi-core as well as the total execution time if we were to execute the thread on a big core in isolation, see Equation 5.1. The former, total execution time on the heterogeneous multi-core, is readily available by counting the number of time slices TS_i the thread has been running so far (on both core types). The latter, total execution time on the big core in isolation, is not readily available and needs to be estimated during run-time. We estimate the isolated, big core execution time by counting the number of time slices the thread was run on the big versus small cores, and by rescaling the time run on the small cores with an estimated big-versus-small-core scaling factor R. A thread's slowdown is then computed as follows:

$$S_i = \frac{T_{het,i}}{T_{big,i}} = \frac{TS_{big,i} + TS_{small,i}}{TS_{big,i} + TS_{small,i}/R_i}. \tag{5.3}$$

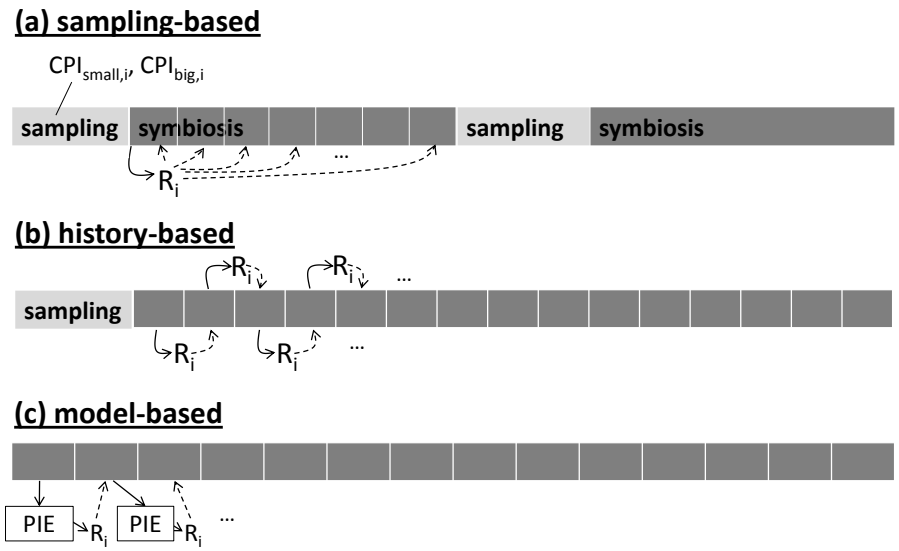


Figure 5.4: Equal-progress scheduling: sampling-based, history-based and model-based.

This formula can be trivially extended to heterogeneous multi-cores with more than two core types.

We explore three methods for estimating the big-versus-small-core scaling factor R, see also Figure 5.4.

- *Sampling-based* scheduling considers a sampling and symbiosis phase. During the sampling phase, the scheduler maps each thread at least once on each core type, and computes the scaling factor R as the ratio between the CPI on the small versus big core:

  $$R = CPI_{small}/CPI_{big}.$$

  The scheduler then uses the computed R factor during the symbiosis phase. We assume the symbiosis phase is ten times longer than the sampling phase in our setup.

- *History-based* scheduling computes the CPI seen on both small and big cores and uses the ratio for estimating slowdown. The benefit over sampling-based scheduling is that history-based scheduling continuously adjusts the computed big-to-small ratio based on the most recent CPI values. In order for history-based scheduling to work in practice, it needs a bootstrap phase in which all threads get to run on all core types at least once.

- *Model-based* scheduling continuously monitors CPI on either core type and estimates the big-to-small-core ratio using an analytical model. The key benefit is that model-based scheduling continuously updates the big-to-small-core ratio based on the most recent time slice. Sampling- and history-based scheduling on the other hand use stale CPI values to compute the ratio. We use the PIE model [Van Craeynest et al. 2012] for estimating the big-to-small-core ratio. The effectiveness of the model-based approach depends on the accuracy of the model, and may require hardware support for computing the inputs for the model (as is required for PIE). The sampling-based and history-based methods on the other hand do not require hardware support, and they use real performance measurements, which may be more accurate than model estimates, albeit being stale.

5.3.3 Trading fairness for throughput

So far, we considered fairness as the only optimization criterion, i.e., the proposed fairness-aware scheduling mechanisms strive at achieving fairness and is oblivious to system throughput. However, in some practical use cases, fairness is not the only optimization criterion, and system throughput is at least equally important. For example, in a batch-style, throughput-oriented system, maximizing system throughput might be of primary importance, and fairness among users or jobs might be a secondary concern. Hence, it might make sense to provide a flexible scheduling algorithm that enables trading off fairness for throughput, and vice versa.

We therefore propose a scheduling approach that trades off fairness and system throughput: *Guaranteed-fairness* scheduling optimizes for system throughput, yet when fairness drops below a given threshold $\theta_{fairness}$, scheduling defers to optimizing fairness until fairness reaches at least the threshold, after which it defers again to throughput-optimized scheduling. Guaranteed-fairness scheduling thus needs to continuously monitor slowdowns and estimate fairness, as done for fairness-aware scheduling. We consider different fairness thresholds for guaranteed-fairness scheduling in the evaluation section.

5.3.4 Rescheduling granularity

The fairness-aware scheduling algorithms proposed dynamically reschedules threads across core types during run-time. This is done at the gran-

ularity of a time slice. There are a number of factors that affect a good choice of time slice granularity. A small time slice potentially makes the system more reactive, i.e., the scheduling algorithm can guarantee fairness at smaller time scales and more quickly react to time-varying execution behavior. On the other hand, a small time slice also incurs more migration overhead when threads are more frequently rescheduled. The migration overhead not only includes overhead due to a context switch, it also incurs overhead for warming hardware state, especially in the memory hierarchy. Whereas context switch overhead incurs a fixed cost for restoring architecture state, the overhead for warming hardware state depends on the workload and its working set size, as well as the memory hierarchy.

In the previous chapter, we did an extensive evaluation to quantify migration overhead for both shared and private last-level caches (LLCs) as a function of time slice granularity. We found the migration overhead to be less than 1.5% across all workloads for a 4 MB shared LLC for a 1 ms time slice, and less than 0.6% for a 2.5 ms time slice. They also explored migration overhead for private LLCs and found the overhead to be small as well (although slightly higher compared to shared caches) because the cache coherency protocol can get the data from another core's private LLC instead of memory. Our own experimental evaluation confirms these findings, and we consider a 1 ms time slice unless mentioned otherwise.

5.3.5 Hardware versus software scheduling

Implementing fairness-aware scheduling can be done both in hardware and in software. When implemented in hardware, the fairness-aware scheduling would need a small timeslice, e.g., 1 ms, while system software (the OS or VMM) uses a larger timeslice, e.g., 4+ ms. By doing so, the hardware would be able to provide the abstraction to software of homogeneous hardware, while dynamically rescheduling threads among the cores in a heterogeneous multi-core within an OS time slice. For example, a heterogeneous multi-core with one big and three small cores, may then be exposed to software as a homogeneous multi-core with four cores and a 4 ms time slice; the hardware however, would then dynamically reschedule threads among the cores at a 1 ms time slice. By scheduling for fairness, hardware would expose itself as a homogeneous multi-core in which all threads make equal progress. System software does not need to be changed and is oblivious to hardware heterogeneity. In contrast, implementing fairness-aware scheduling in software requires modifications to the OS or VMM

to keep track of each thread's progress, and enables guaranteeing fairness at a larger time scale. Fairness-aware scheduling can be implemented in both software and hardware, and works at different time scales, as we will demonstrate later.

5.4 Experimental Setup

Before describing and analyzing results, we first describe our experimental setup.

5.4.1 Simulated architectures

We use Sniper [Carlson et al. 2011] for conducting the simulation experiments in this chapter. Sniper is a parallel, hardware-validated, x86-64 multi-core simulator capable of running both multi-program and multi-threaded applications. We configure Sniper to model heterogeneous multi-core processors with big and small cores. The big core is a 4-wide out-of-order processor core; the small core is a 4-wide (stall-on-use) in-order processor core. We assume both cores run at a 2.6 GHz clock frequency. Further, we assume a cache hierarchy with separate 32 KB L1 instruction and data caches, and a 256 KB L2 cache; we assume the L1 and L2 caches to be private per core. The L3 last-level cache (LLC) is shared among all cores, for a total size of 16 MB. We consider the LRU replacement policy in all of the caches.

As mentioned before, the time slice granularity is set to be 1 ms, in order to be able to exploit time-varying workload execution behavior while keeping migration overhead small. The overhead for migrating a workload from one core to another (storing and restoring the architecture state) is set to 1,000 cycles, plus the time it takes to drain a core's pipeline prior to migration. Finally, we do account for the migration overhead due to cache effects.

5.4.2 Workloads

We consider both multi-program and multi-threaded workloads in our experiments. The multi-program workloads are composed out of SPEC CPU2006 benchmarks; there are 26 benchmarks in total, which along with all of their reference inputs leads to 55 benchmarks in total. We select representative simulation points of 750 million instructions each; these

Suite	*Benchmark*	*Input*
PARSEC	blackscholes	simmedium
	canneal	simmedium
	swaptions	simmedium
	streamcluster	simmedium
	fluidanimate	simmedium
	dedup	simmedium
	ferret	simmedium
MapReduce	histogram	1.5 GB image
	word count	100 MB file
	linear regression	100 MB file
	PCA	1024 x 1024 matrix
	K-means	64 clusters, 65536 points
		256-dimension vectors
	string match	100 MB file

Table 5.1: Multi-threaded benchmarks used in this study.

simulation points were selected using PinPoints [Patil et al. 2004]. When running multi-program workloads, we stop the simulation as soon as the first benchmark in the workload mix reaches the end of its simulation point; this corresponds to hundreds of time slices. We quantify system throughput using the STP metric [Eyerman and Eeckhout 2008] (also called weighted speedup [Snavely and Tullsen 2000]) which quantifies the aggregate throughput achieved by all cores in the system. We use Equation 5.2 when reporting fairness. We consider 500 randomly chosen two-job workload mixes, and 200 randomly chosen four-job and eight-job mixes.

The multi-threaded benchmarks that we use in this study are selected from Phoenix [Ranger et al. 2007] and PARSEC [Bienia et al. 2008], see Table 5.1. The Phoenix benchmarks are MapReduce workloads with Map, Reduce and Merge phases, using the Metis [Mao et al. 2010] library for shared-memory multi-core processors. These workloads are homogeneous (i.e., all threads run the same code) and barrier-synchronized between parallel phases. Most of the PARSEC benchmarks are homogeneous and barrier-synchronized as well, expect for `dedup` and `ferret` which are pipelined programs. The latter two benchmarks are therefore heterogeneous, i.e., different threads execute different codes and communicate through a producer-consumer relationship. We use the `simmedium` inputs for PARSEC. The run-times for the multi-threaded benchmarks are such that we simulate several hundreds upto a couple thousands of time slices. We run the benchmarks to completion and measure total run-times.

5.5 Evaluation

We now evaluate fairness-aware scheduling and compare against two alternative scheduling policies, namely throughput-optimized and pinned scheduling. Throughput-optimized scheduling is state-of-art dynamic PIE scheduling [Van Craeynest et al. 2012] which uses a simple analytical model to predict on which core type to map which program or thread in order to optimize system throughput. PIE dynamically reschedules threads to exploit time-varying execution behavior, see also the previous chapter. Pinned scheduling is our baseline, and reflects current practice in contemporary operating system schedulers, as done in the Linux 2.6 kernel [Jones 2006]. Pinned scheduling maps threads to cores and keeps threads pinned to cores in order to improve data locality and affinity. When reporting performance numbers for pinned scheduling, we consider multiple random mappings of threads to cores and report average performance across those random mappings. We believe pinned scheduling is a reasonable baseline to compare against. In case fairness-aware scheduling were implemented in hardware, pinned scheduling reflects system software (randomly) mapping and pinning threads to virtual cores, while hardware scheduling reschedules threads to physical cores to optimize fairness. In case fairness-aware scheduling were implemented in software, pinned scheduling reflects optimizing for data locality.

5.5.1 Multi-program workloads

We first evaluate fairness-aware scheduling in the context of multi-program workloads.

Equal-time versus equal-progress Scheduling

Figure 5.5 reports throughput and fairness for both equal-time and equal-progress fairness-aware scheduling, compared to pinned scheduling and throughput-optimized scheduling, for a 1B1S heterogeneous multi-core with one big and one small core. We consider history-based equal-progress scheduling here, and evaluate other equal-progress policies later. The workloads on the horizontal axis are sorted. Pinned scheduling performs the worst in terms of fairness, with an average fairness of 55%. Throughput-optimized scheduling improves fairness somewhat to 72% on average. By dynamically rescheduling threads among cores, throughput-optimized scheduling not only improves throughput, but also improves

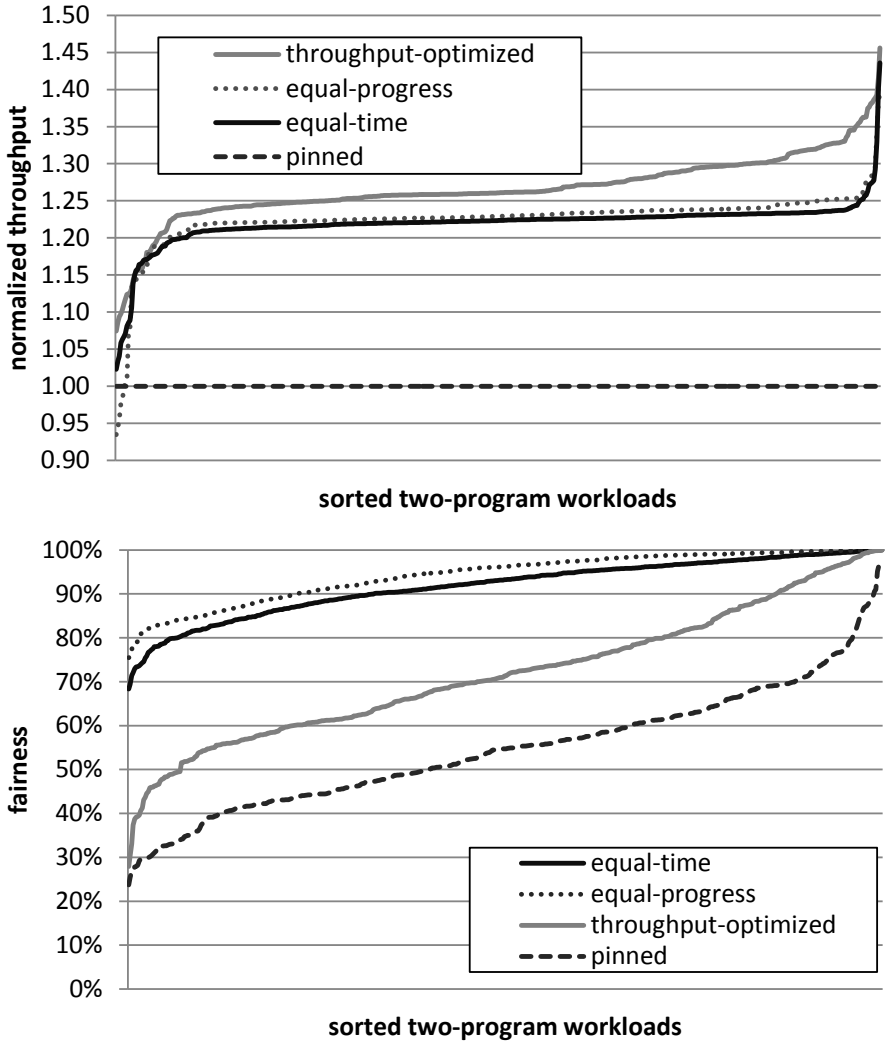


Figure 5.5: Comparing scheduling algorithms relative to pinned scheduling in terms of throughput (top graph) and fairness (bottom graph) for a 1B1S heterogeneous multi-core with one big and one small core.

fairness; a thread may get to run on either core type for at some fraction of time. Fairness-aware scheduling achieves the highest fairness (92% on average), and equal-progress scheduling slightly outperforms equal-time scheduling. The highest unfairness observed across all the 500 job mixes is no higher than 38%, which is a substantial improvement over both pinned and throughput-optimized scheduling with unfairness numbers up to 73%. Fairness-aware scheduling results in a slightly lower system throughput compared to throughput-optimized scheduling: 22.0% and 21.9% for equal-progress and equal-time scheduling versus 26.6% for throughput-optimized scheduling. The reason why equal-progress scheduling outperforms equal-time scheduling in terms of throughput is that it takes into account the amount of work done on either core type, and not just time.

Scalability

We now evaluate fairness-aware scheduling as a function of the number of cores and different ratios of big to small cores. Figure 5.6 shows average fairness as well as throughput values for 1B1S, 1B3S, 3B1S, 1B7S and 7B1S systems. The overall conclusion is that fairness-aware scheduling achieves the highest fairness across the board, with average fairness values ranging between 79% and 92%, which is significantly higher compared to pinned and throughput-optimized scheduling with average fairness values around 50 and 70%, respectively. Equal-progress scheduling achieves higher fairness compared to equal-time scheduling for the 1B3S and 1B7S systems, but achieves similar average levels of fairness for the other systems. The reason is that equal-progress scheduling computes slowdowns based on actual progress — not time — which is more accurate and turns out to be more critical when the number of big cores is small compared to the number of small cores. In other words, as the relative number of big cores decreases, the big core becomes a bottleneck and making accurate slowdown predictions becomes more critical towards optimizing fairness.

Note also that fairness degrades with decreasing relative fractions of big cores, even under fairness-aware scheduling. Fairness degrades from 92% for a 1B1S system (1/2 the cores are big cores), to 79% for a 1B7S system (1/8th the cores are big cores). This can be understood intuitively because the big core is increasingly becoming a bottleneck as the relative number of big cores decreases in the system. In other words, fairness is easier to be achieved when the number of big versus small cores is more balanced.

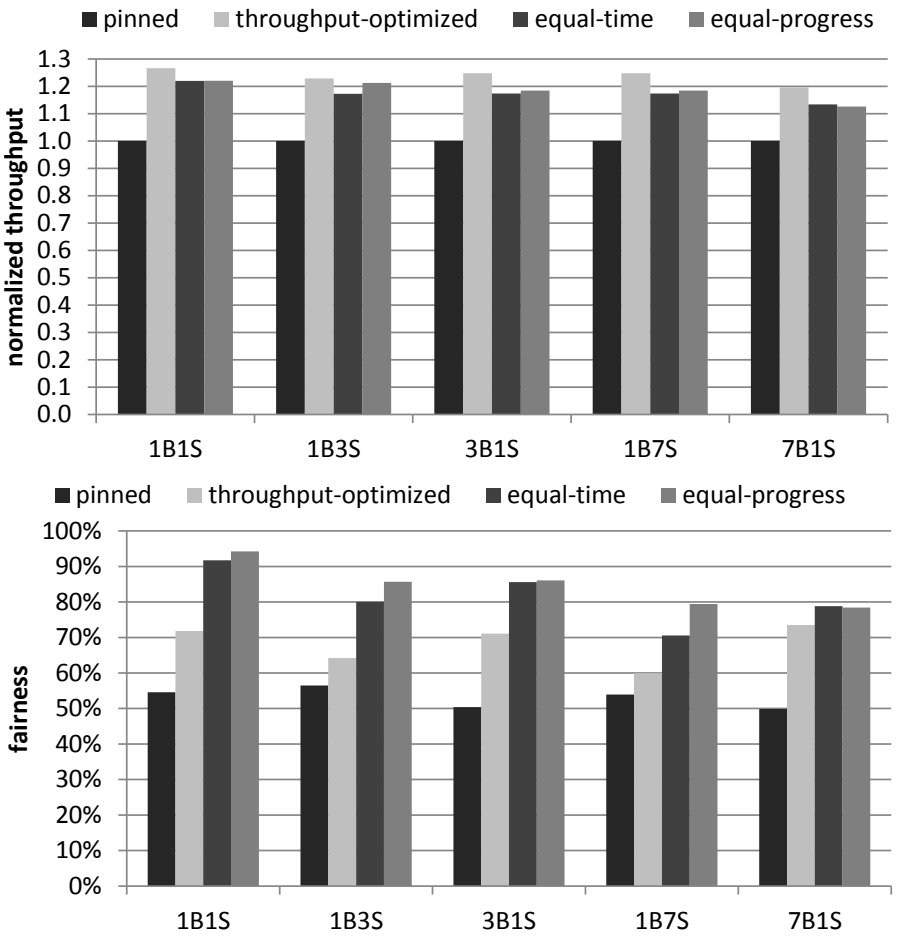


Figure 5.6: Fairness-aware scheduling as a function of core count in terms of throughput (top graph) and fairness (bottom graph).

Finally, note that equal-progress scheduling systematically outperforms equal-time scheduling in terms of throughput, albeit by a small fraction of around 2%. The reason again is that equal-progress scheduling takes into account actual work done on each core type as opposed to assuming time corresponds to work as done by equal-time scheduling. It is interesting to note though that fairness-aware scheduling improves system throughput over pinned scheduling by a significant margin across the board, ranging from 23.7% for a 1B1S system to 14.8% for a 7B1S system. In other words, system throughput also improves while optimizing for fairness. The reason is that optimizing for fairness involves that all threads get to run on the big core(s) for some fraction of the time, and by doing so, overall system performance gets improved. Fairness-aware scheduling in fact, is quite effective

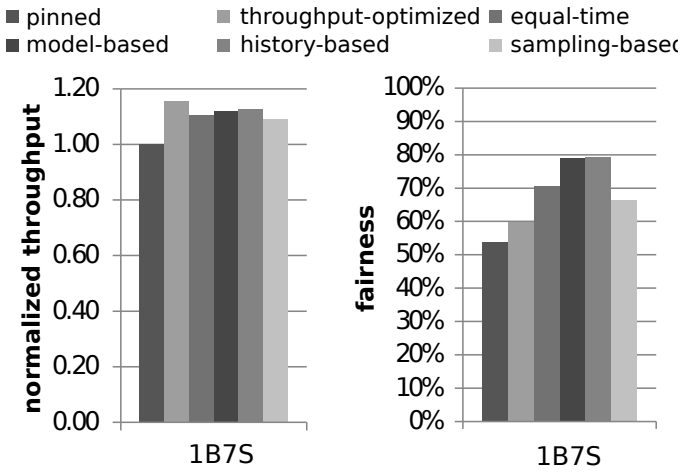


Figure 5.7: Evaluating different methods for estimating the big-to-small-core scaling factor in equal-progress scheduling for a 1B7S system.

at achieving high system throughput. Compared to throughput-optimized scheduling using PIE (current state-of-the-art), fairness-aware scheduling incurs a small average degradation in system throughput ranging between 4.5% (for 1B1S), 3.6% (for 1B3S) and 7.0% (for 7B1S).

Equal-progress scheduling

As mentioned earlier, there are a number of ways for estimating the big-to-small-core scaling ratio in equal-progress scheduling. We considered the history-based method so far; we now evaluate the other two, sampling- and model-based, methods. Figure 5.7 compares these three methods in terms of throughput and fairness for a 1B7S system. Sampling-based scheduling performs worst (both in terms of fairness and throughput) because it periodically estimates the big-to-small-core scaling ratio, after which the scaling ratio is used during the symbiosis phase. Sampling incurs overhead and is unable to quickly adapt to time-varying workload behavior. Note sampling-based scheduling performs even worse compared to equal-time scheduling. The history-based and model-based methods perform much better, and both outperform sampling-based scheduling and equal-time scheduling, as they continuously update the big-to-small-core scaling ratio.

We find history-based scheduling to typically outperform model-based scheduling, albeit by a small margin. As mentioned before, model-based

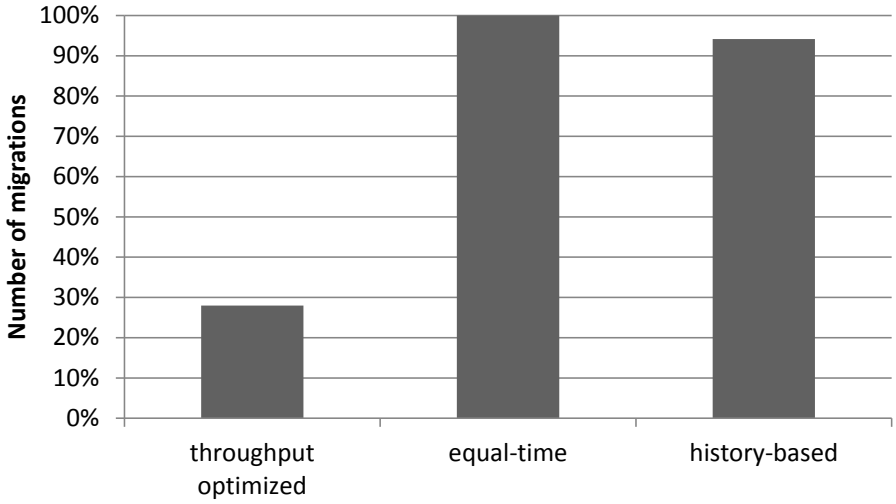


Figure 5.8: The number of migrations across core types in a heterogeneous multi-core system under various scheduling policies.

scheduling does not rely on stale data to compute the big-to-small-core ratio but is limited by the accuracy of the underlying model; history-based scheduling on the other hand computes the big-to-small ratio based on real hardware measurements instead of a model, which might provide more accurate big-to-small-core scaling ratios in case the hardware measurements are fairly recent. Now, the reason why history-based scheduling outperforms the model-based approach is that optimizing for fairness enforces threads to migrate across core types, which enables the history-based approach to continuously update the big-to-small-core ratio using fairly recent performance numbers on both the small and big cores. This is further supported by the data presented in Figure 5.8 which shows the normalized number of migrations across core types. Pinned scheduling, by definition, does not incur any migrations, whereas equal-time scheduling incurs the largest number of migrations. Equal-progress scheduling incurs slightly fewer migrations; history-based scheduling is thus able to continuously update the big-to-small-core ratio which is more accurate that the predicted ratio using the model in model-based scheduling.

Trading fairness versus throughput

As mentioned earlier, optimizing system performance is often a complex trade-off in terms of system throughput (i.e., getting as much jobs done per unit of time) versus user-level experience (i.e., all users should be treated

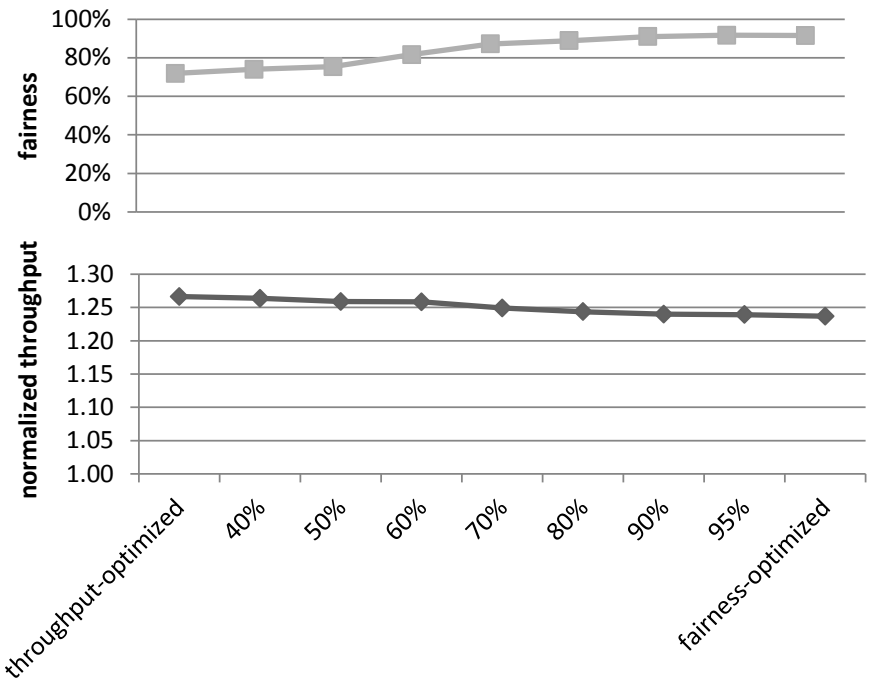


Figure 5.9: Trade-off between fairness and throughput-optimized scheduling for a 1B1S system.

in a fair way). The throughput-optimized and fairness-aware scheduling policies optimize system throughput and fairness, respectively, and completely oblivious to the other optimization criterion. We now evaluate guaranteed-fairness scheduling which optimizes system throughput unless fairness drops below a given threshold, after which it defers to fairness-aware scheduling; once fairness is above the threshold, it optimizes system throughput again. Figure 5.9 evaluates guaranteed-fairness scheduling in terms of system throughput and fairness. This graph illustrates that turning the threshold 'knob' enables trading off fairness for throughput and vice versa. Setting the threshold to a higher value leads to high fairness, alike fairness-aware scheduling. Setting the threshold to a lower value leads to high levels of system throughput, alike throughput-optimized scheduling.

Time slice granularity

So far, we assumed a 1 ms time slice. Figure 5.10 evaluates fairness-aware scheduling across different time slices, including 1, 5 and 10 ms, for a

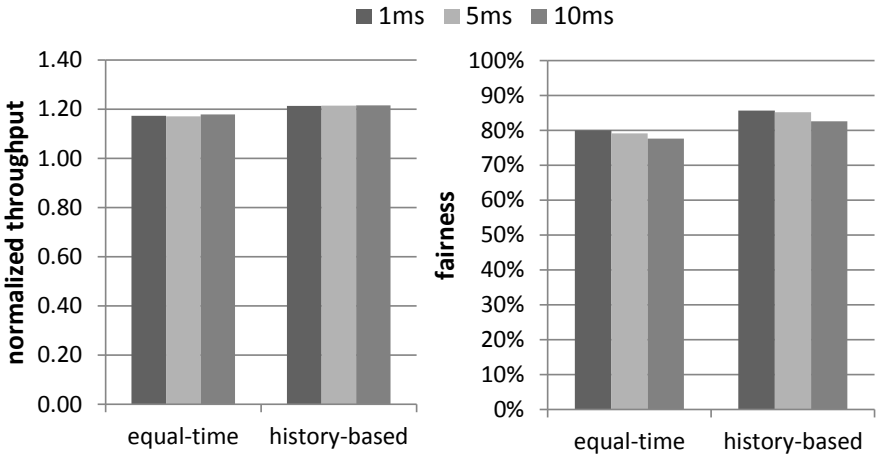


Figure 5.10: System throughput and fairness for equal-time and equal-progress (history-based) scheduling as a function of time slice granularity.

1B3S system. The motivation for exploring larger time slices is to evaluate fairness-aware scheduling in system software; smaller time slices correspond to implementing fairness-aware scheduling in hardware so that system software is oblivious to hardware heterogeneity, as previously discussed. We conclude from Figure 5.10 that both equal-time and equal-progress scheduling are (largely) insensitive to time slice granularity, i.e., similarly high levels of system throughput and fairness are achieved across different time slices. Fairness only slightly decreases with increasing time slices, the reason being that fairness converges slower with larger time slices; given the fixed workload (and run-time) this leads to slightly lower fairness values.

5.5.2 Multi-threaded workloads

We now evaluate fairness-aware scheduling for multi-threaded applications. We first consider homogeneous workloads in which all threads execute the same code, followed by heterogeneous workloads; these types of workloads lead to different execution behavior and results.

Homogeneous workloads

We first consider all the MapReduce workloads as well as the homogeneous workloads from the PARSEC benchmark suite. Figure 5.11 compares

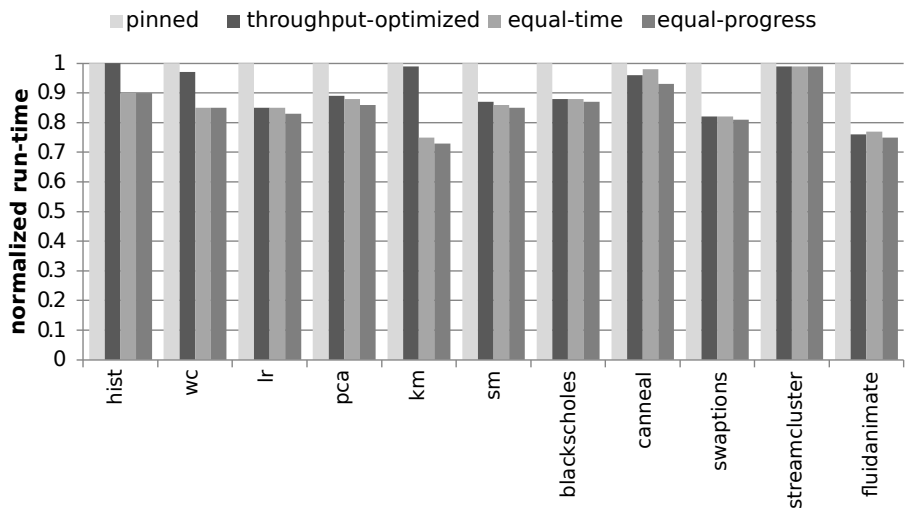


Figure 5.11: Comparing scheduling algorithms relative to pinned scheduling for a 1B3S system running homogeneous multi-threaded applications.

the various scheduling policies for a 1B3S heterogeneous multi-core system in terms of execution time normalized to pinned scheduling. The key result from this graph is that fairness-aware scheduling improves execution time by 14% on average and up to 25% over pinned scheduling. Interestingly, equal-time and equal-progress scheduling perform equally well. The intuitive understanding is that these workloads are homogeneous (all threads execute the same code and exhibit the same execution behavior), and enforcing equal time therefore leads to enforcing equal progress as well. Fairness-aware scheduling forces all threads to make equal progress by running on the big core for an equal share. This eventually leads to all threads reaching the barriers at roughly the same time. Under pinned scheduling on the other hand, the one thread that gets scheduled onto the big core reaches the barrier before the other threads; because this thread has to wait for the other threads on the small cores to reach the barrier, scheduling one of the thread on the big core does not contribute to overall performance, yielding no benefit from heterogeneity. By making sure all threads benefit from the big core, fairness-aware scheduling forces all threads to make equal progress, thereby reaching the barrier at the same time and improving overall performance.

Throughput-optimized scheduling improves performance for most benchmarks but not all, leading to an average improvement of 10% on average. The reason is that throughput-optimized scheduling improves fairness, as we have seen for the multi-program workloads, by migrating

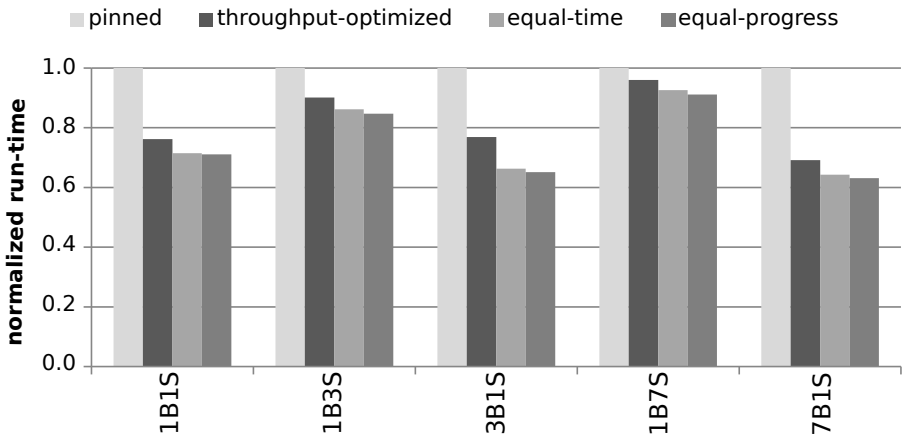


Figure 5.12: Fairness-aware scheduling for different heterogenous multi-core configurations for the homogeneous multi-threaded applications.

threads across core types while optimizing system throughput. However, the fact that fairness improves is a side-effect from optimizing for throughput; fairness-aware scheduling which specifically optimizes fairness achieves higher levels of fairness, which eventually leads to significantly better performance for multi-threaded workloads.

Figure 5.12 reports average results across different heterogeneous multi-core configurations for the homogeneous multi-threaded workloads. The conclusion is essentially the same as what we reported earlier for the individual benchmarks. Fairness-aware scheduling improves performance substantially over pinned scheduling: we report average performance improvements ranging between 7.5% (for 1B7S) and 35% (for 7B1S). Further, equal-time and equal-progress scheduling perform equally well (the benefit from equal-progress scheduling over equal-time scheduling is marginal). Interestingly, performance improves with larger fractions of big cores in the system (compare 3B1S versus 1B3S, and 7B1S versus 1B7S) under fairness-aware scheduling. (In contrast, performance does not improve under pinned scheduling because the application has to wait for the slowest thread running on a small core anyways.) The reason is that fairness-aware scheduling gets to distribute and map threads across small and big cores, and when there are more big cores in the system, the average performance seen by all threads will be higher with more big cores in the system, leading to better overall performance.

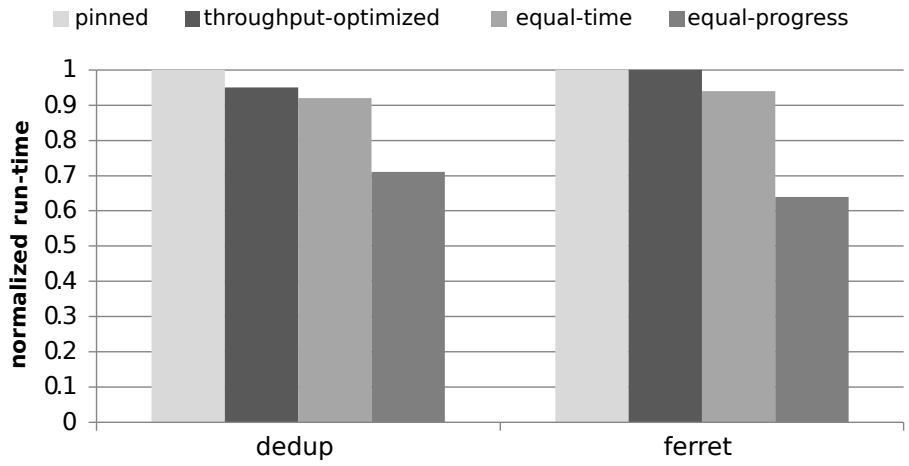


Figure 5.13: Comparing scheduling algorithms relative to pinned scheduling for a 1B3S system running heterogeneous multi-threaded applications.

Heterogeneous workloads

Figure 5.13 reports normalized execution time for the various scheduling policies for the two heterogeneous PARSEC benchmarks, `dedup` and `ferret`. The key conclusion from this graph is that although equal-time scheduling improves performance somewhat over pinned scheduling (8% for `dedup` and 6% for `ferret`), equal-progress scheduling improves performance by as much as 29% for `dedup` and 36% for `ferret`. The reason is that these workloads are heterogeneous, and, as a result, equal time does not necessarily correspond to equal progress. As the different threads execute different code and exhibit different execution behavior, they experience different big-to-small-core performance ratios and hence accounting for the different ratios is important for achieving fairness. Equal-progress scheduling does account for the fact that different threads make different progress and schedules threads such as to improve fairness, which ultimately leads to better overall performance.

5.6 Related Work

Rangan et al. [2011] explore throughput-optimizing and fairness-aware scheduling algorithms for homogeneous multi-cores for which each core runs at a different clock frequency due to within-die process variations. The proposed scheduling algorithms do not readily apply to hetergeneous

multi-cores with core microarchitecture diversity; in addition, they consider multi-program workloads only. Michaud et al. [2007] propose a scheduling algorithm for temperature-constrained multi-cores in which threads migrate between hot and cold cores in order to avoid hotspots while achieving fairness among threads. Fedorova et al. [2007] propose a scheduling approach for multi-cores (with different cores running at different clock frequencies) that ensures that each thread's execution time is balanced across all cores; the key difference with the equal-time scheduling approach proposed in this paper is that we balance time across core types (not cores), and by doing so we avoid unnecessary migrating among cores of the same type.

A large body of recent work has looked into multi-core and multi-threading fairness issues. Most of this prior work focused on fairness issues in shared caches [Iyer 2004, Jaleel et al. 2008b, Kim et al. 2004, Nesbit et al. 2007, Qureshi and Patt 2006, Luque et al. 2009, Guo et al. 2007, Zhou et al. 2009], main memory [Mutlu and Moscibroda 2007], or both shared cache and main memory [Iyer et al. 2007, Ebrahimi et al. 2010; 2011]. Several studies looked into fairness issues in simultaneous multithreading (SMT) processors [Cazorla et al. 2004, Eyerman and Eeckhout 2009, Snavely et al. 2002] and coarse-grained multi-threading processors [Gabor et al. 2007]. None of this prior work looked into fairness issues that arise from core heterogeneity.

5.7 Summary

Current multi-core schedulers in modern operating systems affinitize or pin threads to cores, which leads to unfair performance on heterogeneous multi-cores. For multi-threaded workloads, unfair performance leads to thread(s) running on a big core to wait at barriers for the other threads running on the small cores, yielding no performance benefit from heterogeneity. For multi-program workloads, unfair performance may compromise quality-of-service because of large variability in performance across simultaneously running programs. Optimizing for system throughput, as proposed in a significant body of recent work, improves fairness somewhat by dynamically scheduling threads across core types during run-time while optimizing for throughput in response to time-varying workload behavior, yet, fairness is still poor.

We have proposed fairness-aware scheduling which optimizes for fairness as its primary optimization objective. We described two techniques

for making sure all threads get to run on either core types for equal shares. Equal-time scheduling schedules threads such that they all spend equal amounts of time on either core type. Equal-progress scheduling strives at getting all threads to make equal progress, and we described three methods for dynamically estimating a thread's progress. We further explored a scheduling mechanism that trades off fairness for throughput, and described ways for implementing fairness-aware scheduling at different time scales and both in software and hardware.

Our experimental results demonstrate the significance of fairness-aware scheduling for heterogeneous multi-cores. We report substantial improvements in fairness over pinned scheduling (current multi-core schedulers) and throughput-optimized scheduling (current state-of-the-art in heterogeneous multi-core scheduling for system throughput), achieving average fairness levels of 86% for a 1B3S system running multi-program workloads. Fairness-aware scheduling also improves system throughput by 21.2% over pinned scheduling, while being within 3.6% compared to throughput-optimized scheduling. For homogeneous multi-threaded workloads, fairness-aware scheduling improves performance by 14% on average and up to 25%, and equal-progress and equal-time scheduling perform equally well. For heterogeneous multi-threaded workloads, equal-progress scheduling significantly outperforms equal-time scheduling, leading to an overall performance improvement of 32% on average over pinned scheduling.

The overall conclusion is that fairness-aware scheduling is key to optimizing performance on single-ISA heterogeneous multi-cores for both multi-threaded and multi-program workloads.

Chapter 6

Future Work

Veni, vidi, bici.

6.1 Summary

We presented the Multi-Program Performance Model (MPPM), a method for quickly estimating multi-program multi-core performance based on single-core simulation runs. Because MPPM involves analytical modeling, it is very fast, and it estimates multi-core performance for a very large number of multi-program workloads in a reasonable amount of time with good accuracy (in addition, it provides confidence bounds on its performance estimates). We report an average performance prediction error of 2.3% and 2.9% for system throughput (STP) and average normalized turnaround time (ANTT), respectively, while being up to five orders of magnitude faster than detailed simulation. Subsequently, we demonstrate that randomly picking a limited number of multi-program workloads, as done in current practice, can lead to incorrect design decisions in practical design and research studies, which is alleviated using MPPM. In addition, MPPM can be used to quickly identify multi-program workloads that stress multi-core performance through excessive conflict behavior in shared caches; these stress workloads can then be used for driving the design process further.

We then used the MPPM model for exploring the heterogeneous multi-core design space. MPPM allows us to do performance predictions for arbitrary compositions of heterogeneous architectures and workloads. Moreover, it allows for quantifying heterogeneous architecture performance for

a large number (hundreds) of possible job mixes in a reasonable amount of time. Using architectural simulation, simulating and exploring a large heterogeneous architecture design space for a very large number of job mixes is impossible in a reasonable amount of time. This analysis leads to several interesting and insightful observations in some of the fundamental trade-offs and design choices.

- Improving system throughput (while assuming a fixed chip area), decreases average per-program performance. Conversely, trading a number of simple in-order cores for an aggressive out-of-order core improves per-program performance, but it also decreases total system throughput.
- Homogeneous architectures cover a broad range of the performance spectrum in terms of throughput versus job turnaround time. Heterogeneity on the other hand allows for designing multi-core processors with more fine-grained trade-offs in system throughput versus job turnaround time.
- Interestingly though, some homogeneous configurations are optimal for particular throughput versus job turnaround time trade-offs.
- We find that two core types offer most of the performance benefits from heterogeneity, i.e., going to a larger number of core types does not contribute much.
- We found that some compositions of core types do not yield Pareto-optimal configurations.
- Limited off-chip bandwidth has a significant impact on the fundamental design choices in heterogeneous architectures. When limiting off-chip bandwidth, increasing system throughput comes at the cost of a proportionally larger degradation in per-program performance.
- We also find that the effectiveness of heterogeneous architectures heavily depends on how jobs are mapped on the different core types i.e., the effectiveness of the scheduler.

In Chapter 4, we have proposed Performance Impact Estimation (PIE) as a mechanism to schedule workloads on a single-ISA heterogeneous multi-cores. PIE collects CPI stack, MLP and ILP profile information at runtime, and estimates the performance impact if the workload were

to run on a different core type. Dynamic PIE scheduling exploits time-varying execution behavior by adjusting the scheduling on a per-interval basis; hardware support for PIE scheduling is limited. We show that PIE scheduling makes accurate scheduling decisions and outperforms state-of-the-art application schedulers for heterogeneous multi-cores. PIE also outperforms sampling-based scheduling because it does not incur sampling overhead as it predicts (rather than samples) the performance impact of a scheduling decision. A key advantage of PIE scheduling is that it is scalability: sampling-based scheduling does not scale with an increasing number of small cores, in contrast to PIE scheduling because the latter estimates (and does not sample) performance. In addition, we demonstrate that high-frequency workload migration can be done with low overhead for both private and shared LLCs, which enables fine-grained scheduling. Finally, we provide the insight that scheduling policies benefit from intelligent shared LLC management.

We evaluate PIE scheduling using a large number of multi-programmed SPEC CPU2006 workload mixes. We considered a set of scheduling-sensitive workload mixes on a heterogeneous multi-core designs consisting of out-of-order and in-order cores. We report an average performance improvement of 5.5% over recent state-of-the-art scheduling proposals. We also evaluate PIE scheduling and demonstrate its scalability across a range of heterogeneous multi-core configurations, including private and shared last-level caches and different cache replacement policies. Finally, we show that PIE outperforms a sampling-based scheduling by an average of 8.7%.

In Chapter 5, we made a case for fairness-optimizing scheduling. A fair heterogeneous system has many desirable properties. For one, it provides to opportunity to hide the heterogeneity from the operating system (or more generally, software). Current software assumes that all threads make equal progress, by guaranteeing this property through fairness-optimizing scheduling, we allow for an easier transition to heterogeneous multi-core processors. Also, for multi-program workloads, fairness is very importance when it comes to system-level priorities and quality-of-service. We described two techniques for making sure all threads get to run on either core types for equal shares. Equal-time scheduling schedules threads such that they all spend equal amounts of time on either core type. Equal-progress scheduling strives at getting all threads to make equal progress, and we described three methods for dynamically estimating a thread's progress. We further explored a scheduling mechanism that trades off fairness for throughput, and described ways for implementing fairness-aware schedul-

ing at different time scales and both in software and hardware. Additionally, we showed that by providing a fair schedule, throughput can be improved as well.

We show substantial improvements in fairness over a pinned scheduling and state-of-the-art throughput-optimized scheduling, achieving average fairness levels of 86% for a 1B3S system running multi-program workloads while improving system throughput by 21.2% over pinned scheduling. For homogeneous multi-threaded workloads, fairness-aware scheduling improves performance by 14% on average and up to 25%, and equal-progress and equal-time scheduling perform equally well. For heterogeneous multi-threaded workloads, equal-progress scheduling significantly outperforms equal-time scheduling, leading to an overall performance improvement of 32% on average over pinned scheduling.

6.2 Future Work

6.2.1 Modeling

There is ample room for further evaluation, improvements and extensions to MPPM. As it is, MPPM uses the frequency of access contention model for shared caches with a least-recently used replacement policy. This is a fairly simple model that partitions the cache proportional to the relative access intensity for all co-running threads. Although we have shown that this model is accurate enough for the studies we have done with MPPM, it can be improved, further increasing the accuracy of the MPPM framework. One way of doing so could be to use the cache access patterns themselves (instead of access intensity) to more accurately assign cache ways to the threads sharing the cache. This can be done by normalizing the stack distance counters and using them as a distribution. Using basic probabilistic theory, we could then estimate the probability that a certain way will be used by a certain thread. The resulting partitioning (and estimated contention for resources) could then be used in the same way as we used the FOA model.

As on-chip caches continue to increase in size, their properties change. One such example is non-uniform cache access (NUCA) caches: the (typically last-level) cache is divided into different cache banks, where the access time to nearby banks is lower than the access time to far away banks. The cache block placement can be static or dynamic [Kim et al. 2002, Jaehyuk Huh et al. 2007, Dybdahl and Stenstrom 2007]. The simplest policy of

uniformly distributing memory addresses across cache banks could be easily integrated into MPPM by incurring different miss penalties for different cache banks. The dynamic schemes cannot be readily integrated, however, additional profiling of bank access patterns could be collected during the profiling step and used to assign miss penalties in MPPM.

Currently, MPPM only supports LRU-managed caches, recent research Jaleel et al. has shown that the commonly used LRU replacement policy is outperformed by contention-aware replacement policies such as RRIP[Jaleel et al. 2010]. Therefore, it is important to extend the MPPM framework could be extended to support replacement policies other than LRU. For this, a performance model of the replacement policy is needed and to our knowledge, there exists no analytical models to accurately estimate the cache performance of these novel replacement policies. We believe that creating a model of contention-aware policies could be more straightforward than for traditional contention-agnostic. The reason is twofold: firstly, there is less contention and hence accurate modeling is much less critical than for LRU-managed caches. Secondly, contention-aware policies are designed specifically to minimize cache contention and as a result, the contention occurs in an engineered, well-defined and understood manner. Once such a model is available, it could be easily integrated into the MPPM framework.

As it is, MPPM does not support multi-threaded applications. Multi-threaded workloads not only incur negative interference among co-executing threads but also positive interference, i.e., one thread fetching data that is later accessed by other threads. Furthermore, the thread interaction goes beyond cache sharing alone because of locking, barriers, synchronization, etc. We believe that MPPM can be extended to accurately model the impact of positive interference on overall performance. One way for doing so could be to capture the amount of inter-thread data sharing in the isolated profiling step and incorporate this information into the contention model. Instead of dividing all the ways in the cache according to the access intensity, we would only consider the fraction of the cache that does not contain shared data. Obviously, this will only work if all threads exhibit the same behavior (in the case of a homogeneous multi-threaded application). If not, for instance when a single thread is pulling in all the data for the other threads to use, the per-thread slowdown might not be accurately estimated (per-program performance however could still be estimated accurately).

We have used MPPM for modeling heterogeneous multi-core performance and exploring the heterogeneous multi-core design space. A frame-

work like MPPM could be created to model resource sharing in simultaneous multi-threading (SMT) cores. Yet other avenues for future work are to improve the modeling of sources of contention other than last-level cache sharing, such as bandwidth sharing, TLB sharing, contention in main memory and the impact of prefetching.

6.2.2 Scheduling

The majority of the work we have done on scheduling, has focused on scheduling multi-program workloads. Although we have shown in Chapter 5 that our scheduling algorithms work for multi-threaded applications as well, there are many opportunities left. Multi-threaded applications have different properties compared to multi-program workloads because multiple threads cooperate to get a single task done. This often results in data-sharing and (frequent) thread synchronization. Because the performance of all the application's threads are linked, composing good thread-to-core mappings is a lot harder. Assigning valuable resources (for example, a big core) to a specific thread might not yield better performance for that thread if its performance depends on other threads that are assigned fewer resources. Generally, the performance of a set of tightly coupled threads will be determined by the rate of progress of the thread that has the least resources assigned to it. Obviously, this could lead to suboptimal resource management (scheduling). This observation provides us with many interesting opportunities for future research on scheduling multi-threaded applications on single-ISA heterogeneous multi-core architectures. A first step would be to dynamically identify clusters of threads whose performance are coupled (for instance by quantifying data-sharing by observing coherency traffic). Once these clusters are identified, they can then be scheduled as a single entity on a group of cores (of the same type).

As the number of cores on a single chip continues to increase, the way they are interconnected will start to matter a lot more. Our research has shown that migrating a thread to a different core can be done at low cost. At the very least, this cost will become non-uniform in the future: migrating a thread to a far-away core over a (possibly) complex interconnection network will be more costly than migrating to a nearby core. This effect would need to be taken into account when considering possible thread-to-core mappings. Also, as the number of cores on a chip increases, the number of thread migrations to achieve an optimal schedule increases as well. Additional research is needed to determine if and when the on-chip inter-

connect bandwidth and latency will become restrictive to achieve optimal job-to-core mappings.

Yet another important case to consider are Java workloads (or more generally, any workload written in a managed language). When using managed languages, the application threads are not the only threads in the system. Other co-running threads include garbage collection, compilation and other helper threads. It remains an open question on how to dynamically schedule all these threads to improve overall application performance. Scheduling algorithms need to treat application threads separately from the runtime service threads, and how assigning different resources to different types of threads impact performance.

Multi-threaded applications may also do their own internal thread management and scheduling. A simple example of such behavior is work stealing, where tasks (work) is dispatched to worker threads by a central managing thread. It is unclear how and if dynamic scheduling (either at the OS or hardware level) would interfere with the application-level scheduling. Obviously, the answer to these questions will heavily depend on the particular flavor of internal scheduling, making it a very challenging and interesting topic to consider.

Bibliography

A. Alameldeen and D. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 7–18, 2003.

AMD. The Future is Fusion: The Industry-changing Impact of Accelerated Computing. http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf, 2008.

M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law Through EPI Throttling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 298–309, 2005.

M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor architectures. *Journal of Instruction-Level Parallelism (JILP)*, 10:1–26, 2008.

E. Berg and E. Hagersten. Fast Data-locality Profiling of Native Execution. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 169–180, 2005.

C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

G. Blake, R. G. Dreslinski, T. N. Mudge, and K. Flautner. Evolution of Thread-level Parallelism in Desktop Applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 302–313, 2010.

S. Borkar. Thousand core chips — a technology perspective. In *Proceedings of the Design Automation Conference (DAC)*, pages 746–749, 2007.

T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2011.

F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 171–182, 2004.

D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-thread Cache Contention on a Chip-multiprocessor architecture. In *Proceedings of the Eleventh International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, 2005.

J. Chen and L. K. John. Efficient Program Scheduling for Heterogeneous Multi-core Processors. In *Proceedings of the 46th Design Automation Conference (DAC)*, pages 927–930, 2009.

Y. Chou, B. Fahs, and S. Abraham. Microarchitecture Optimizations for Exploiting Memory-level Parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 76–87, 2004.

J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated Quad-core Opteron Processor. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, pages 102–103, 2007.

H. Dybdahl and P. Stenstrom. An adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 2–12. IEEE, 2007.

E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt. Prefetch-aware Shared-resource Management for Multi-core Systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 335–346, 2010.

D. Eklöv, D. Black-Schaffer, and E. Hagersten. Fast Modeling of Cache Contention in Multicore Systems. In *Proceedings of the Sixth International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*, pages 147–158, 2011.

D. Eklöv and E. Hagersten. StatStack: Efficient Modeling of LRU Caches. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–65, 2010.

P. G. Emma. Understanding Some Simple Processor-performance Limits. *IBM Journal of Research and Development*, 41(3):215–232, 1997.

S. Eyerman and L. Eeckhout. System-level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

S. Eyerman and L. Eeckhout. Per-thread Cycle Accounting in SMT Processors. In *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–144, 2009.

S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Performance Counter Architecture for Computing Accurate CPI components. In *Proceedings of The Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, 2006.

S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2), 2009.

A. Fedorova, D. Vengerov, and D. Doucette. Operating System Scheduling on Heterogeneous Core Systems. In *Proc. of OSHMA workshop, 16th PACT*, 2007.

M. Franklin and G. S. Sohi. Register Traffic Analysis for Streamlining Interoperation Communication in Fine-grain Parallel Processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO)*, pages 236–245, 1992.

R. Gabor, S. Weiss, and A. Mendelson. Fairness Enforcement in Switch on Event Multithreading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3):34, 2007.

S. Ghiasi, T. Keller, and F. Rawson. Scheduling for Heterogeneous Processors in Server Systems. In *Proceedings of the Second Conference on Computing Frontiers (CF)*, pages 199–210, 2005.

P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, 2011.

F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for Providing Quality of Service in Chip Multi-processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 343–355, 2007.

T. R. Halfhill. Intel's tiny Atom. *Microprocessor Report*, 22:1–13, 2008.

G. Hamerly, E. Perelman, and B. Calder. How to use SimPoint to Pick Simulation Points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):25–30, 2004.

M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

Intel. 2nd generation Intel Core vPro Processor Family. http://www.intel.com/content/dam/doc/white-paper/core-vpro-2nd-generation-core-vpro-processor-family-paper.pdf, 2008.

R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 257–266, 2004.

R. Iyer, L. Zhao, F. G. amd R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 2007.

J. Jaehyuk Huh, C. Changkyu Kim, H. Shafi, L. Lixin Zhang, D. Burger, and S. W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(8):1028–1040, 2007.

A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based On-the-fly Multi-core Cache Simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the International Symposium on Computer Architecture (ISCA)*, 2008.

A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. S. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 208–219, 2008.

A. Jaleel, H. Najaf-Abadi, S. Subramaniam, S. Steely, and J. Emer. Cruise: Cache Replacement and Utility-aware Scheduling. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 249–260. ACM, 2012.

A. Jaleel, K. Theobald, J. S. S. Steely, and J. Emer. High Performance Cache Replacement Using Re-reference Interval prediction (rrip). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 60–71, 2010.

M. T. Jones. Inside the Linux Scheduler: The Latest Version of this All-important Kernel Component Improves Scalability. `http://www.ibm.com/developerworks/linux/library/lscheduler/index.html`, 2006, [Online; last accessed 10-February-2013].

N. Jouppi and D. Wall. *Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines*, volume 17. ACM, 1989.

J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49:589–604, 2005.

J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49:589–604, 2005.

T. Kgil, S. D'Souza, A. Saidi, B. N, R. Dreslinski, S. Reinhardt, K. Flautner, and T. Mudge. PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, 2006.

C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-delay Dominated On-chip Caches. In *Acm Sigplan Notices*, volume 37, pages 211–222. ACM, 2002.

S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT))*, pages 111–122, 2004.

P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 125–138, 2010.

R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the ACM/IEEE Annual International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.

R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 23–32, 2006.

R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 64–75, 2004.

N. B. Lakshminarayana, J. Lee, and H. Kim. Age Based Scheduling for Asymmetric Multiprocessors. In *Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable Performance Regression for Scalable Multiprocessor Models. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 270–281, 2008.

T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-asymmetric Multi-core Architectures. In *Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2007.

T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing new Server Architectures for Emerging Warehouse-computing Environments. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 315–326, 2008.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 190–200, 2005.

A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. Composite Cores: Pushing Heterogeneity into a Core. In *45th Annual International Symposium on Microarchitecture (ISCA)*, 2012.

K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT processors. In *Proceedings of the IEEE International Symposium*

on Performance Analysis of Systems and Software (ISPASS), pages 164–171, 2001.

C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. ITCA: Inter-task Conflict-aware CPU Accounting for CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 203–213, 2009.

Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for Multicore Architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

R. McGowen, C. A. Poirier, C. Bostak, J. Ignowski, M. Millican, W. H. Parks, and S. Naffziger. Power and Temperature Control on a 90-nm Itanium Family Processor. *IEEE Journal of Solid-State Circuits*, 41(1):229–237, 2006.

A. Mericas, 2006. Performance Monitoring on the POWER5 Microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press.

P. Michaud, A. Seznec, D. Fetis, Y. Sazeides, and T. Constantinou. A Study of Thread Migration in Temperature-constrained Multicores. *ACM Transactions of Architecture and Code Optimization (TACO)*, 4(9), 2007.

G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), 1965.

T. Mudge and U. Hölzle. Challenges and Opportunities for Extremely Energy-efficient Processors. *IEEE Micro*, 30(4):20–24, 2010.

O. Mutlu and T. Moscibroda. Stall-time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–160, 2007.

O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, 2003.

K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual Private Caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 57–68, 2007.

NVidia. The Benefits of Multiple CPU Cores in Mobile Devices. http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf, 2010.

NVidia. Variable SMP – a Multi-core CPU Architecture for Low Power and High Performance. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance-v1.1.pdf, 2011.

K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-chip Multiprocessor. In *ACM Sigplan Notices*, volume 31, pages 2–11. ACM, 1996.

H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, pages 81–93, 2004.

G. Patsilaras, N. K. Choudhary, and J. Tuck. Design Trade-offs for Memory-level Parallelism on a Asymmetric Multicore System. In *Proceedings of the Third Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA), held in conjunction with ISCA*, 2010.

G. Patsilaras, N. K. Choudhary, and J. Tuck. Efficiently Exploiting Memory-Level Parallelism on Asymmetric Coupled Cores in the Dark Silicon Era. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8, 2012.

M. K. Qureshi and Y. N. Patt. Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–432, 2006.

K. K. Rangan, M. D. Powell, G.-Y. Wei, and D. Brooks. Achieving Uniform Performance and Maximizing Throughput in the Presence of Heterogeneity. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 3–14, 2011.

C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 13–24, 2007.

V. J. Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web Search using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 26–36, 2010.

A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling Performance Variation Due to Cache sharing. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems. *Operating Systems Review*, 43:66–75, 2009.

T. Sherwood and B. Calder. Time Varying Behavior of Programs. Technical report, Citeseer, 1999.

T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, 2002.

A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for Simultaneous Multithreading Processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, 2000.

A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 66–76, 2002.

R. Teodorescu and J. Torrellas. Variation-aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 363–374, 2008.

N. Tuck and D. M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 26–34, 2003.

M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all Starting Points for Simultaneous Multithreading Simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 143–153, 2006.

M. Van Biesbrouck, L. Eeckhout, and B. Calder. Representative Multiprogram Workloads for Multithreaded Processor Simulation. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 193–203, 2007.

M. Van Biesbrouck, T. Sherwood, and B. Calder. A Co-phase Matrix to Guide Simultaneous Multithreading Simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 45–56, 2004.

K. Van Craeynest and L. Eeckhout. The Multi-Program Performance Model: Debunking Current Practice in Multi-core Simulation. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 26–37. IEEE, 2011.

K. Van Craeynest and L. Eeckhout. Understanding Fundamental Design Choices in Single-ISA Heterogeneous Multi-core Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):32, 2013.

K. Van Craeynest, S. Eyerman, and L. Eeckhout. MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor. *High Performance Embedded Architectures and Compilers*, pages 110–124, 2009.

K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 213–224, 2012.

J. Vera, F. J. Cazorla, A. Pajuelo, L. J. Santana, E. Fernández, and M. Valero. FAME: Fairly Measuring Multithreaded Architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 305–316, 2007.

D. Wall. *Limits of Instruction-Level Parallelism*, volume 19. ACM, 1991.

W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23:20–24, 1995.

X. Zhou, W. Chen, and W. Zheng. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 384–393, 2009.