UCLA
UCLA Electronic Theses and Dissertations

Title
Improving Fine-Grained Resource Mapping on Tightly Coupled Heterogeneous Multi-cores

Permalink
https://escholarship.org/uc/item/8h13h3q6

Author
Chen, Robert

Publication Date
2017

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Improving Fine-Grained Resource Mapping on Tightly Coupled Heterogeneous Multi-cores

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Robert Chen

2017

© Copyright by

Robert Chen

2017

ABSTRACT OF THE DISSERTATION

Improving Fine-Grained Resource Mapping on Tightly Coupled Heterogeneous Multi-cores

by

Robert Chen

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2017

Professor Glenn D Reinman, Chair

With increasing power and application demands, heterogeneous multi-core processors are becoming more prevalent. However, the key to proper utilization of heterogeneous multi-cores is assigning, or mapping, the right application to the right core type. Recent work has shown that fine-grained mapping takes advantage of short program phases with highly variant performance requirements, and can elicit greater benefits from tightly coupled heterogeneous multi-cores. This work explores various methods to improve fine-grained mapping techniques to better utilize heterogeneous multi-cores.

The dissertation of Robert Chen is approved.

Mihaela van der Schaar

Todd D Millstein

Milos D Ercegovac

Glenn D Reinman, Committee Chair

University of California, Los Angeles

2017

To my mother, father, and sister . . .

Karen, Yeu-Ching, and Helen Chen,

for their endless support in all ways,

and for whom I would not be here without.

Table of Contents

List of Figures

List of Tables

Acknowledgments

The journey through scholarship can be a lonely one, but I am lucky and grateful to have met others who lit my path well. Apart from my family, to whom I owe the most, I have had the great fortune of meeting many quality people who have not only supported my academic endeavors, but my life as well. To the following people, I express my deepest gratitude:

My committee members -

Glenn Reinman, for your patience and wisdom in guiding me through my work; always polite, witty, and professional, you are the manager I wish to become.

Todd Millstein, for your help and advice at various points through my stay at UCLA.

Mihaela van der Schaar, for providing insightful comments on my work, even when you were not required to do so.

Milos Ercegovac, for your support of my work and your unflagging positivity at all times.

Professors -

Don Browne, for teaching me how to teach.

Lieven Vandenberghe and Bill Howden, for giving me my first shot at research.

Labmates -

Beayna Grigorian, for your positivity, and showing me a life outside of the lab.

Michael Gill, for your wisdom in architecture and current events.

Mohammadali Ghodrat, for being an exemplary teammate, and showing me the true nature of humility.

Hyun Jin Kim, for the late night work sessions, and showing me true time management.

CSGSC members -

Chris Moghbel, for your monumental recruiting efforts and organizational help.

Deepak Rama, for your overall helpfulness in doing the work nobody else wanted to do.

Emanuel Lin, for helping me get everything started on good footing, may you rest in peace.

Justin Collins, David Jurgens, Eric Osterweil, Ran Gelles, Michael Shindler, Katherine Wang, Chris Buonocore, Daniel Quach, Peter Peterson, and Kirill Minkovich, for your various help with the group and our cherished memories working together for the common good.

The good souls -

Shu-yu Guo, for your friendship and singular wit.

Brian Lum, for your friendship in times of need.

Sharath Gopal, for your friendship, and setting the high standard of teaching to which I aspire.

Ganesh Venkatesh, for your friendship, wisdom, and open-armed hospitality.

Jamie Macbeth, for your friendship and your constant good-nature.

Olivera Grujic, for your friendship and willingness to listen.

Jason Lu, and Eric Chang, for your friendship and discussions throughout the years.

Karthika Mohan, Jason Shen, and Arthur Choi, for your friendship and help in times of great stress.

Yitao Liang, and Tal Friedman, for your friendship and our shared discussions of life and the world.

Byron Boon, and Max Salter, for being ideal roommates at the end of each day.

To the aforementioned people and many more unnamed, I am deeply indebted.

Vita

2008	M.S. Computer Science, University of California, San Diego.
2004	B.S. Electrical Engineering, University of California, Los Angeles.

CHAPTER 1

Introduction

The current trend in transistor fabrication is that as transistor size decreases, the power required to run them does not scale accordingly. Yet, chip sizes remain relatively constant. As transistor size decreases, Dennard scaling ends because chip supply voltages can no longer scale with increasing transistor counts [EBA11]. The result is that increasing leakage (static power) causes runaway thermal output, which chip designers call the "power wall". The power wall forces a decision: either not all transistors can be powered simultaneously, or chip sizes must also decrease to scale with power demands. In order preserve computational flexibility, manufacturers have chosen to create multi-core chips where not all parts are powered on simultaneously [RNA12] [NVI15]. This "utilization wall" [VSG10] allows only a fraction of a chip to be powered at any point in time. In this regime, it is imperative that computation be as energy efficient as possible.

One solution to obtaining energy efficiency, while maintaining high utility of an increasing number of transistors, is the use of heterogeneous multi-cores. Heterogeneous multi-core processors have been shown to be more energy efficient and sometimes better in performance than traditional homogeneous multi-cores [BSC08, KFJ03, KTR04]. With mobile devices becoming increasingly mainstream, the energy efficient characteristics of heterogeneous multi-cores make them especially compelling for device manufacturers. Heterogeneous designs are realities today in the form of ARM Ltd.'s big.LITTLE cores [ARM13], and NVIDIA's Tegra X1 [NVI15]. The big.LITTLE design features performance oriented but power hungry "big" out-of-order (OoO) cores, combined with "LITTLE" power efficient but reduced performance in-order cores. The Tegra X1 also features 8 big.LITTLE cores along with 256 power efficient GPU cores.

A key factor in obtaining energy and performance efficiency on heterogeneous multi-cores is the use of scheduling program phases on cores which are suited to phase characteristics. Prior works have experimented with scheduling in a coarse-grained manner with epochs of millions of instructions [BC06, SSJ09, KRH10, CJE12, CJ09]. However, recent work by Lukefahr [LPD12] and Padmanabha [PLD13] show that shorter phases of thousands or hundreds of instructions can achieve greater gains in performance and energy savings than coarse-grained scheduling. This is because fine-grained phases can experience greater performance variance which allows more opportunities for optimization. Coarse-grained policies are less variant in performance because longer phases experience more instruction level parallelism (ILP) that amortize and hide the costs of expensive instructions. For our work, we will refer to "scheduling" as "resource mapping", in order to differentiate ourselves from the OS schedulers of prior work.

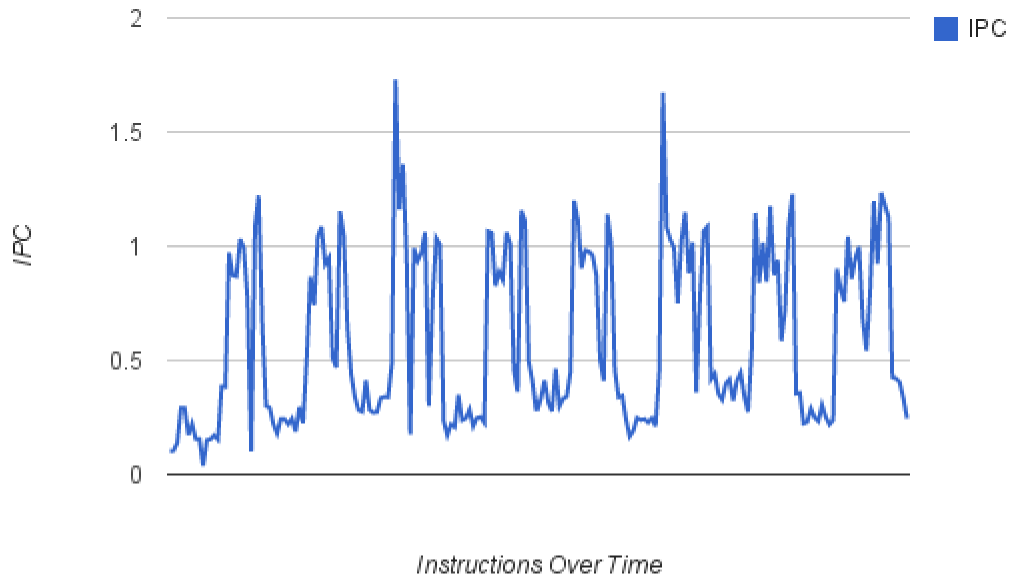


Figure 1.1: Mcf, variant IPC, 500 instr epochs.

1.1 Advantages of Fine-Grained Resource Mapping

Fine-grained resource mapping can be advantageous on tightly-coupled architectures because it can utilize performance and energy differences of a general-purpose heterogeneous core more efficiently. However, producing a resource mapping via sampling or reactive-based approaches can lead to inefficiencies. In cores where fine-grained switching occurs, the short advantageous phases can end soon after a significant sample is taken, or before a reactive approach can cater to the program's resource needs [LPD12, PLD13].

Shorter phases (Figure 1.1) experience more varied and extreme program IPC than longer phases (Figure 1.2), which create more opportunities for performance and energy optimization. Figures 1.1 and 1.2 show the IPC of the mcf benchmark

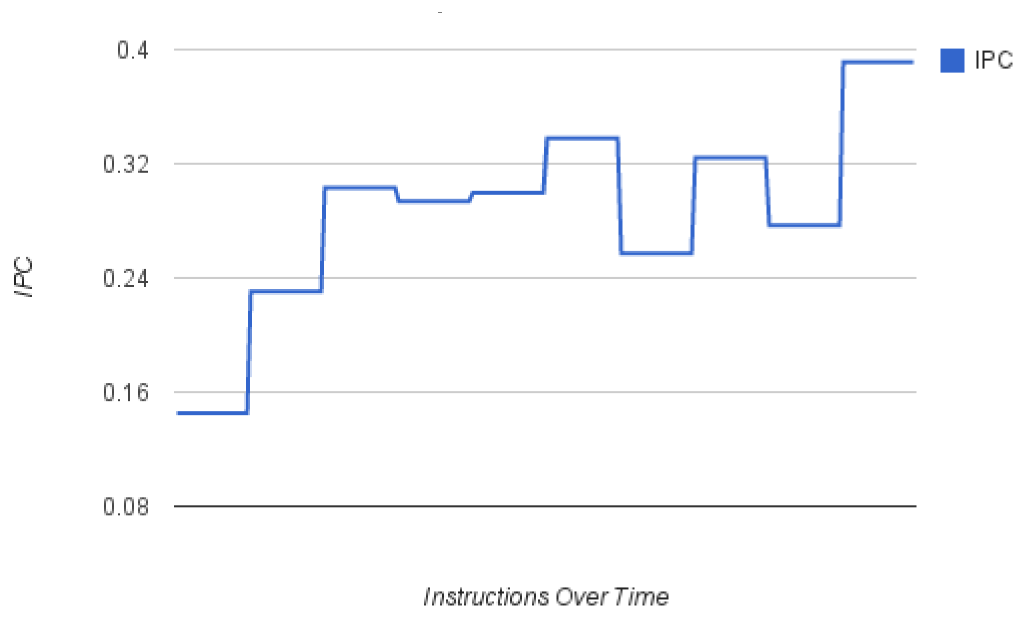


Figure 1.2: Mcf, steady IPC, 10K instr epochs.

for 100K instructions, in periods of 500 and 10K instructions, respectively. The IPC of the short phases in Figure 1.1 vary from 0.15-1.7, whereas the longer phases in Figure 1.2 vary from 0.15-0.4. Longer phases exhibit more stable IPC because costly instructions are amortized, and consequently are unable to realize the benefits of more fine-grained optimization inherent in programs. Even for moderately sized phases of 1028 instructions or more (Figure 2.13), the performance levels off while energy savings decrease, which means anticipating and mapping short phases properly is important. Short phases also incur more switching, which obviates the need for tightly-coupled cores with low switching overhead.

CHAPTER 2

CHILL: Fine-Grained Mapping of Chained High Impact Long-Latency Load Phases

We begin the journey of exploring fine-grained mapping techniques by finding chained, high impact, long-latency load phases (CHILL). Bottlenecks in performance can occur in fine-grained program phases during chains of high impact long-latency loads. We design a system that detects these bottleneck phases, and propose accelerating these phases on the out-of-order core for better performance and energy efficiency.

2.1 CHILL Phases

Long-latency loads, or loads which miss in the last level of cache, have often been found to be sources of bottlenecks in programs [SSJ09, CWT01, AR94]. Chains of LLLs which depend upon each other are also important bottlenecks in program performance because they consume memory bandwidth, MSHR entries, and other instructions dependent on them consume space in the instruction window. These chains of LLLs also attract more dependent non-load "shadow" instructions than normal individual LLLs (Figure 2.2), making these LLLs “high impact”. CHILLs have geomean shadows of 3.15 instructions, compared to individual LLL shadows of 2.1. We find that the IPC of CHILL phases is 9.58x worse (geomean) than that of non-CHILL phases (Figure 2.1), and that programs can spend a geomean of 21% of cycle time in CHILL phases. The combination of reduced IPC and the amount of time spent on CHILL phases means that efficiently executing through these phases is worthwhile.

CHILL phases refer to both epochs in which either actual CHILLs or their dependent shadow instructions occur, while non-CHILL phases have neither. This is illustrated in Figure 2.3, where a period of 20 epochs is shown from mcf. The first three epochs contain CHILLs, and experience the lowest IPC. The next 14 epochs do not contain CHILLs, but experience variant IPC because they contain shadow instructions dependent on the CHILLs. Knowing this, it is important to accelerate regions beyond the epochs in which CHILLs themselves occur, in order to capture the entire CHILL phase.

To determine which type of core is better suited to running CHILL phases, we tested their performance on OoO and in-order cores (Figure 2.4). CHILL phases perform better on the OoO core, with a 33.8% geomean increase in IPC over the in-order core. Although some benchmarks like astar, bzip, and gcc exhibit small losses in IPC on the OoO, we are still able to achieve reasonable gains.

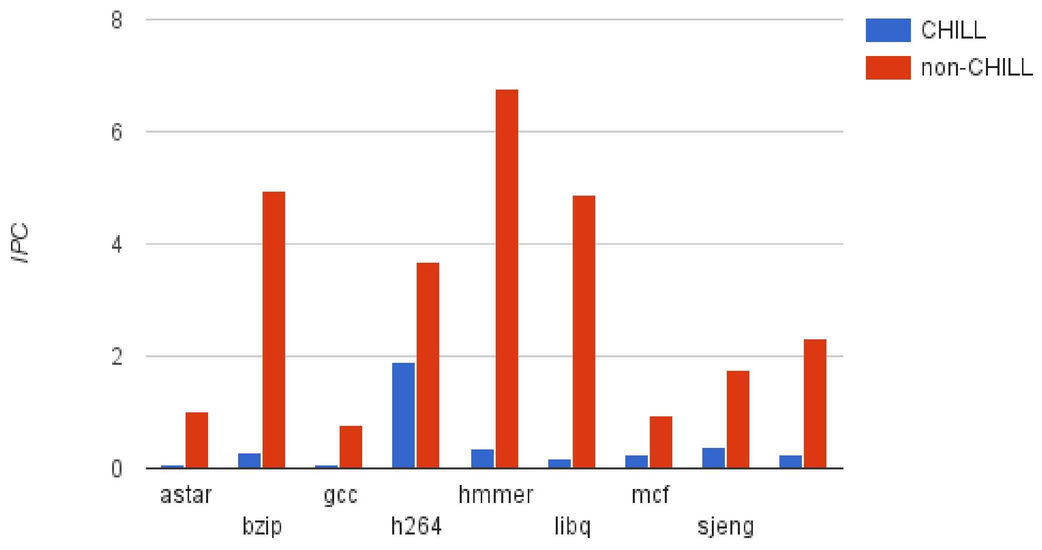


Figure 2.1: Worse IPC for CHILL phases.

2.2 Work Related to CHILL

2.2.1 Load Criticality and Dependence

Part of our work relies on finding LLLs at runtime, and amongst various techniques we highlight the work that is more closely related to our analytical techniques. Collins, et. al. speculatively pre-compute slices of programs which lead to delinquent loads, which leads to reduced wait times for future data cache misses [CWT01]. Panait, et. al. refine the static technique for identifying delinquent loads via basic block profiling in post-compilation [PSW04]. Both techniques rely on identifying possible "delinquent" loads as critical during compilation or post-compilation. Our system identifies dynamic LLLs and correlates them directly with criticality at runtime. Srinivasan, et. al. dynamically identify critical loads in hardware based on various factors [SJL01]. They classify critical

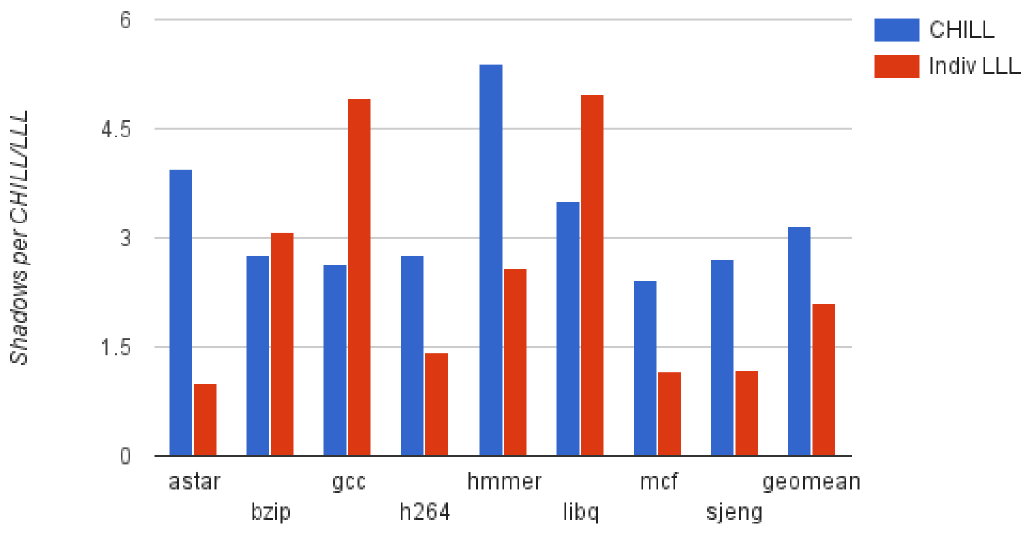


Figure 2.2: Shadow instr differences.

loads into three categories: 1) loads which feed mispredicted branches, 2) loads which feed other loads that miss in L1, and 3) loads in which the number of independent instructions within a window of instructions is below a certain threshold. With this criticality information, they modify the cache to speed up critical loads in a manner similar to that of a victim cache. Our analysis differs in that we find critical phases during chains of high impact L2 load misses to be bottlenecks, and take advantage of modern heterogeneity of processors to accelerate through bottleneck phases. Furthermore, our solution for utilizing this information is different in that we take advantage of modern heterogeneity in processors to accelerate through critical phases while saving energy.

Another class of work finds data dependencies at runtime in hardware, but their techniques differ from our system in various ways. Roth, et. al. create a prefetching scheme based on finding chains of pointer-based loads for linked data

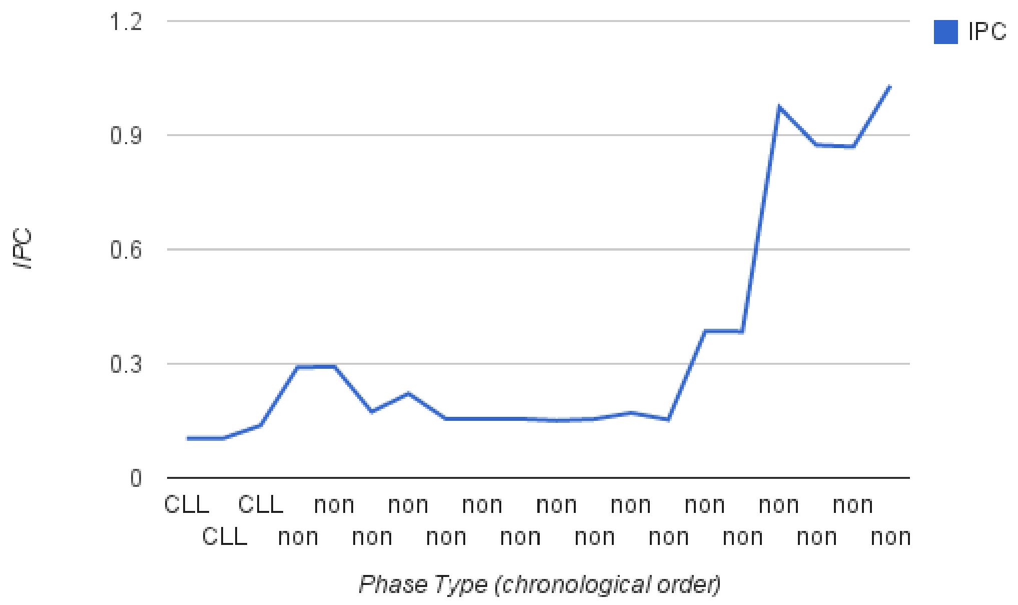


Figure 2.3: Snapshot of mcf, certain non-CLL phases require acceleration.

structures [RMS98]. Their system includes tables for loads in progress and known dependencies. However, their technique is limited to identifying dependencies between two instructions, rather than long chains of loads. Raasch, et. al. enhance the performance of the instruction queue with dependency lanes [RBR02]. Each lane represents a chain of instructions dependent on a load to a particular register. Instructions are sniffed for source dependencies, and added to chains with estimated delays until issue time. Our mechanism differs in that we keep simpler bit-vectors to store dependence chains, and perform merging of chains to save space. Chen, et. al. create a system for tracking data dependencies dynamically for enhancing branch prediction [CDA03]. They use a FIFO of bit-vectors to track dependencies of all running instructions. Our system differs in that we only track LLLs and their dependent instructions, and remove entries when the lifetime of

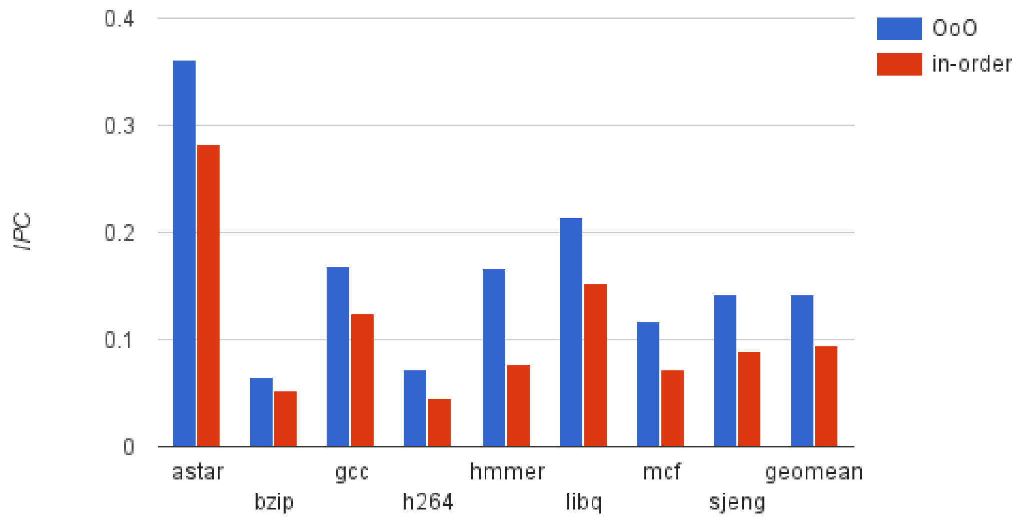


Figure 2.4: CHILL IPC on OoO and in-order.

the LLLs end.

A different set of work attempts to utilize LLL information to directly influence pipeline design. The load slice architecture [CHA15] adds OoO properties to an in-order core. It enhances an in-order core with structures that allow out-of-order execution of loads and the address-generating instructions which precede them. This is accomplished by performing iterative backward dependency analysis via the backward slice technique [ZS00]. Our system differs in that we find and accelerate the instructions after a series of chained LLLs, although the load slice architecture's backward dependency analysis could potentially improve our performance during these bottlenecks. Another system that uses a similar type of backward slice analysis is the Long Term Parking system [SCH15]. Their goal is to decrease the number of instruction queue and and register file of an OoO core by creating a separate staging area (long term parking) for non-critical instructions.

The idea is to keep non-critical instructions from polluting the OoO pipeline during critical phases. They use the same iterative backward slice analysis [ZS00] as in the load slice architecture, but add their analysis to the RAT during the renaming phase. Our system differs in that we track the critical LLL phases and their dependent instructions directly, and burst through these phases. Both systems also differ from ours by using iterative backward slice analysis, which finds dependent instructions one at a time per loop iteration, incurring a warm up time. In contrast, our technique is able to find the full set of dependent instructions in one loop iteration.

2.2.2 Heterogeneity and Scheduling

Prior work show the benefits of heterogeneous cores for both energy efficiency and performance [BSC08, KFJ03, KTR04, LPD14]. An abundance of other work use various metrics to schedule programs statically [CJ09, SSJ09] or dynamically [BC06, CJE12, KRH10]. These techniques operate at granularities from millions of instructions to milliseconds of time, and often operate in the OS [MPM14] or even at the user-level [PLD15b].

For finer granularities, Lukefahr [LPD12, LPD16] and Padmanabha [PLD13] show that thousands or hundreds of instructions can achieve more gains, which is the computing substrate on which we focus. Lukefahr proposed a tightly coupled multi-core with two backends, one OoO and one in-order, which can be dynamically switched at runtime. Their mapping algorithm is based on an a priori ridge regression analysis of core metrics, and uses the runtime core metrics to make decisions based on the prior analysis. With this, they achieve fine-grained mapping at 1000 [LPD12] [LPD16] instructions per mapping interval. Padmanabha's work extends Lukefahr's by finding program phases with branch history signatures. They coalesce pieces of past branch target addresses to mark loop starting points, and associate these points with runtime performance. Our tightly-coupled heterogeneous core model is similar to that in [LPD12], while our mapping granularity is similar to the trace-based work [PLD13] at hundreds of instructions. However, we differ by extending the mapping algorithm with CHILL analysis, and we do not use the branch history signatures in the trace-based system. In DynaMOS [PLD15a], the same authors create a technique for memoizing OoO schedules that are consistently repetitive, and running them on the in-order core for energy efficiency. By capturing and storing certain OoO instruction sequences, they can replay these sequences on the in-order core for better energy efficiency. Our system differs from DynaMOS in that it discerns problematic phases, accelerates these phases accordingly, and performs reactive mapping in other phases.

Another proposed fine-grained mapping scheme is a morphing core [SRA13]. Srinivasan et. al. create an architecture similar to Lukefahr's, however, it contains one backend that changes between OoO and in-order configurations on-the-fly. A transition from OoO to in-order mode requires turning off parts of the fetch and decoding logic, LSQ, ROB, RAT, and some execution units. They base their switching mechanism on a power/Watt metric, and switch in intervals of 500 instructions. Our scheme differs in that we use two backends, one for each OoO and in-order cores, with shared fetch logic. We also differ in that our switching mechanism is based on both CHILL analysis and performance metrics, on which we perform a priori ridge regression analysis, and adapt at runtime.

2.3 CHILL System Design

Overall, our hardware system captures retiring instructions, tracks their dependencies in a table of bit-vectors which correspond to active registers, and finds and records CHILL phases from the tables of dependencies. We also store CHILL phase starting points by tracking the last backward branch PC, and keep count of the duration of CHILL phases. CHILL phases end when all of the LLLs in the chain are "killed"; when other instructions write to the same destination register as the LLLs'. During the CHILL phases, we track shadow instructions and migrate between the OoO and in-order cores in a semi-reactive manner: encountering more shadows keeps the program on the OoO core, while fewer shadows initiate migration to the in-order core. In non-CHILL phases, we default to a standard reactive mapping mechanism. The tightly coupled multi-core architecture (Figure 2.5) used contains one frontend fetch engine, with two execution backends (OoO and in-order), both feeding statistics into the CHILL system.

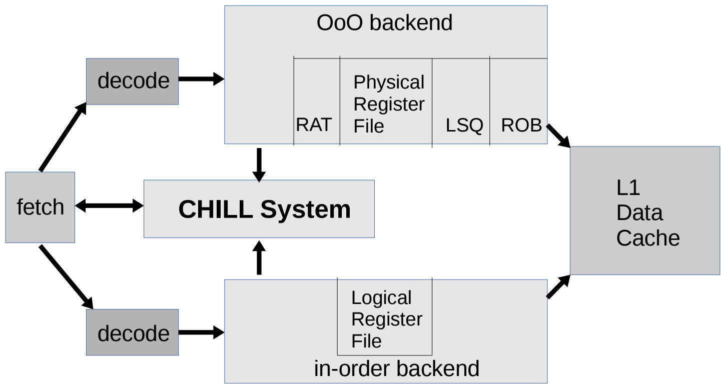


Figure 2.5: Two backends, one fetch engine, with backends communicating with CHILL system.

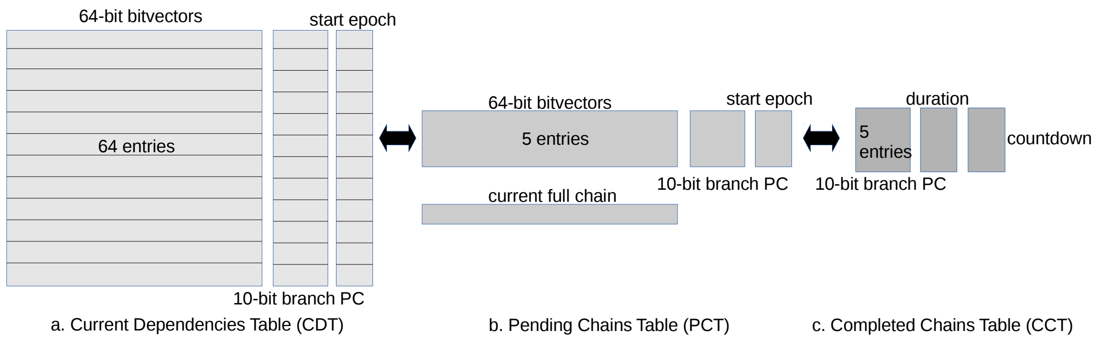


Figure 2.6: CHILL system: 64 64-bit bit-vectors (CDT), 5 PCT, and 5 CCT entries.

2.3.1 System Components

The CHILL system uses three tables (Figure 2.6) to detect and manage chains of LLLs: Current Dependencies Table (CDT, Figure 2.6a), Pending Chains Table (PCT, Figure 2.6b), and Complete Chains Table (CCT, Figure 2.6c). As instructions retire, they are analyzed in the manner described in Section 2.3.2 for dependencies to LLLs in the CDT. When a kill of a LLL occurs, certain entries

Program Order			a.	b.	c.	d.
	A. **ld r1 imm**	8.	11001	10000	0	11000
	B. add r2 r1 r3	7.	1001	0	1001	1000
	C. add r3 r2 r4	6.	1001	0	1001	1000
	D. **ld r4 r1**	5.	11001	10000	0	11000
	E. **ld r5 r4**	4.	1001	0	1001	1000
	F. add r6 r4 r7	3.	1	1	1	0
	G. add r7 r6 r8	2.	1	1	1	0
▼	H. add r8 r5 r4	1.	1	1	1	0
		entry #	a.	b.	c.	d.

Figure 2.7: Example of code analysis during CHILL phase.

are removed from the CDT, and placed into the PCT. The PCT holds chains of LLLs that are being constructed and have not all been killed. When a chain in the PCT ends, its information is passed into the CCT, which aids in predicting CHILL phases. In essence, the CDT tracks all LLL dependencies, the PCT tracks ongoing CHILLs (graduated from the CDT), and the CCT remembers the completed CHILL phases (graduated from the PCT) for future prediction.

2.3.2 Dependence Tracking (Populating the CDT)

The CHILL dependence analysis relies on observing relations between instructions' source and destination registers. Figure 2.7 provides an illustrative sample of source code, with bolded instructions being LLLs. We track the dependencies in a table of bit-vectors, seen in Figures 2.7a, b, c, and d. Each of Figures 2.7a, b, c, and d represent the entire CDT at various points in the program, with row entries numbered in increasing order from bottom to top. The 64 CDT entries contain a 64-bit dependence bit-vector, a 10-bit last backward branch PC, and a 5-bit start

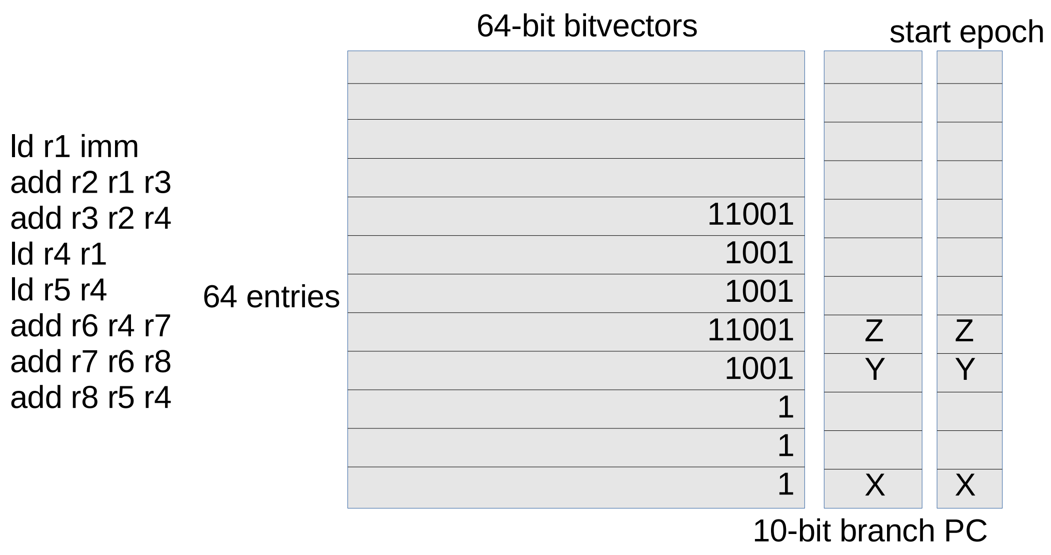


Figure 2.8: CDT with code example, branch PC, and start epoch filled for CHILLs.

of epoch entry.

To begin, Figure 2.7a represents the CDT after all of the instructions in the sample code have been retired in order. Each row in the table of bit-vectors (Figure 2.7a) represents the dependencies for the Nth register; the first row (bottom row) represents the dependencies for register 1, etc. When a LLL is encountered, like instructions A, D, and E, we set the Nth bit in the Nth row to 1 in order to signify a LLL dependence (where N is the destination register number). For example, instruction A in Figure 2.7 sets the first bit of the first bit-vector to 1. Instruction D sets the 4th bit in the 4th bit-vector to 1, and because the result is dependent on register 1, it stores the union of bit-vectors 1 and 4 (1001) in row 4.

Other shadow instructions which depend on the LLLs will add their dependencies to the bit-vector table, such as instructions B, C, F, G, and H in Figure 2.7.

To create the dependence bit-vector for every instruction, we use the instruction's source register numbers and index into those rows in the CDT. We then compute the union of those source rows in order to find all of the other registers the instruction is dependent upon. Then, we use the instruction's destination register number to index into the CDT, and compute the union of the previous source register rows with the destination row. The entire union of bit-vectors is stored in the row represented by the destination register. For example, in Figure 2.7a, instruction H creates an entry of 11001 in the 8th row of the table. The union of the 4th and 5th rows is computed, because the instruction's source registers are r5 and r4. This union is then stored in the 8th entry, which signifies that r8 depends on LLLs to r1, r4, and r5. Figure 2.8 shows the CDT filled with the example code in Figure 2.7. If the CDT is modified with a shadow instruction, the system indicates to the prediction mechanism in Section 2.3.5 that the epoch has encountered high impact shadows.

Chains of LLLs can be identified as Nth rows whose Nth bit is set to 1, and which have other bits within the bit-vector set to 1, such as rows 4 and 5 in Figure 2.7a. Rows 2, 3, 6, 7, and 8 do not represent CHILLs because their Nth bits are not set to 1. Although row 1 has only its first bit set to 1 and may seem like an independent LLL, rows 4 and 5 also have their first bits set to 1, and hence r1 is part of the CHILL of 1, 4, and 5. It should be noted that only the occurrence of LLLs can begin the population of the bit-vectors. The core can identify loads as being long-latency if they do not return a result within the L2 hit time, which

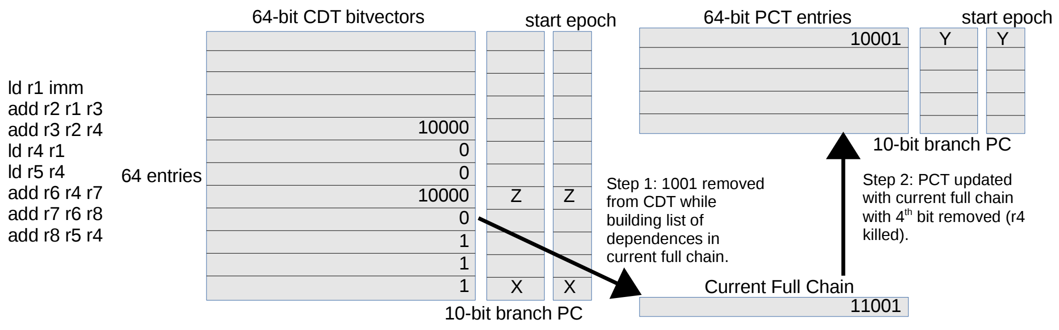


Figure 2.9: R4 killed, graduating from CDT to PCT, pending (not yet completed) CHILL phase recorded.

is 20 cycles in our system.

Bit-vector entries are cleared when a LLL is killed. Figure 2.7b shows the result of register 4 being killed; only subsets of bit-vectors matching exactly the 4th bit-vector are set to 0. Figure 2.7c shows a superset bit-vector kill, in which only exactly matching supersets of the 5th bit-vector are set to 0 in the table. Finally, Figure 2.7d shows the result of a single LLL dependency being killed in register 1, where all subsets are set to 0.

2.3.3 Collecting and Merging CHILLs (CDT to PCT)

When a LLL's register is killed in the CDT, we begin to form chains in the PCT (Figure 2.6b). The PCT contains 5 entries of 64-bit bit-vectors which hold chain register identities, and the corresponding 10-bit last backward branch PCs, and a 5-bit start epoch of the first LLL in the chain. The current full chain entry is a temporary store for the remaining unkilled LLL registers that the entire chain is waiting on.

Referring back to Figure 2.7 illustrates what occurs in the CDT during kills of LLLs, and how they are graduated to the PCT. Figure 2.7a shows the state of the CDT after retiring all instructions in the example code before any kills occur. Upon a LLL kill, we first examine the CDT entry being killed, and traverse the CDT to search for other entries containing the killed register's identity, and place this result into the current full chain entry of the PCT. In doing so, we attempt to find the longest current chain for which the killed entry is a member. For example in step 1 of Figure 2.9, if r4 is killed, the system uses row 4 and begins searching from row 1. Row 1 does not contain a 1 in the fourth bit position (because r4 is being killed), so row 1 is left untouched. The system continues until it reaches row 5, which has a 1 in the 4th bit position, meaning that r5 somehow depends on r4. Here, row 4 is unioned with row 5, and the result is saved in the current full chain entry for further iterations in the CDT. Traversing rows 6, 7, and 8 result in three more unions because all three rows contain a 1 in the 4th bit position. After traversing the entire CDT, the final result of all the unions is 11001, and becomes the current full chain entry, which represents all LLLs dependent on r4. During traversal, the entire sub-bit-vector of r4 (1001) is removed from any row which contains a perfect match, resulting in a CDT that looks like Figure 2.7b. In other words, only the 1001 pattern is removed from every CDT row because that was the entry in row 4 at the time of r4's killing. Notice that killing any of the LLLs in Figure 2.7 will result in a current full chain entry of 11001.

Next, the system attempts to add the current full chain entry into the PCT, seen in step 2 of Figure 2.9. If the PCT already contains entries, we search the PCT for any matches with any of the valid bits in the current full chain. If a match occurs, signifying a pending chain merge, the matching PCT bit-vector entry is updated with the union of its bit-vector and the current full chain. If no current PCT entries match any of the valid bits in the current full chain, a new PCT entry is created with the current full chain as the bit-vector. The corresponding backward branch PC, and start epoch from the killed CDT entry are copied into the new PCT entry. Finally, we kill the actual identity bit in the corresponding PCT bit-vector. For example, killing r4 will result in a PCT bit-vector of 10001, which means that r4's load dependencies have ended, and that registers 5 and 1 remain active in the chain.

2.3.4 CHILL Tracking (PCT to CCT)

The CCT holds CHILL phases which have been completed, which are graduated pending chains from the PCT. Each of the 5 CCT entries is represented by a 10-bit last backward branch PC, a 5-bit duration entry, and a 5-bit countdown entry. After killing a PCT entry, we check whether that entry became 0, which signifies that a pending chain has ended with no more LLLs, and is ready to become a completed chain in the CCT. If so, a chain has ended and we must update the CCT in Figure 2.6c. We first check whether the killed PCT's backward branch PC matches any entries in the CCT. If there is a match, this indicates that an already completed chain is being updated, and we update the CCT entry with the oldest backward branch PC, and the longer duration. The duration is calculated by subtracting the start epoch from the current epoch number during the kill. If there is no matching CCT entry, we create a new one by moving the killed PCT's last backward branch PC, and duration into the CCT. Finally, the countdown entry in the CCT is set to 0.

2.3.5 Prediction Mechanism

The prediction mechanism utilizes the information in the CCT, and whether the program is encountering high impact shadow instructions. The mechanism runs as instructions are retired, and predictions will vary depending on which core the program is currently located. Overall, the system tries to execute the program on the OoO when it encounters shadows during CHILL phases, and uses a standard reactive prediction mechanism during non-CHILL phases.

When a program begins, there will be no completed chains, and the system will have no knowledge of pending chains. To mitigate such "cold start" phases, we check every epoch whether the CDT has created a PCT entry, or if a CDT entry was modified with non-load shadow dependencies. If this occurs, we migrate the program to the OoO core for the next epoch. This technique allows us to run more optimally during cold start by running on the OoO when shadow instructions dependent on CHILLs have an impact.

As a program executes, it will populate the PCT and CCT with pending chains in progress and completed chains, respectively. Every CCT entry contains a duration field which contains the number of epochs from the first LLL in the CHILL phase to when the last LLL in the chain is killed. We detect whether the program is within a completed chain by comparing branch target PCs of retiring instructions to the last backward branch PCs recorded in the CCT. If there is a match, then the program has entered a CHILL phase, and the program is transitioned to the OoO core for the next epoch. Upon entering a completed chain, the system sets the countdown field of the CCT entry to match the duration field. Every succeeding epoch, the countdown is decremented by 1. Multiple completed chains can be active at once, and their countdowns are decremented simultaneously.

While a program is running when there are positive countdowns in the CCT or active pending chains in the PCT, the program will be active on the OoO core during CHILL phases, but we also need to carefully switch to the in-order core during opportune moments for better energy efficiency. To take advantage of in-order energy savings during active pending chains and live completed chains, we use a 4-bit saturation counter that counts the number of high impact shadow instructions. If shadow instructions are encountered during the epoch, the saturation counter is incremented by 1, and decremented otherwise. While executing on the OoO, when the saturation counter falls below half of its maximum value, we estimate that the program has not modified an active chain for a significant period of time, and switch execution to the in-order core. If a CHILL is encoun-

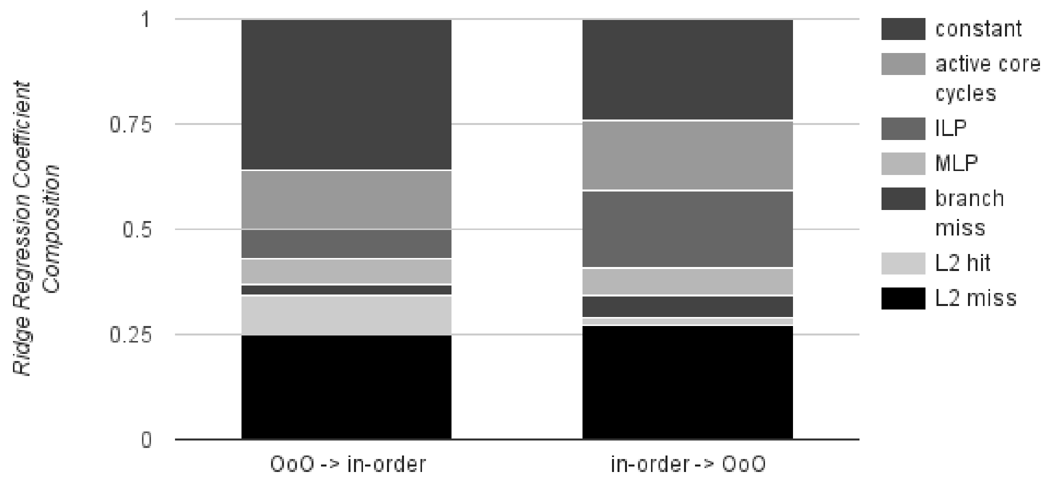


Figure 2.10: Ridge Regression Analysis Coefficient Composition

tered while on the in-order core, the saturation counter is set to its maximum value and the program switches to the OoO in anticipation of new shadows. In this way, we attempt to maximize performance and energy savings during CHILL phases while in the presence of active chains.

2.3.6 Base Reactive Mechanism

During non-CHILL phases, we revert to a base reactive mechanism which depends on measurements of MLP, ILP, L2 miss and hit rates, and branch misprediction rates, similar to the one used in [LPD12]. For ILP and MLP measurements, we use a system similar to one described in [CDA03], and leverage information in our own CDT. Prior to experimental runs, we gathered data on the first 10 millions instructions of our benchmarks with the SPEC 2006 test input. We perform a ridge regression analysis on the metrics to determine a correlation between these statistics on the OoO core and the in-order core's performance, and vice versa.

$$\Delta CPI_{OoOdecision} = \sum CPI_{observed} + \alpha \sum CPI_{pasterror} + \beta CPI_{currenterror} - CPI_{in-orderestimated} \quad (2.1)$$

Figure 2.10 shows the results of our ridge regression analysis, and how OoO and in-order statistics correlate to the performance of their counterparts.

With this profiling information, we estimated the performance of the non-active core at runtime, and map the program reactively using a standard proportional-integral controller mechanism like the one in [LPD12]. Equation 2.1 shows the proportional-integral used to calculate a mapping decision while currently running on the OoO core. We track the total $CPI_{observed}$ and all past errors in $CPI_{pasterror}$, as part of the integral term. Our proportional term is based on the estimate of CPI on the in-order core $CPI_{in-orderestimated}$, and the error of that estimate, $CPI_{currenterror}$, which comprises of the difference between $CPI_{in-orderestimated}$ and the observed CPI of the current epoch.

The decision is made depending on the result of $\Delta CPI_{OoOdecision}$. If the result is negative, that means that the estimated in-order CPI will be better, which indicates that the system should map to the in-order core. If the result is positive, the overall CPI experienced on the OoO core is better than the estimated in-order CPI, indicating that the program should remain on the OoO.

2.3.7 CHILL Buffering

Although the CHILL system assists with quality-of-service, which does not require absolute time constraints for its operation, we nevertheless analyze its timing characteristics in CACTI [SJ01]. When the system encounters a CHILL kill it accesses all 3 of the CDT, PCT, and CCT (86 max accesses), this requires 21.72 cycles. LLLs which do not kill a register require 64 accesses to the CDT, for a total of 18.53 cycles. For all other instructions, only the CDT is accessed 3 times (destination register and 2 source registers), which requires 0.87 cycles. Our system runs in epochs of 512 instructions, and ideally, the CHILL system would need to finish its analysis within the span of an epoch. Our 3-wide issue OoO core experiences an average run time of 1663.75 cycles during any epoch with CHILL kills, while our 2-wide in-order core experiences an average run time of 2686.25 cycles. At most, our benchmarks experience 46 LLLs during an epoch, of which at most 6 are CHILL kills, for a total of 1404.54 cycles. Hence, in the worst case of epochs with LLLs and CHILL kills, the CHILL system will finish its analysis within the span of an epoch.

Epochs in which there are no LLLs of any kind experience much better CPI, which means that either the CHILL analysis needs to be completed more quickly, or the system needs to buffer some instruction signatures to keep pace with the performance of the cores. The theoretical peak performance of our fastest 3-wide issue OoO core would be 170.67 cycles per 512-instruction epoch. For epochs

Table 2.1: Core Parameters

OoO core (1GHz)	3-wide issue, 15-25 stage pipeline, 128 ROB, 160 reg file
In-order core (1GHz)	2-wide issue, 8-10 stage pipeline, 64 entry reg file
Memory System (shared)	32kB L1 I/D-cache (4 MSHR), 2MB L2 cache (8 MSHR), 4GB RAM

with no LLLs, each instruction requires 0.87 cycles of analysis time, for a total of 445.44 cycles. It would seem that the CHILL system needs to buffer 275 instruction signatures to keep pace during times of high performance, but this is unnecessary because our benchmarks never achieve theoretical peak performance. Our benchmarks run an average of 576.88 cycles per epoch on the OoO, and 1202.75 cycles on the in-order, which means that most of the time the CHILL analysis will finish before an epoch completes. In the worst case, libquantum on the OoO experiences 357 cycles in an epoch, which means that the CHILL system requires 89 instruction signature buffers at most.

2.4 Methodology

We model our tightly-coupled heterogeneous core similar to the Composite Core described in [LPD12], in which the OoO and in-order cores are modeled after ARM's big.LITTLE [Gre12]. The big OoO core is modeled after the Cortex-A15, which is 3-way issue with a pipeline depth of 15-25 stages. The LITTLE in-order core is modeled after the Cortex-A7, which has a shorter pipeline of 8-10 stages with 2-way issue capability, depending on instruction dependencies.

In the tightly-coupled design, the big and LITTLE share L1 and L2 caches, a fetch unit, branch predictor, and the CHILL system. Each core has its own decode and OoO/in-order execution engines. Switching, or migrating, from the big to the LITTLE requires draining the OoO pipeline (commit as many instructions as possible, and flush non-committable instructions), and traversing the RAT to restore values from reservation stations into the architecturally visible registers. The vice versa occurs during LITTLE to big transitions. The switching delay depends on the number of instructions in the pipeline and the amount of register state which needs to be preserved, which requires 30 cycles on average [LPD12]. Current OS switching on big.LITTLE takes 20ms [Gre12]. Although it is possible to clock gate the inactive part of the core, we opt for the most pessimistic power consumption model, in which we account for both big and LITTLE static power during the entirety of execution. We find that this static power model would be fairly realistic in near future tightly coupled architectures.

Our CHILL system requires 5615 bits in table space for the CDT/PCT/CCT, and 1691 bits for 89 instruction signature buffers. Each instruction signature entry contains 3 6-bit entries for storing 1 destination and 2 source register numbers, and 1 extra bit to signify whether the instruction retired was a LLL. Using CACTI [SJ01], we find that our design requires 0.023 mm2 of area, which is negligible overall. The complete system is accessed on every CHILL kill, and once at the end of every epoch. Non-load instructions access only the CDT. This leads to an estimated power consumption of 0.0433W. We factor the power consumption into our simulations, but it does not affect our results significantly.

We run benchmarks from SPEC 2006 [Cor06] in the Simics [MCE02] and GEMS [MSB05] simulators. Core power and energy are modeled in McPAT [LAS09]. Table 2.1 provides more details on the architectural parameters simulated. All simulations execute over 15 million instructions, with 512-instruction epoch size, and compiled for SPARC v9 with -O3 optimization.

2.5 Results and Analysis

2.5.1 Mapping Mechanisms

We compare CHILL against other types of fine-grained resource mapping systems. First, we compare against an oracle resource mapper, which tolerates a 5% decrease in performance while seeking opportunities for energy efficient computation. Second, we compare with a reactive mapper with characterstics described in Section 2.3.6. It dynamically correlates core performance to certain core statistics gathered at runtime. Prior to simulation, the MLP, ILP, L2 hit and miss rates, and branch misprediction characteristics are correlated to performance in pure OoO and in-order runs. The mapping is then included in the system at runtime for reactive mapping. Third, we compare against a reactive mapper with a 4-bit saturation counter to help smooth perturbations inherent to a purely reactive scheme. Finally, we compare with the trace-based performance prediction scheme proposed in [PLD13]. This scheme relies on creating identifiers for different phases

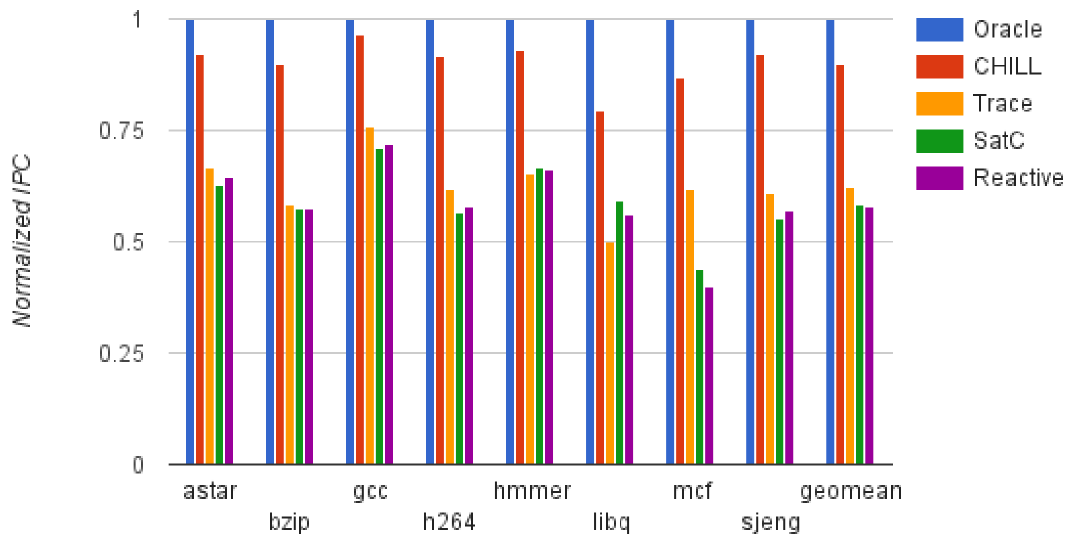


Figure 2.11: IPC normalized to oracle.

of execution by combining pieces of past backward branch PCs, and phases are correlated with performance at runtime. Although there are energy improvements to the trace-based system in DynaMOS [PLD15a], we find that the small in-order phases of DynaMOS are not captured frequently enough to make a significant impact on the trace-based system, so we elect to compare with trace-based without DynaMOS. Both the saturated and purely reactive systems we test are similar to the reactive system in the trace-based scheme. The advantage of the trace-based scheme is in its identification of phases, while the reactive systems do not contain such a feature.

2.5.2 Performance

Figure 3.8 shows IPC normalized to the oracle for all four mapping schemes. Overall, the CHILL system operates at a 10% loss to the oracle, while the trace-based

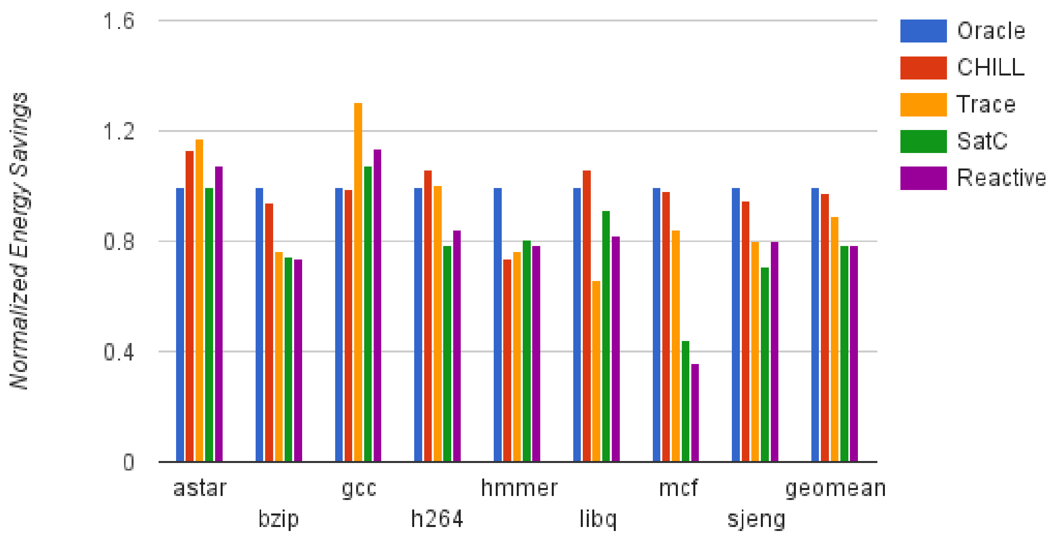


Figure 2.12: Energy savings normalized to oracle.

system operates at a 37.8% loss. The saturated counter scheme loses 42.5%, while the reactive scheme loses 42.9%. For the hmmer benchmark, the trace-based system experiences a 1% performance loss compared to the saturated counter and reactive systems. This is because hmmer contains more branch mispredictions which cause more core switches than necessary because the trace-based system uses branch PCs as phase markers. This also explains the libq drop in performance for the trace-based system. Here, the saturated counter and reactive systems experience about a 7.5% increase in performance than trace-based. The overall performance could be improved for CHILL phases with more serious branch misprediction rates, as in libq and mcf. Further details in performance are also explained in Section 2.5.3

2.5.3 Energy

Energy savings normalized to the oracle are shown in Figure 3.9. Overall, the CHILL system operates at a 2.6% loss in energy compared to the oracle. The trace-based system experiences a 10.8% loss, while the saturated counter and reactive schemes experience 21.6%, and 21.7% energy losses compared to the oracle, respectively.

There are a few instances of systems beating the oracle on energy savings. The astar benchmark experiences relatively few CHILL phases, which means there are more opportunities for energy-saving in-order execution. Both the CHILL and trace-based systems detect this, with the trace-based system gaining 4% more energy savings than CHILL. The saturated counter and reactive systems also detect the increased opportunities for energy savings, but save about 10% less energy than the CHILL and trace-based systems. The price for saving energy on astar is decreased performance compared to the oracle, as seen in Figure 3.8.

The gcc benchmark experiences phases with many CHILLs, but mixed with high branch misprediction rates during those CHILL phases. This causes the CHILL system to run unnecessarily on the OoO during the CHILL phases, while the trace-based, saturated counter, and reactive systems run with more energy efficiency on the in-order. They save 30%, 7%, and 14% more than the oracle, respectively. However, these energy savings come at costs in performance of 24.2%, 29.1%, and 28%, respectively in Figure 3.8. CHILL only operates at a 1.4% energy loss, with respect to the oracle, while maintaining a 3.8% loss in performance.

H264 exhibits more streaming data behavior, which the CHILL system can detect and power through quickly and switch to the in-order core for other phases. The CHILL system achieves 6.3% more energy efficiency than the oracle, but pays a performance penalty of 10.3%. The branching behavior is more predictable, so the trace-based system is also able to achieve near oracle-level energy efficiency.

For hmmer, the trace-based, saturated counter, and reactive systems collectively operate at about 5% more energy efficiency than the CHILL system. This is because hmmer has a combination of more frequent CHILL phases with varying branch misprediction rates. The frequent CHILL phases map more frequently to the OoO core, which reduces energy savings. However, the trace-based system is more reactive during phases with varying branch misprediction rates on the in-order core, because its trace mechanism is based on merging backward branch PCs. This causes more unnecessary core switching in the trace-based system, which leads to a slight 1% performance drop, relative to the reactive systems (Figure 3.8). The saturated counter and reactive systems both have built-in mechanisms to detect branch misprediction rates (Section 2.3.5), and are able to obtain comparable energy efficiency. Nevertheless, the CHILL system exhibits about 23% more in performance than the other systems.

The libq benchmark experiences relatively few CHILL phases, but has separate periods of high branch misprediction rate. This combination of characteristics allows the CHILL system to find more opportunities for energy savings, with

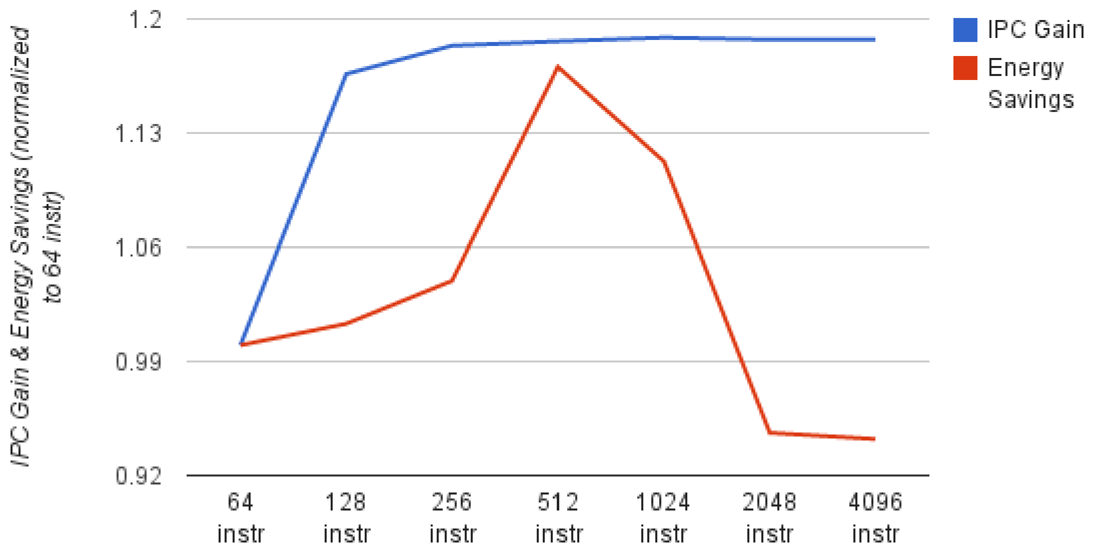


Figure 2.13: Sensitivity epoch size (512 best); avg IPC/energy oracle mapping.

6.1% more savings. However, the energy savings are not as high as astar because the varying branch misprediction rate causes more transitions between the cores, which is also reflected in the saturated counter and reactive schemes' energy declines. The trace-based system operates at 65.8% energy efficiency because it is based on backward branch PCs to define phases, and high branch misprediction rates cause the system to switch cores unnecessarily. This unnecessary switching also causes a 7% performance drop (Figure 3.8) relative to the saturated counter and reactive systems.

2.5.4 Sensitivity

In order to determine the optimal epoch size for fine-grained scheduling in our benchmarks, we tested multiple epoch sizings on the oracle resource mapper. Figure 2.13 shows the IPC and energy savings achieved by the oracle mapper for

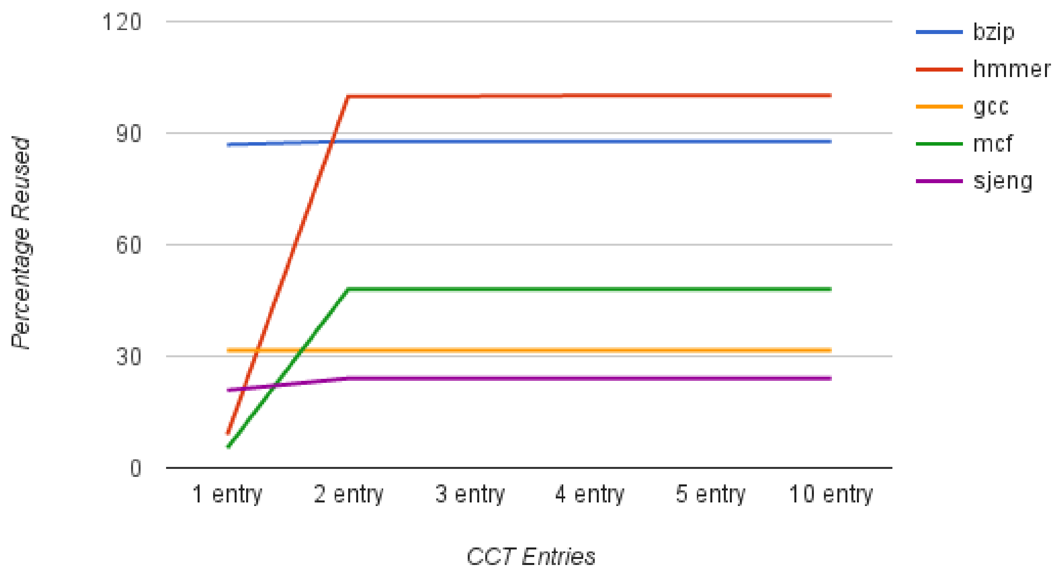


Figure 2.14: Sensitivity CCT size (5 best).

exponentially increasing numbers of instructions. The IPC remains relatively steady after 128 instructions, and the energy savings peaks at 512 instructions. As stated in Section 1.1, longer epoch sizes experience more uniform performance behavior, and experience less opportunities for energy savings. We use 512 instruction sized epochs in our system to obtain the best combination of IPC and energy savings.

To find the optimal number of CCT entries, and consequently PCT entries, we tested for how often the CCT entries were actively reused for increasingly more total CCT entries. A reused CCT entry means that the entry has an active countdown initiated, and that there are active pending chains with the same branch PC as the CCT entry. This indicates that a past chain of long-latency loads is again experiencing a CHILL phase currently. Figure 2.14 shows that most benchmarks experience the most reuse at 2 entries. We omit astar and libq because although they experience reuse of CCT entries, they also have high data locality. This leads them to experience past CHILL phases without long-latency loads because the data is being reused. Most of the gains in astar and libq come from our cold start policy for pending chains described in Section 2.3.5. Hmmer experiences its maximum reuse of 101% with 4 entries, indicating that sometimes more than one pending chain is correlated to one CCT entry. To accommodate hmmer, we use 5 CCT and 5 PCT entries in our system.

2.6 Conclusion

High performance variance in fine-grained program phases exposes opportunities for performance and energy savings, especially on tightly-coupled heterogeneous multi-cores. A particularly important fine-grained program phase occurs during bottleneck CHILL phases. These phases are troublesome because they produce long-latency loads, and clog the instruction window with other dependent shadow instructions. However, the shadow instructions present opportunities for phases of high ILP and performance. We have designed a system to track and predict bottleneck CHILL phases, map the phases to suitable cores, and achieve near oracle performance and energy savings.

CHAPTER 3

Limiting the Effects of Branch Impact Phases in Fine-Grained Mapping

When a conditional branch instruction is encountered during fetch and decode in a pipeline, the result of the condition is not determined until the execute stage [HP03]. This results in a potential performance bubble, where fetch needs to be stalled while the core awaits the resulting path of the branch. To potentially eliminate this performance bubble, chip architects speculate on the path of the branch with branch prediction hardware. The prediction hardware speculates on which path the branch will most likely take, and then speculatively fetches and executes instructions down that path, until the results of the branch's condition is known. Sometimes, this path speculation is incorrect, leading to a branch misprediction. Mispredictions are resolved by flushing the pipeline to eliminate the wrongly speculated branch path instructions.

Past works have noted that branch misprediction penalties can be high, and more recent works have also observed that such phases can be run on in-order cores [LPD12, PLD13]. These mispredictions can be particularly harmful when running on an OoO core because many speculative instructions could be issued into the pipeline. Upon the branch misprediction's pipeline flush, all of the speculative instructions are wasted. Mispredictions will happen regardless of the accuracy of the branch predictor, making this type of penalty unavoidable. However, we can utilize a fast-switching heterogeneous core to reduce the harm of these painful phases.

3.1 Related Work

Although certain prior work have noted in passing that branch mispredictions might be better handled on in-order cores [LPD12, PLD13], they do not distinguish between truly harmful mispredictions and ones with less impact. Our work will find the branch mispredictions with high impact, and run those on the in-order core. By only handling the high impact mispredictions, we avoid unnecessary switching of cores during mispredictions with low impact, and we save energy recovering from high impact mispredictions on the in-order core. DynaMOS [PLD15a] uses a technique where repetitive OoO schedules are memoized, and run on the in-order core for more energy efficiency. They target predictable branch behavior in order to find consistently repetitive instruction sequences, which creates an OoO-in-IO computational model. Our work will differ in that we will find the misbehaving branches, and minimize the negative effects of their faulty behavior on the in-order cores.

There is also an abundance of work which deal with reducing branch misprediction penalties by recovering the branch-independent instructions without re-fetching them [GAS04, CFS99, CV01, CTW04, KMS05, JAO15]. The DCI system [CFS99] attempts to dynamically find control-independent instructions, and execute them out-of-order during branch mispredictions, thereby overlapping useful computation with a branch misprediction. They accomplish this by using a shadow ROB, which records dependencies between newly issued instructions and branches. To find instructions fully independent of a branch path, they also record and use bitmasks of registers consumed by branch paths. In contrast, the Skipper system [CV01] does not execute any of the speculated branch path, and instead only executes control-independent instructions (after the speculated path) while a branch result is in progress. In the SBR system [GAS04], the authors try to elide branch mispredictions as much as possible. They track the convergence points of branches using an Alternate Target Buffer, where the addresses of the next sequential instruction of the block after a branch are kept. Upon a mispredicted branch, any faultily updated registers have their values restored to their pre-branch state with "mov" operations. This preserves the data dependences of the convergent path. Collins, et. al. [CTW04] attempt to make the prediction of branch reconvergence points more accurate by tracking reconvergence type data in at runtime. In Wish Branches [KMS05], the authors compile two different versions of branches: normal branch code, and predicate branch code (where conditions are turned in to data dependences). Their architecture uses misprediction rates to determine dynamically when to selectively use the predicated branch code. Finally, the Mower system [JAO15] attempts to reduce the branch misprediction penalty by directly reducing the overheads associated with flushing mispredicted instructions. This is done by walking the ROB to find mispredicted instructions, and associating them to register file, LSQ, reservation stations, and RAT entries. Essentially, the renaming of new convergent instructions is overlapped with flushing the ROB during the walk. In summary, the ideas of these works are based on dynamically discovering instructions independent of branches at runtime, and either preserving them in the pipeline so they do not need to be re-fetched after a misprediction, or speculatively executing them for performance benefits. While this type of prior work is complementary to ours, we will differ by running the high impact mispredictions on the in-order core, rather than optimizing execution on an OoO.

3.2 Branch Impact Motivation Statistics

The negative impact of branch mispredictions cannot be overstated. Mispredictions cause the speculative instructions to be flushed from the pipeline, thereby creating wasted cycles and instructions. On OoO cores, the number of flushed instructions due to mispredictions is naturally greater than that of in-order cores. OoOs have deeper pipelines built for such speculation, but suffer a greater penalty when the speculation is incorrect. Figure 3.1 shows that OoO cores experience a

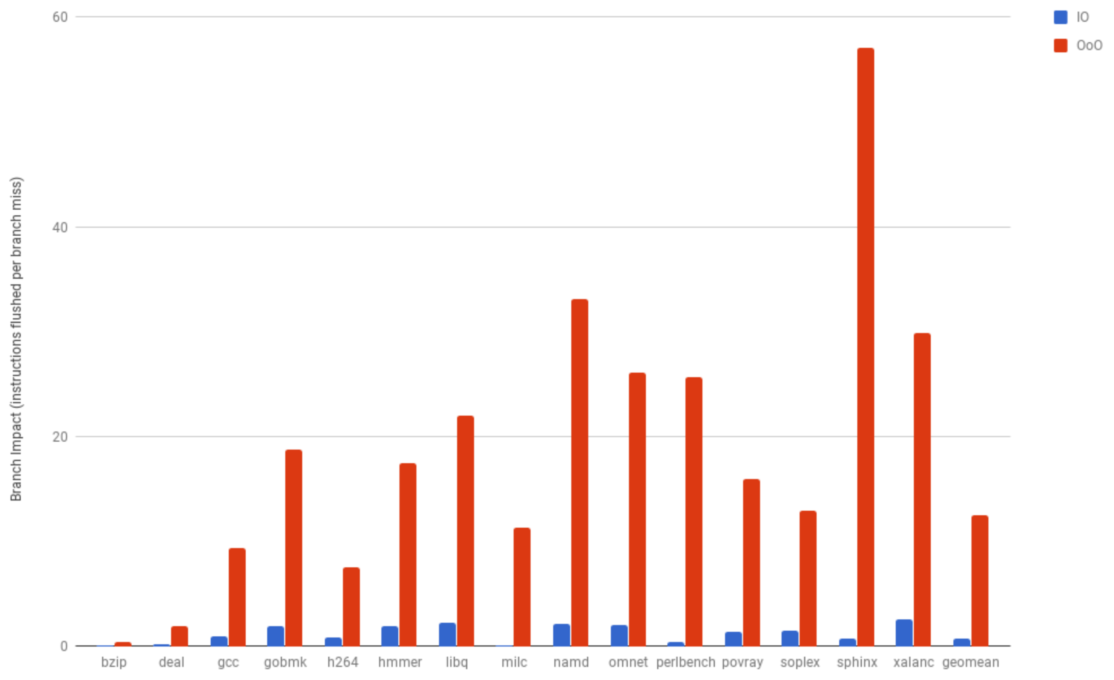


Figure 3.1: Instructions flushed per branch misprediction.

much larger flush penalty per branch misprediction. On average, the OoO core flushes 12.49 instructions during a misprediction, while the in-order core flushes 0.72 instructions. For every benchmark, the OoO waste is greater than in-order waste.

Over time, this waste on OoOs becomes more problematic. As more branches are encountered through the course of a program, more chances for mispredictions and consequent flushes will occur. Figure 3.2 illustrates the percentage of waste which occurs across entire benchmarks, relative to the total number of retired instructions. The OoO core experiences 27.29% waste, while the in-order core experiences 1.56% waste. Again, every benchmark naturally experiences more total percentage waste on the OoO than on the in-order core.

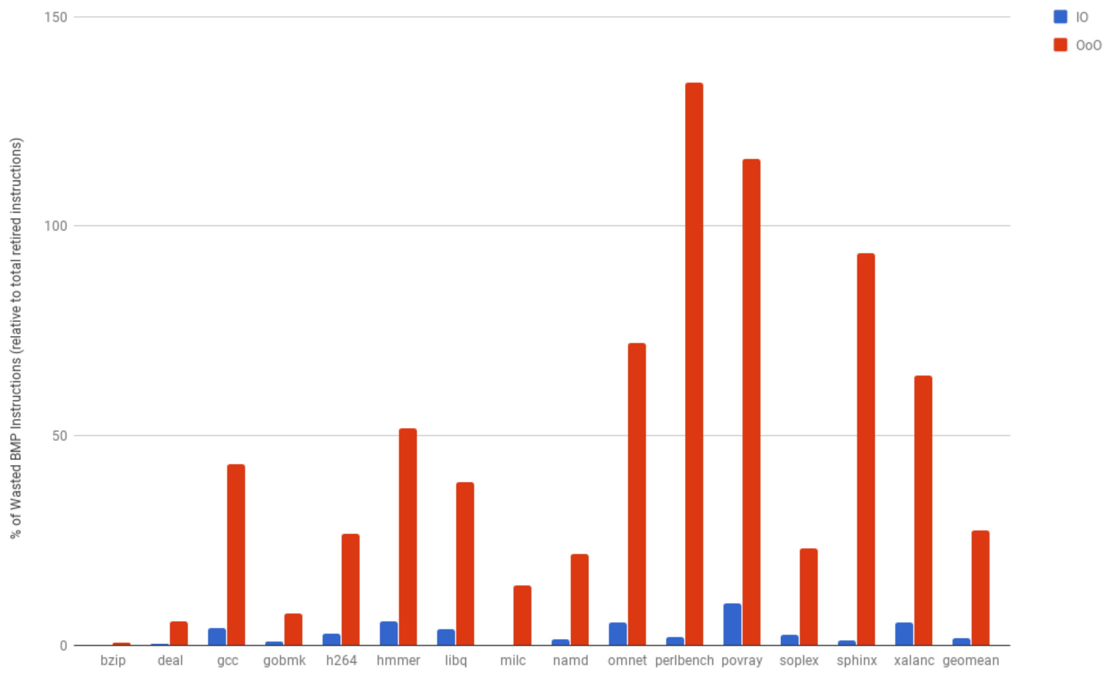


Figure 3.2: Percent wasted instructions (relative to total retired instructions).

The wasted instructions also translate to a certain percentage of wasted program cycles. Figure 3.3 shows the percentage of wasted cycles across entire benchmarks. The OoO core wastes 19.07% of all cycles during branch mispredictions on average, while the in-order core wastes 1.8%. These numbers coincide with the wasted instruction statistics in Figure 3.1 because most of these speculative instructions have only spent 1-2 cycles in the pipeline without retiring or committing.

To be clear, we present these statistics as an opportunity for improvement, rather than as an argument against OoO cores in general. The benefits of branch prediction have been made clear for the past three decades. When branch prediction is accurate, then the correctly speculated instructions provide a performance

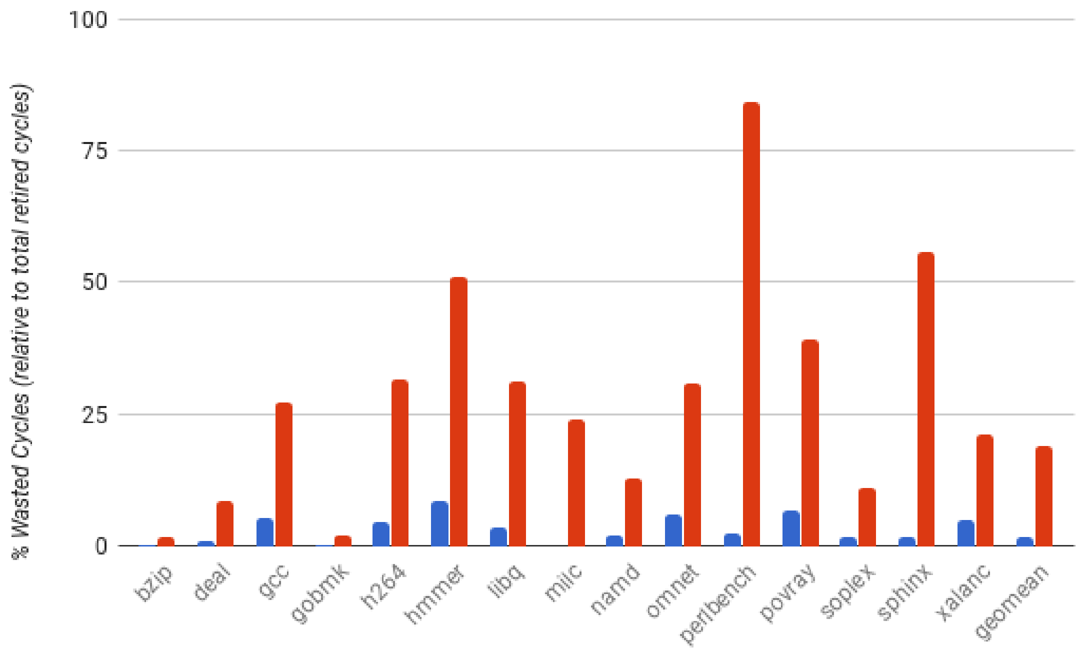
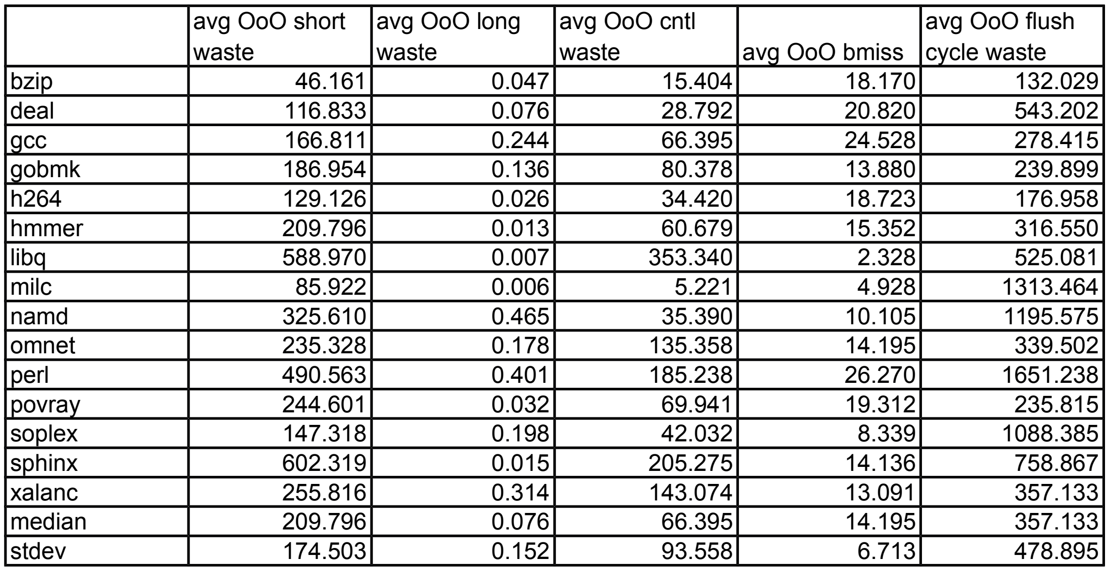


Figure 3.3: Total cycles wasted due to branch mispredictions.

boost. However, the penalties of branch mispredictions are still rather numerically significant, which presents an opportunity for optimization. Our work focuses on running branch misprediction phases on the in-order core, to reduce such penalties.

3.3 Methodology: Selection of Linear Regression

Our scheme to identify problematic program phases where branch mispredictions may occur, relies on using a linear regression to correlate speculated instructions from branch mispredictions, to cycles wasted. We hypothesized that certain types of instructions which are dependent on branches, would have various impact on wasted cycle time. For example, long latency loads and other control flow instruc-



	avg OoO short waste	avg OoO long waste	avg OoO cntl waste	avg OoO bmiss	avg OoO flush cycle waste
bzip	46.161	0.047	15.404	18.170	132.029
deal	116.833	0.076	28.792	20.820	543.202
gcc	166.811	0.244	66.395	24.528	278.415
gobmk	186.954	0.136	80.378	13.880	239.899
h264	129.126	0.026	34.420	18.723	176.958
hmmer	209.796	0.013	60.679	15.352	316.550
libq	588.970	0.007	353.340	2.328	525.081
milc	85.922	0.006	5.221	4.928	1313.464
namd	325.610	0.465	35.390	10.105	1195.575
omnet	235.328	0.178	135.358	14.195	339.502
perl	490.563	0.401	185.238	26.270	1651.238
povray	244.601	0.032	69.941	19.312	235.815
soplex	147.318	0.198	42.032	8.339	1088.385
sphinx	602.319	0.015	205.275	14.136	758.867
xalanc	255.816	0.314	143.074	13.091	357.133
median	209.796	0.076	66.395	14.195	357.133
stdev	174.503	0.152	93.558	6.713	478.895

Figure 3.4: Average out-of-order core waste per branch misprediction.

tions would be more negatively impacted if they were dependent on a mispredicted branch because more cycles would be spent on such instructions, and subsequently wasted due to a pipeline flush.

We detailed the breakdown of average numbers of speculative instruction types in Figure 3.4. "Short waste" instructions are integer and floating point operations. "Long waste" instructions are long latency loads, and L2 cache hits. "Control waste" instructions are other flushed branch instructions, but are dependent on another branch misprediction. We also tracked the total number of branch misses per scheduling epoch ("bmiss"). Finally, we correlated all of these statistics with the cycles wasted during branch mispredictions per epoch ("flush cycle") on the opposite (in-order) core type. In other words, we use the OoO metrics of branch waste, and correlate that to wasted cycles on the in-order core (not depicted), and

	avg OoO short waste	avg OoO long waste	avg OoO cntl waste	avg OoO bmiss	avg OoO flush cycle waste
bzip	46.161	0.047	15.404	18.170	132.029
deal	116.833	0.076	28.792	20.820	543.202
gcc	166.811		66.395		278.415
gobmk	186.954	0.136	80.378	13.880	239.899
h264	129.126	0.026	34.420	18.723	176.958
hmmer	209.796	0.012	60.679	15.352	316.550
libq		0.007			525.081
milc	85.922	0.006	5.221		
namd	325.610		35.390	10.105	
omnet	235.328	0.178	135.358	14.195	339.502
perl					
povray	244.601	0.032	69.941	19.312	235.815
soplex	147.318	0.198	42.032	8.339	
sphinx		0.015		14.136	758.867
xalanc	255.816		143.074	13.092	357.133

Figure 3.5: Selected linear regression baseline benchmarks after elimination (in bold).

vice versa. Note that Figure 3.4 presents average statistics per epoch for the OoO core.

Our linear regression modeling takes per epoch breakdowns of mispredicted instruction flushes (like the statistics in Figure 3.4 but not averaged across all epochs), and correlates those types of waste to the actual branch misprediction cycles wasted on the other core type. However, because every benchmark is different, we pared down the number of benchmarks used to create the linear regression. To do this, we observed the median of every category of waste, and eliminated from consideration every benchmark which did not fall within one standard deviation. In the end, we only selected benchmarks for the linear regression which experienced no eliminations in any waste categories (Figure 3.5). The selection process only considers the OoO statistics because that is where the majority of

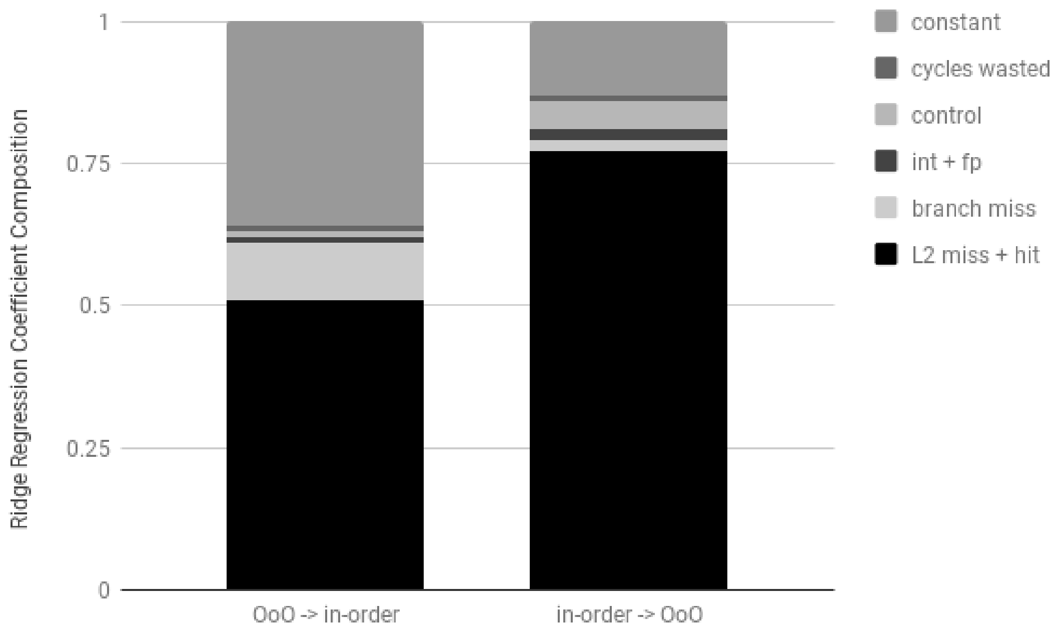


Figure 3.6: Branch impact linear ridge regression composition.

the waste occurs. When performing the in-order -> OoO regression, we use the in-order statistics and correlate them with OoO waste, but they are not used in the selection process because the waste is mostly too small to be useful. Using this selection process, we were able to train the linear regression model on the selected benchmarks, and apply the trained regression to all benchmarks. We train on a series of 2 million instructions.

The training process generates the factors and constants of the linear regression. Figure 3.6 shows the weight each feature contributes to the regression, for each prediction mode (OoO -> in-order, and in-order -> OoO). Long events, like L2 misses and hits, contribute significantly to the model. This makes sense because L2 misses and hits require significant execution time, and often have several

$$CPI_{in-orderbranchimpact} = \sum CPI_{observedbranchimpact} + \alpha \sum CPI_{pasterror} + \beta CPI_{currenterror} \quad (3.1)$$

$$\Delta CPI_{OoOdecision} = \sum CPI_{observed} + \alpha \sum CPI_{pasterror} + \beta CPI_{currenterror} - CPI_{in-orderestimated} - CPI_{in-orderbranchimpact} \quad (3.2)$$

instructions dependent upon them. When L2 misses and hits are speculated, they bring many of their dependent instructions speculatively into the pipeline, making their impact on a misprediction weighty. The regression breakdown also shows other branch misses and control instructions to be impactful. This is because they too bring more speculated instructions into the pipeline, making a potential misprediction more hurtful to the program's performance.

The linear regression we chose is calculated at runtime to be a cycles per instruction performance metric. Equation 3.1 shows the linear regression created to measure the negative impact of branch mispredictions, with the corresponding proportional-integral controller. $CPI_{currenterror}$ represents the proportional part of the controller, accounting for error in waste estimation during the current epoch. $CPI_{pasterror}$ represents the integral part of the controller, which is a summation of the error from the past 5 epochs. Both alpha and beta are determined experimentally.

To make a fine-grained core decision while running on the OoO core, Equation 3.2 shows the process. The original estimation from Composition Cores is used, but with the $CPI_{in-orderbranchimpact}$ from Equation 3.1 factored as a part of the calculation. Estimated branch impact wasted cycles are subtracted from the decision calculation, in order to add negative weight to the original performance estimate of the in-order core. If the resulting $CPI_{OoOdecision}$ is positive, meaning that the cycles spent on the OoO core is greater than the in-order core, then decision is made to switch to the in-order. While running on the in-order core, the same type of calculation is made, but with opposite estimates.

Note that in general, not all branch misprediction program phases will be mapped to the in-order core. Sometimes a branch misprediction phase can still be run on the OoO, if its negative impact is not high enough to cross the threshold set by the linear regression. This is the purpose of the selection process (Figure 3.5), to select benchmarks fairly, such that branch misprediction phases with high impact are captured. If we attempted to run all branch misprediction phases on the in-order core, we would miss opportunities for correct speculation that are helpful to program performance.

3.4 Architectural Modifications

Figure 3.7 shows the tightly-coupled heterogeneous multi-core architecture we simulate. The architecture is similar to Composite Cores, in that there is one core with two decode and execution backends, one each of an OoO engine and in-order

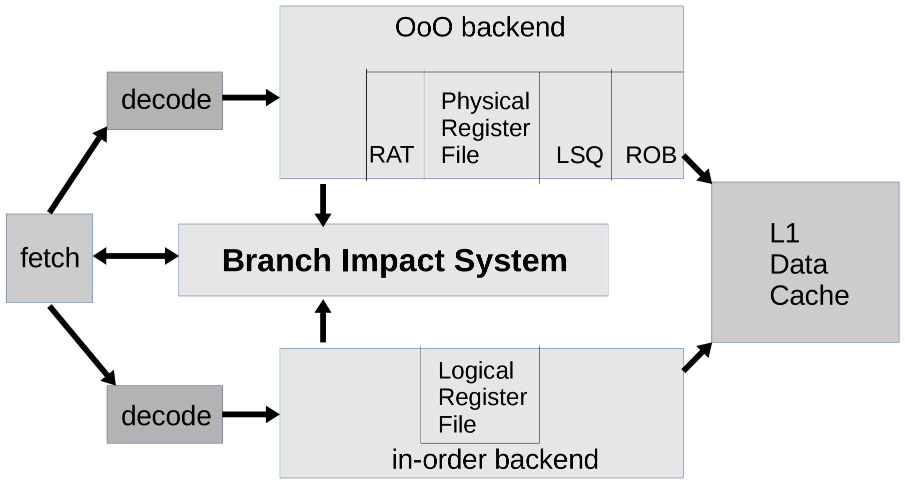


Figure 3.7: Architectural overview of branch impact in relation to heterogeneous multi-core components.

engine. The branch impact decision making system features the original metrics of Composite Core performance estimates, but also adds the branch impact waste estimation. Fetch, cache, and the decision making system are all shared by both exeuction backends. Also shared is the register and core state transfer system described in Composite Cores [LPD12].

3.5 Simulation Methodology

Our simulation parameters match ARM's big.LITTLE [Gre12] as described in 2.4. The big OoO core is modeled after the Cortex-A15, which is 3-way issue with a pipeline depth of 15-25 stages. The LITTLE in-order core is modeled after the Cortex-A7, which has a shorter pipeline of 8-10 stages with 2-way issue capability, depending on instruction dependencies. Each of these cores correlates to their respective backends from Figure 3.7. We model power savings as clock gating each of the backends. This sort of static power model is in line with current technology, although greater savings could be produced if we had used power gating.

We run benchmarks from SPEC 2006 [Cor06] in the Simics [MCE02] and GEMS [MSB05] simulators. Core power and energy are modeled in McPAT [LAS09]. Table 2.1 provides more details on the architectural parameters simulated. All simulations execute over 15 million instructions, with 512-instruction epoch size, and compiled for SPARC v9 with -O3 optimization.

The branch predictor used is a YAGS (yet another g-share) predictor [ET98]. We recognize the fact that the choice of branch predictor can determine how many mispredictions may occur. A bad branch predictor will incur more mispredictions, while a strong one will incur fewer, thereby possibly skewing our branch impact results. To our knowledge, YAGS is still the basis for many modern branch prediction schemes in industry, which affirms our choice. Regardless, no branch predictor is perfect, which means that branch impact estimation is relevant irrespective of branch predictor choice.

The GEMS simulator also handles branch mispredictions in a fair manner. It flushes the entire pipeline upon a misprediction, which is what is expected in industry chips. There exists work which attempts to constrain the flushing to only speculated instructions (mentioned in Section 3.1). However, we are not aware of industry-wide adaptation of such techniques, and we do not generate a comparison with them. We also note that the branch impact instructions counted are only speculated instructions, and not all flushed instructions.

3.6 Results

We compared our scheme against Composite Cores and a throttled branch prediction OoO core. Our scheme was implemented as an addition to Composite Cores, so their work is a natural point of comparison. Because our work observes that OoO cores experience rather heavy misprediction penalties, a natural solution would be to throttle speculation during branches, which would reduce any misprediction penalty while not switching cores. The throttling scheme we chose imposed a 45 instruction limit on the instruction window while any unresolved conditional branch was in the pipeline. We chose a limit of 45 because our in-order core has an instruction window of 15, and the OoO core has 3-way issue and fetch stages. This turns out to be roughly one third of the original instruction window size of the OoO core. Note that the branch throttled OoO is also implemented on top of a Composite Core baseline, meaning that the in-order core can still be used, and throttling only occurs on the OoO execution phases.

In terms of performance, our scheme achieves 0.4% more IPC than Composite Cores, and 21.6% more IPC than a branch throttle OoO (Figure 3.8). It is reasonable for our scheme to perform relatively the same as Composite Cores because we are executing more frequently on the in-order core to reduce waste on the OoO.

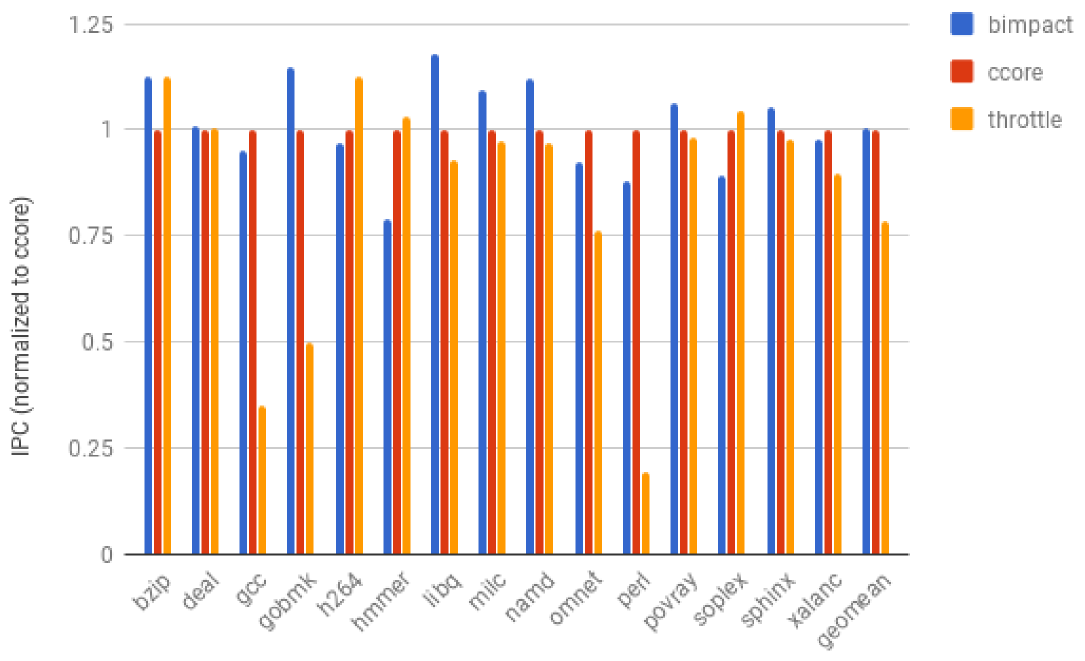


Figure 3.8: IPC normalized to Composite Cores.

The in-order core invariably has worse overall performance, which offsets the performance wasted from branch mispredictions on the OoO. Both schemes outdo the branch throttled OoO because the OoO experiences the branch misprediction waste.

The gains of our scheme come mostly from energy savings obtained by running more frequently on the in-order core. Overall, our scheme saves 26.2% more energy than Composite Cores, and only uses 35.55% of the energy that a branch throttled OoO core uses (Figure 3.9). In general, this result is intuitive because our scheme finds more opportunities for in-order core usage, while reducing wasted time spent on the OoO core. While the inherent performance of the in-order is worse than the OoO, more energy is saved during in-order exeuction. The branch throttled OoO

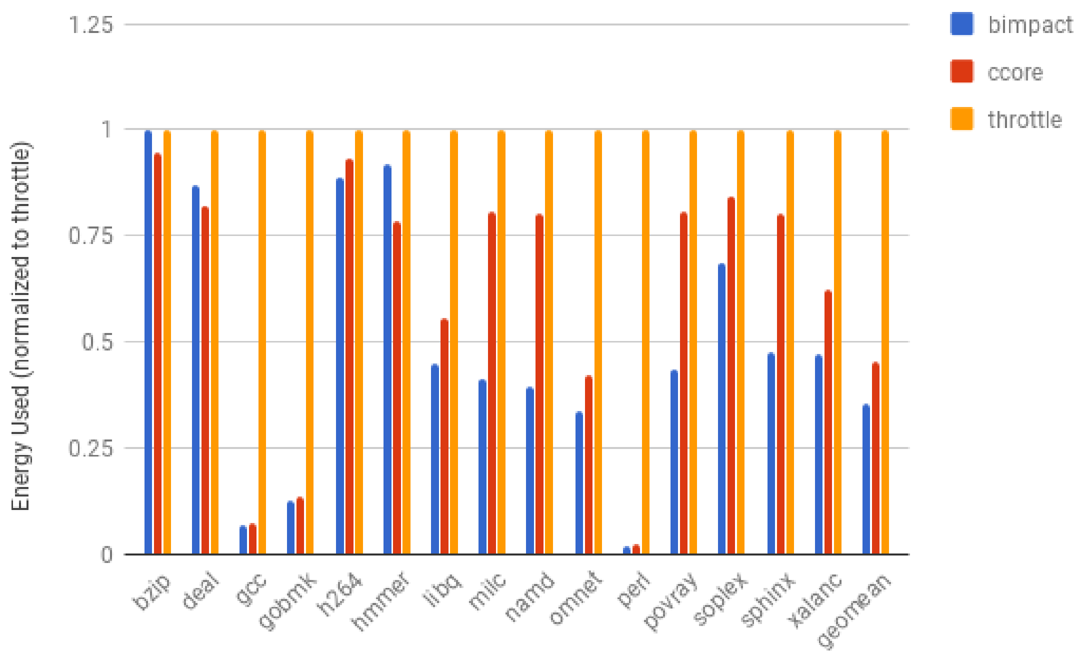


Figure 3.9: Energy consumed, normalized to throttled out-of-order.

would naturally experience the most energy consumption because it experiences the branch misprediction waste while running on the power hungry OoO. While the branch throttled OoO certainly experiences a reduced quantity of branch misprediction waste, for several benchmarks the level of waste is still too high for the throttling to overcome. The throttling also limits the effectiveness of the OoO, causing longer execution while consuming high OoO power.

The hmmer benchmark is different in that it exhibits 21% worse performance and consumes 17% more energy than Composite Cores. This is because hmmer experiences nearly all of its program phases with at least one branch misprediction event. Its branch impact composition (Figure 3.4) is also closest to the median of all the benchmarks, while experiencing variance nearly as much as the standard

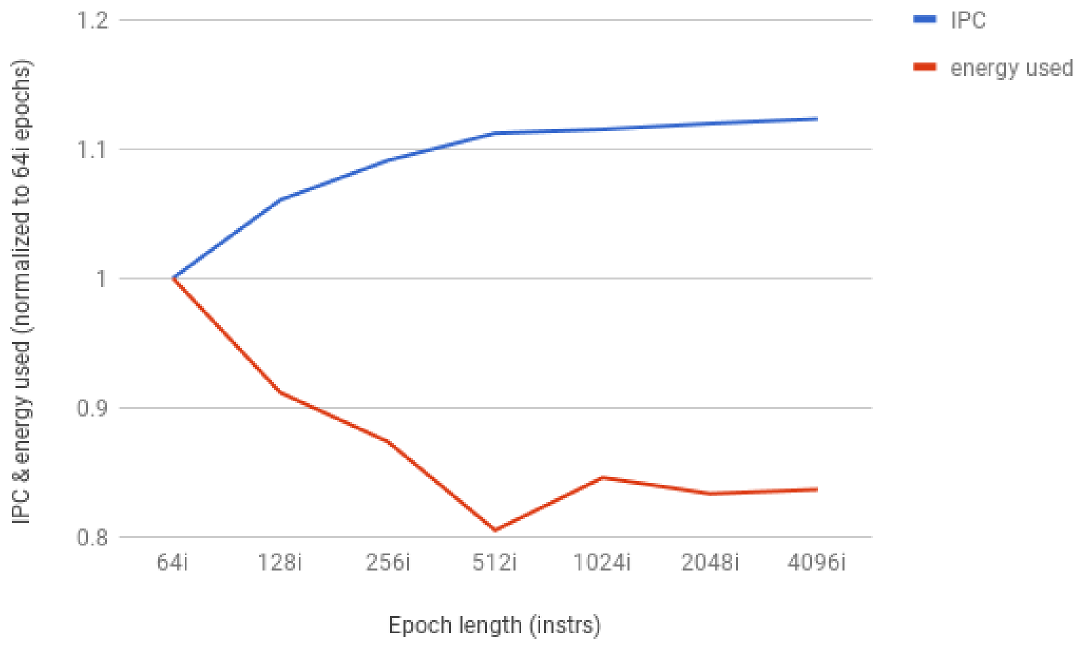


Figure 3.10: Branch impact sensitivity analysis of epoch size on performance and energy consumption.

deviations. Hence, hmmer flip flops constantly between the OoO and in-order cores, causing inefficiency in both performance and energy consumption.

A few other data points exhibit decreases in performance, but have reasonable levels of energy savings. Omnet shows 8% decrease in performance, but also allows for 20.9% more energy savings. Perlbench has a 12.3% decrease in performance, but experiences 14.7% energy savings, which is close to a 1-to-1 tradeoff. Finally, soplex loses 11.1% performance, but redeems this loss with 18.5% energy savings.

Branch impact is best exploited in a fine-grained manner. Figure 3.10 shows sensitivity analyses of the effects of various epoch sizes on performance and energy consumption. As epoch length increases, performance levels off after 512 instruction length epochs. Smaller epochs experience a bit worse performance because there is more backend switching activity, causing more time to be spent in transition and slowing the program down. This effect is mitigated somewhat at larger epochs, but epochs beyond 512 instructions are less able to take advantage of fine-grained behavior. Energy consumption encounters a trough at 512 instructions per epoch, but increases afterward. This is because larger epoch sizes are also less able to take advantage of fine-grained behavior. Again, small epoch sizes (below 512) consume more energy because they encounter increased switching frequency.

3.7 Conclusion

In general, we found that the negative impact of branch mispredictions can be mitigated on tightly coupled heterogeneous general-purpose multi-cores. We sorted out highly impactful branch misprediction phases with a linear regression, and ran these phases on the in-order backend to mitigate mis-speculation penalties. Our system matches the performance of prior work, but reduces the energy consumed by avoiding mis-speculation performance losses.

CHAPTER 4

Analyzing the Benefits of Early Scheduled Instructions on Out-of-Order Cores

Determining whether a program phase is suitable for OoO execution can be challenging. In Composite Cores [LPD12], the authors use ILP and MLP metrics while running on the OoO engine. ILP was defined as the number of instructions which stalled in the issue queue, as an inverse metric of ILP. MLP was defined as the number of MSHR entries used, representing the number of last level cache misses. While running on the in-order core, ILP and MLP were measured with a table based dependence tracking mechanism adapted from Chen, Dropsho, and Albonesi [CDA03]. The metrics for ILP and MLP obtained on the in-order core require some processing time to analyze, while the metrics on the OoO may not be adequate enough to determine OoO suitability. Our work seeks to obtain a more accurate picture of OoO core suitability with a different set of metrics.

4.1 Motivation

As instructions are fetched on processors, they are issued or scheduled for execution, typically in program order on in-order cores. OoO cores have mechanisms whereby instructions can be scheduled out of program order. Usually, this occurs because an older instruction is waiting on data, but younger instructions have their source operands fulfilled before an older instruction. Often, the older instruction in this situation is an LLL, or an instruction dependent on an LLL, both of which need to wait for memory to supply data. This leads to some younger instructions being scheduled earlier than some older instructions, which we call "early scheduled instructions" (ESI).

The presence of ESI is direct evidence that an OoO core's resources are being utilized. More ESI in an OoO core indicate that the OoO resources are more likely to be used efficiently. This implies that the program phase requires the OoO resources, and should continue to run on the OoO core. Figure 4.1 shows binned average ESI counts, categorized by different performance ranges. The blue bars represent benchmark epochs with the lowest IPC (< 0.5); red bars represent a medium IPC (0.5 - 1); and yellow bars represent the best IPC (1 - 2). Each bar is normalized the the lowest IPC bin for its respective benchmark. The highest IPC bin of 1 - 2 observes 19% more ESI count than the lowest bin, while the medium bin exhibits 3.4% more ESI than the lowest bin. Note that the IPC of the in-order core for these benchmarks is below 0.5 for all epochs. This means

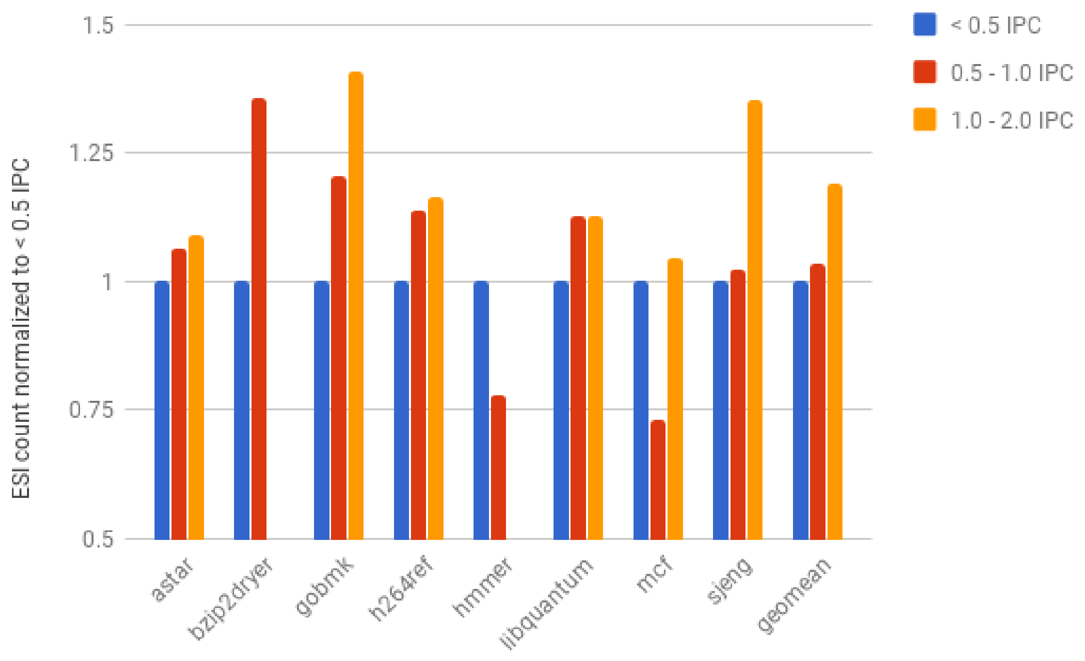


Figure 4.1: Greater average ESI per better IPC bin (bin sizes: 0 - 0.5, 0.5 - 1, 1 - 2); normalized to each benchmark's "less than 0.5" bin

that in order to obtain better performance on these benchmarks, certain phases need to be run on the OoO core. On average, a better IPC is represented by a greater ESI count.

A couple benchmarks show lower ESI counts in their medium bins, when compared their lowest bins. Hmmer and mcf have roughly 25% less ESI during their medium IPC phases. This is due to these benchmarks having several long latency loads, and consequent dependent instructions, in their low IPC phases. These phases have naturally low IPCs because they wait for long latency loads to complete. Yet, they have abnormally high ESI (in comparison to the rest of their own program phases) because the pipeline fills with instructions dependent on

	average short ESI	average long ESI	average cntl ESI	average IO cycle difference
astar	221.870	1.739	72.463	-687.452
bzip2dryer	157.268	0.160	56.309	-435.327
gobmk	182.367	0.204	99.869	-3303.261
h264ref	175.938	0.262	45.675	-453.249
hmmer	203.994	0.160	61.557	-506.439
libquantum	309.281	0.417	88.875	-933.541
mcf	166.286	33.756	107.961	-2095.241
sjeng	198.777	5.619	73.190	-360.189
median	190.572	0.340	72.827	-596.945
stdev	48.195	11.653	21.689	-1055.760

Figure 4.2: Average ESIs for different types of instructions, with corresponding in-order decrease in performance; training selected from benchmarks fitting in one standard deviation from median in all four statistical categories

the long latency loads. Regardless, the general trend of these benchmarks show that higher ESI generally corresponds to better performance on an OoO core.

Our work seeks to find program phases where ESI count is high enough to indicate that running the phase on the OoO core will be profitable. Although we have discussed various methods for scheduling in Section 2.2.2, to the best of our knowledge, this is the first use of an ESI metric to determine program phase suitability for an OoO core.

4.2 Selecting Linear Regression Training Data

In order to take advantage of the ESI trend, we need to project it onto in-order core performance. This helps the system determine whether to switch to the in-order core if abnormally low ESI is observed, or to stay on the OoO core if high

	average short ESI	average long ESI	average cntl ESI	average IO cycle difference
astar	221.870	1.739	72.463	-687.452
bzip2dryer	157.268	0.160	56.309	-435.327
gobmk	182.367		99.869	
h264ref	175.938		45.675	-453.249
hmmer	203.994	0.160	61.557	-506.439
libquantum		0.417	88.875	-933.541
mcf	166.286			
sjeng	198.777	5.619	73.190	-360.189

Figure 4.3: Benchmarks chosen (in bold) from fitting in one standard deviation from median in all four statistical categories

ESI is anticipated. To do this, we obtained a breakdown of ESI components, and correlated them with the difference in performance between the OoO and in-order cores. The differences in performance were obtained from traces of benchmarks run on pure OoO and pure in-order core simulations. Figure 4.2 shows the average ESI breakdowns and their corresponding in-order core performance loss for each benchmark.

The breakdowns of ESI in Figures 4.2 and 4.3 were chosen by instruction type. "Short ESI" represents integer and floating point operations. "Long ESI" are L2 hits and misses. "Cntl ESI" represent any control instructions. The in-order cycle differences are negative because they represent cycle loss compared to an OoO core.

To obtain a hardware-friendly ESI prediction mechanism, we chose to use a linear regression which gathers the ESI breakdown features, and correlates them to in-order core performance loss at runtime. First, we need to pare down the set of benchmarks to find a subset of benchmarks as a representative training set for

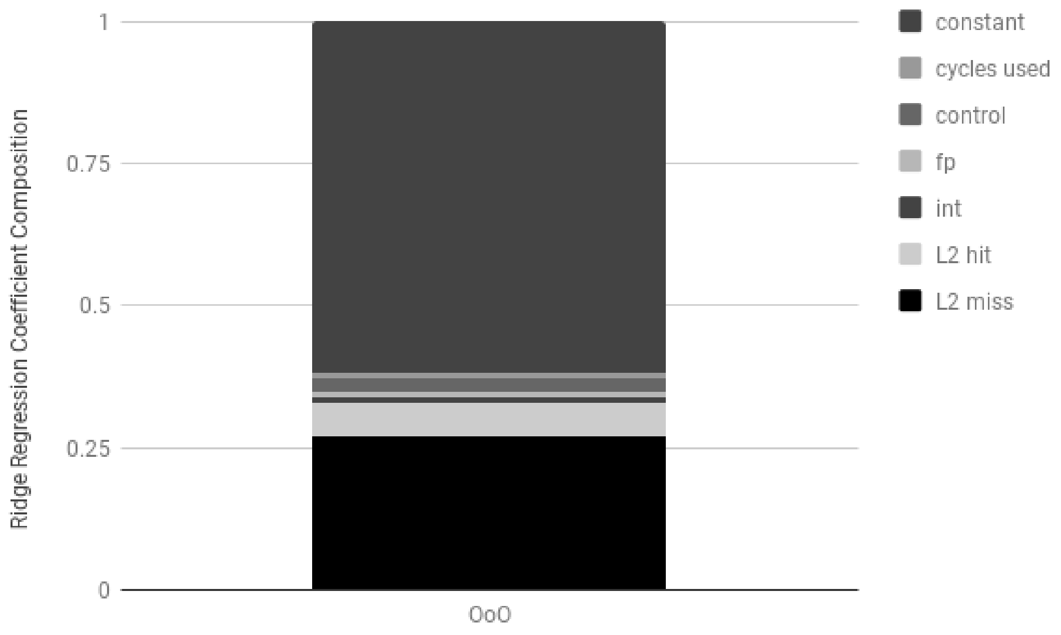


Figure 4.4: ESI linear ridge regression composition.

generating linear regression constants and coefficients. We did this by using the features in Figure 4.2 and eliminating any benchmarks which do not fully fit within one standard deviation of the median for each feature. Consequently, Figure 4.3 shows the result of such an elimination process, with the bolded benchmarks chosen as the selected linear regression training set. From there, we derived the linear regression constants required to represent this set of benchmarks.

The composition of the linear regression coefficient weights is shown in Figure 4.4. L2 misses and hits incur the most weight, which is sensible because they are long latency events which carry other instruction dependent on them into being potentially scheduled late or early. The next highest weighted are control instructions, which also makes sense in that control speculation also carries more

$$CPI_{in-orderloss} = \sum CPI_{observedESI} + \alpha \sum CPI_{pasterror} + \beta CPI_{currenterror} \quad (4.1)$$

$$CPI_{OoOdecision} : if\ CPI_{in-orderloss}\ negative,\ then\ remain\ on\ OoO,\ else\ switch\ to\ in-order \quad (4.2)$$

dependent instructions that could be scheduled early. Note that ESI can only be gathered while running on the OoO core, which means that this type of decision can only be made while currently active on the OoO backend. While running on the in-order core, we revert to the Composite Cores method of ILP and MLP estimation based on tables of dependence analysis.

Equation 4.1 shows the linear regression based on ESI breakdown features, with their proportional-integral controller. It estimates the in-order core performance, relative to the current OoO core's performance. If the estimate is negative, then the in-order core will produce a performance loss, and the next phase will remain on the OoO (4.2). In essence, we replace the normal Composite Cores estimation with ESI's in-order loss estimation, while running on the OoO core.

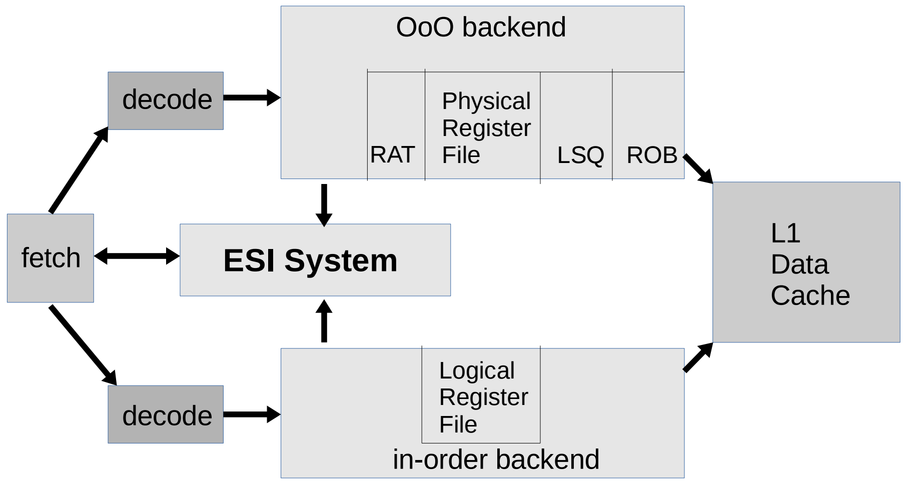


Figure 4.5: Architectural overview of ESI in relation to heterogeneous multi-core components.

4.3 ESI Architectural Modifications

The ESI system (Figure 4.5) compares similarly to Composite Cores. It features a tightly coupled core with separate OoO and in-order execution backends. The shared architectural features are the fetch unit, cache, the decision making system where ESI resides, and the register and core state transfer mechanism [LPD12]. While running on the OoO backend, ESI decisions are made, and while running on the in-order backend, the Composite Core techniques are used to make core choices.

4.4 ESI Simulation Methodology

We emulate an ARM big.LITTLE core [Gre12], with big and LITTLE cores acting as our tightly coupled backend engines. The big OoO core is modeled after the Cortex-A15, which is 3-way issue with a pipeline depth of 15-25 stages. The LITTLE in-order core is modeled after the Cortex-A7, which has a shorter pipeline of 8-10 stages with 2-way issue capability, depending on instruction dependencies.

Our simulations are run on the same software as described in Section 2.4. We maintain the same backend migration mechanism. Again, clock gating is used to model core switching power savings, which is both realistic and pessimistic.

We run benchmarks from SPEC 2006 [Cor06] in the Simics [MCE02] and GEMS [MSB05] simulators. Core power and energy are modeled in McPAT [LAS09]. Table 2.1 provides more details on the architectural parameters simulated. All simulations execute over 15 million instructions, with 512-instruction epoch size, and compiled for SPARC v9 with -O3 optimization.

4.5 Results

Overall, the ESI system shows 14.5% performance improvement over the Composite Cores system. As seen in Figure 4.6, none of the benchmarks suffer worse performance. This is expected because the system chooses the OoO core more often, which increases the chances of obtaining higher IPC.

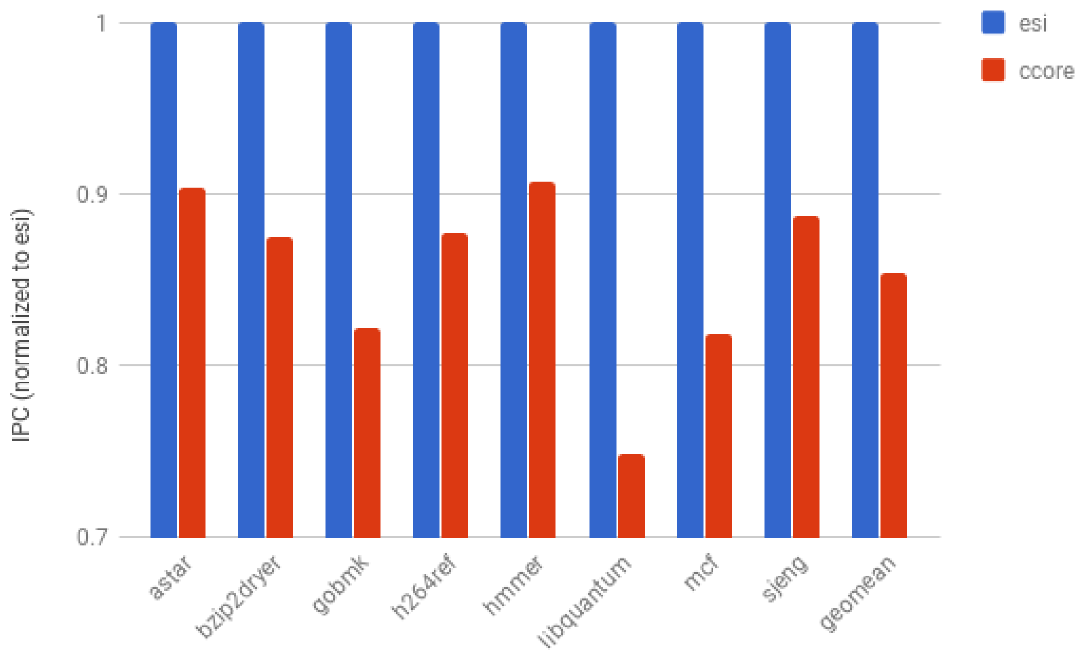


Figure 4.6: ESI exhibits 14.5% more IPC than Composite Cores

The increase in performance comes at the cost of increased energy usage, due to running more often on the power hungry OoO core. Figure 4.7 shows an average of 2.7% increase in energy consumption from the ESI system. Most benchmarks exhibit increased energy usage, but do not exceed 11% more energy consumption than Composite Cores (hmmer). Libquantum and mcf manage to save energy because running they spend more time during high IPC phases, which reduces the overall run time of the program. This reduction in run time is large enough that the benchmarks experience reduced energy consumption.

Both astar and hmmer show roughly 11% and 12% respective increases in energy expenditures, which can be problematic. However, they each receive 9% and 8% respective performance boosts. This means that for every 1% performance

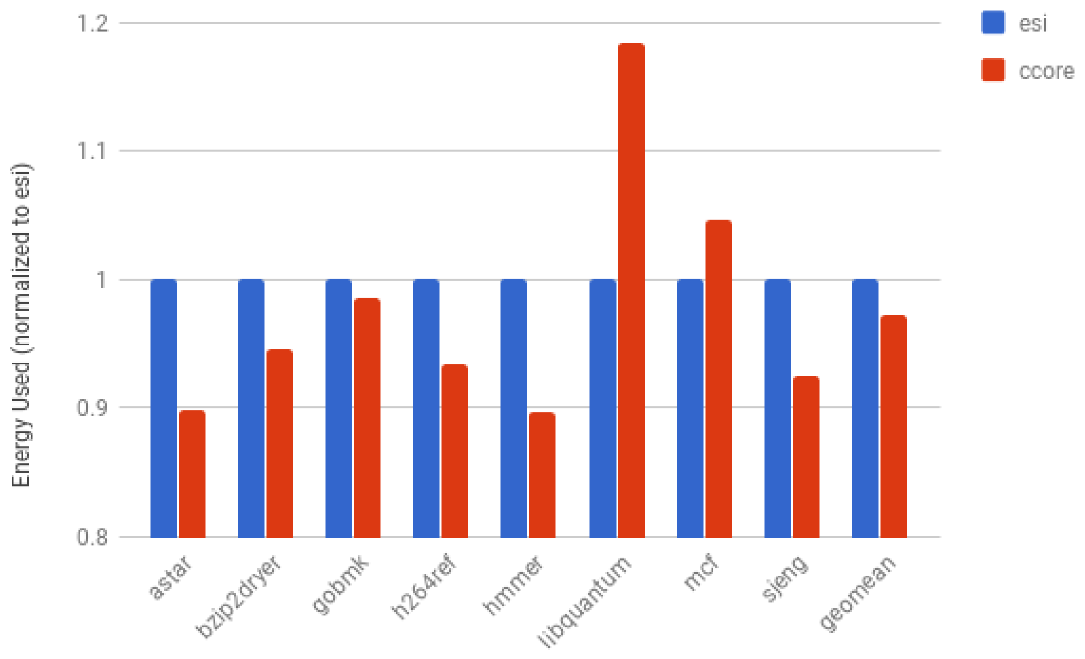


Figure 4.7: ESI uses 2.7% more energy than Composite Cores

boost, they use a little over 1% more energy, which is roughly a fair tradeoff. The other benchmarks all experience better performance/energy tradeoffs.

Finally, we show that ESI operates better in a fine-grained mapping scheme with sensitivity analyses in Figure 4.8. Epochs of 512 instructions receive the best performance boost, with larger epoch sizes not performing much better. The smaller epoch sizes, under 512 instructions, incur more backend switching activity which decreases their performance due to overhead. Similarly, energy consumption does not become significantly better for epoch sizes greater than 512 instructions.

Although 1024 instruction epochs experience about 3% better energy savings, it also experiences a 3% decrease in performance. Epoch sizes larger than 1024 be-

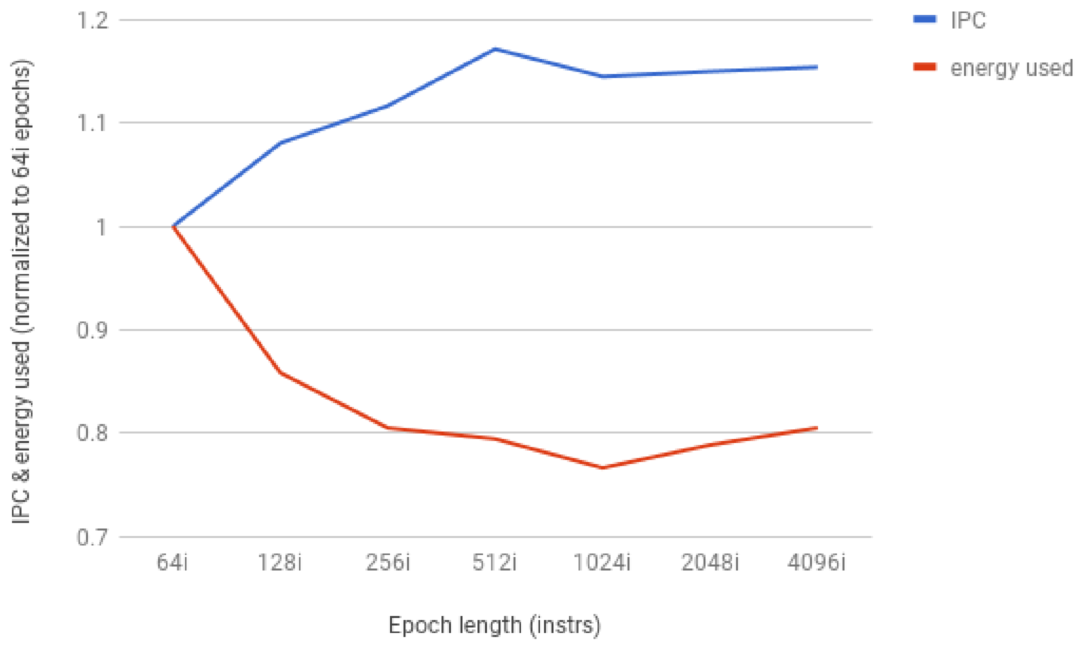


Figure 4.8: ESI sensitivity analysis of epoch size on performance and energy consumption.

gin to show a trend of increasing energy consumption, which represents the larger granularities' inability to take advantage of fine-grained behavior. Smaller epoch sizes consume more energy than 512 instruction epochs because more switching activity creates enough overhead to prolong execution time.

4.6 Conclusion

We have shown that ESI counts can be effective at determining core choice in fine-grained tightly coupled heterogeneous general-purpose multi-core resource mapping. ESI is a direct measure of the suitability of program phases for OoO cores, which is able to obtain a reasonable performance gain at a small energy consumption penalty. Although the ESI metric is only obtainable on the OoO core, it still presents a unique opportunity for further resource mapping optimization.

CHAPTER 5

Combining Linear Regression Techniques for Fine-Grained Mapping

Finally, we combine both the branch impact and ESI linear regression techniques into one system. We exclude the CHILL system from the combination framework because the decisions made with CHILL were absolute. If a CHILL phase were detected, then the Composite Cores linear regression would be skipped, and hence, any other linear regression method would be skipped. As a result, the branch impact and ESI regressions should be compared more accurately with themselves and Composite Cores.

5.1 Regression Modifications

We use both the branch impact and ESI linear regression models derived in previous Chapters. While running on the in-order backend, we use Composite Core OoO performance estimation, in addition to OoO branch impact estimation (Equation 5.1). This is similar to the technique originally used when running on the in-order backend from the branch impact chapter (Section 3).

$$\Delta CPI_{in-orderdecision} = \sum CPI_{observed} + \alpha \sum CPI_{pasterror} + \beta CPI_{currenterror} - CPI_{OoOestimated} - CPI_{OoObranchimpact} \quad (5.1)$$

$$CPI_{in-orderloss} = \sum CPI_{observedESI} + \alpha \sum CPI_{pasterror} + \beta CPI_{currenterror} - CPI_{in-orderbranchimpact} \quad (5.2)$$

If the core is running on the OoO backend, a combination of the ESI and branch impact estimations is used. Equation 5.2 shows the ESI estimation, with branch impact wasted performance factored into the decision-making process. Branch impact is subtracted to represent further performance loss, with a negative $CPI_{in-orderloss}$ still representing a decision to remain on the OoO backend.

The architecture used is the same as Figure 4.5, but with branch impact linear regression logic added. Our simulation parameters remain the same as Section 2.4. The assumptions about branch predictor choice and GEMS simulation limitations from Section 3.5 also remain the same.

5.2 Results

Overall, the combination of ESI and branch impact linear regressions produces an average of 4.7% performance loss in comparison to Composite Cores (Figure 5.1). This represents a push and pull effect of the different goals of each system. ESI

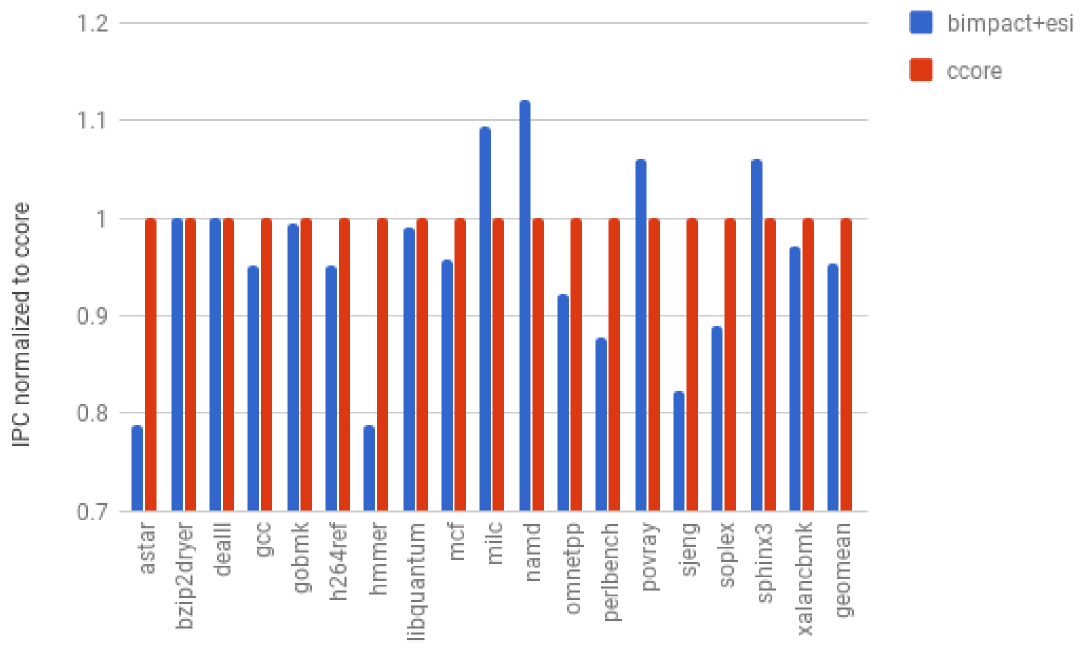


Figure 5.1: Combination IPC within 4.7% of Conservation Cores.

seeks to keep the program on the OoO, while branch impact favors the in-order backend. As a result, there are inevitable inefficiencies in such a system with two opposing ends.

The worst performing benchmarks in the combination scheme are astar and hmmer at roughly a loss of 21% IPC each. Hmmer still straddles the the line of the branch impact linear regression, in that it is the benchmark that is exactly the median. Therefore, it is overly sensitive to flip-flopping across the OoO and in-order backends. Astar experiences an abundance of branch impact phases, and thus over-adjusts to them by running more often on the in-order core. This tendency incurs a performance loss because the in-order core is naturally slower, and consequently a 22 % energy loss as well (Figure 5.2). Sjeng also experiences

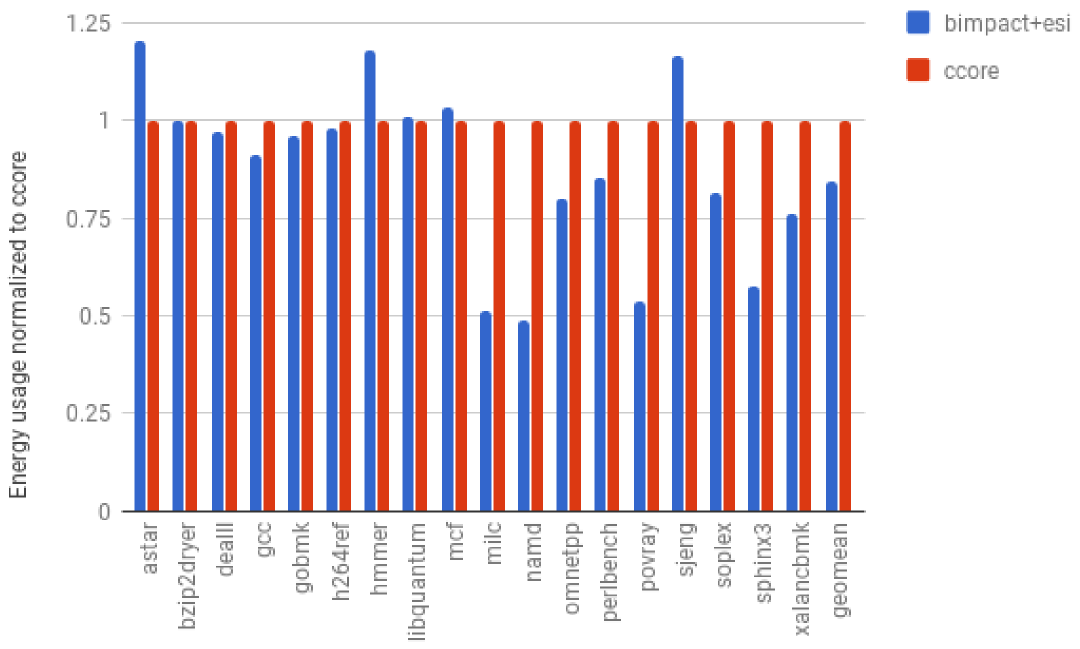


Figure 5.2: Combination saves 15.8% energy compared to Conservation Cores.

an 18% performance loss because it too favors the in-order core more frequently. It too consumes about 19% more energy due to this loss in performance.

In general, the combination scheme produces a 15.8% energy savings on average. Several benchmarks experience small or no energy savings: bzip, deal, gcc, gobmk, h264, libq, and mcf. Gcc, h264, and mcf experience 5% drops in performance, but the other benchmarks experience no drops in performance. This is evidence of the push and pull factor of competing goals in ESI and branch impact evening out for relatively no gains.

On the other hand, several benchmarks experience energy savings: milc, namd, omnet, perl, povray, soplex, sphinx, and xalanc. These benchmarks encounter various degrees of branch impact dominance in the linear regression, which is evidenced by corresponding performance losses. Most of these performance losses are in line with expectations of running more frequently on the in-order backend. Milc, namd, povray, and sphinx even experience performance gains due to saving wasted cycles spent in branch misprediction phases.

5.3 Conclusions on Fine-Grained Resource Mapping

As heterogeneity on-chip increases, the constraints of Dennard scaling and the utilization wall will become more important. Today, heterogeneous general-purpose multi-cores already exist in mobile devices manufactured by ARM and Nvidia. Many server grade chips have GPUs embedded with ISA support. Researchers are attempting to integrate FPGAs on-chip as well. With this current trend of increasing specialization and heterogeneity, fine-grained resource mapping presents a new and exciting method for exploiting more performance and energy savings from processors.

In the future, hardware mapping techniques might be helpful in saving programmer effort. It is already difficult to perform resource allocation on a macro-programmer level, and we as chip architects can seek to eliminate the pain of manually allocating computation resources as one might allocate memory in the C programming language. If done correctly, fine-grained mapping in hardware can be seen as a black-box optimization operation with which the future programmer will not need to concern herself. Although the contribution of this dissertation is small, we humbly hope it helps further the field toward a future of effortless resource mapping.

References

[AR94] S.G. Abraham and B.R. Rau. "Predicting Load Latencies Using Cache Profiling." Technical report, Hewlett Packard Lab, 1994.

[ARM13] ARM. "big.LITTLE Technology: The Future of Mobile." In *ARM Limited whitepaper*, 2013.

[BC06] M. Becchi and P. Crowley. "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures." In *Conference on Computing Frontiers*, 2006.

[BSC08] F.A. Bower, D.J. Sorin, and L.P. Cox. "The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling." In *MICRO*, 2008.

[CDA03] L. Chen, S. Dropsho, and D. Albonesi. "Dynamic Data Dependence Tracking and its Application to Branch Prediction." In *HPCA*, 2003.

[CFS99] Y. Chou, J. Fung, and J.P. Shen. "Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection." In *ICS*, 1999.

[CHA15] T.E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. "The Load Slice Core Microarchitecture." In *ISCA*, 2015.

[CJ09] J. Chen and L. John. "Efficient program scheduling for heterogeneous multi-core processors." In *DAC*, 2009.

[CJE12] K. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. "Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)." In *ISCA*, 2012.

[Cor06] S.P.E. Corporation. "SPEC 2006.", 2006.

[CTW04] J. Collins, D. Tullsen, and H. Wang. "Control Flow Optimization Via Dynamic Reconvergence Prediction." In *MICRO*, 2004.

[CV01] C. Cher and T.N. Vijaykumar. "Skipper: A Microarchitecture for Exploiting Control-flow Independence." In *MICRO*, 2001.

[CWT01] J. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y.F. Lee, D. Lavery, and J.P. Shen. "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads." In *ISCA*, 2001.

[EBA11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. "Dark Silicon and the End of Multicore Scaling." In *ISCA*, 2011.

[ET98] A. N. Eden and T.Mudge. “The YAGS Branch Prediction Scheme.” In *MICRO*, 1998.

[GAS04] A. Gandhi, H. Akkary, and S. Srinivasan. “Reducing Branch Misprediction Penalty via Selective Branch Recovery.” In *HPCA*, 2004.

[Gre12] P. Greenhalgh. “big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7.” In *ARM Limited whitepaper*, 2012.

[HP03] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3rd edition, 2003.

[JAO15] Z. Jin, G. Asilioglu, and S. Onder. “Mower: A New Design for Non-blocking Misprediction Recovery.” In *ICS*, 2015.

[KFJ03] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D. Tullsen. “Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction.” In *MICRO*, 2003.

[KMS05] H. Kim, O. Mutlu, J. Stark, and Y. Patt. “Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution.” In *MICRO*, 2005.

[KRH10] D. Koufaty, D. Reddy, and S. Hahn. “Bias Scheduling in Heterogeneous Multi-core Architectures.” In *EuroSys*, 2010.

[KTR04] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. “Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance.” In *ISCA*, 2004.

[LAS09] S. Li, J. Ahn, R. Strong, J. Brockman, D.M. Tullsen, and N.P. Jouppi. “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures.” In *MICRO*, 2009.

[LPD12] A. Lukefahr, S. Padmanabha, R. Das, F.M. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. “Composite Cores: Pushing Heterogeneity into a Core.” In *MICRO*, 2012.

[LPD14] A. Lukefahr, S. Padmanabha, R. Das, R. Dreslinski, T. Wenisch, and S. Mahlke. “Heterogeneous Microarchitectures Trump Voltage Scaling for Low-Power Cores.” In *PACT*, 2014.

[LPD16] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. “Exploring Fine-Grained Heterogeneity with Composite Cores.” In *IEEE Transactions on Computers*, 2016.

[MCE02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. “Simics: A full system simulation platform.” In *Computer*, 2002.

[MPM14] T. Muthukaruppan, A. Pathania, and T. Mitra. "Price Theory Based Power Management for Heterogeneous Multi-Cores." In *ASPLOS*, 2014.

[MSB05] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. "Multifacet's general execution-driven multiprocessor simulator (gems) toolset." In *ACM SIGARCH Computer Architecture News*, 2005.

[NVI15] NVIDIA. "NVIDIA Tegra X1: NVIDIA'S New Mobile Superchip." In *NVIDIA Corporation whitepaper*, 2015.

[PLD13] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke. "Trace Based Phase Prediction For Tightly-Coupled Heterogeneous Cores." In *MICRO*, 2013.

[PLD15a] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke. "DynaMOS: Dynamic Schedule Migration for Heterogeneous Cores." In *MICRO*, 2015.

[PLD15b] V. Petrucci, M. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang. "Octopus-Man: QoS-Driven Task Management for Heterogeneous Multicores in Warehouse-Scale Computers." In *HPCA*, 2015.

[PSW04] V.M. Panait, A. Sasturkar, and W.F. Wong. "Static Identification of Delinquent Loads." In *CGO*, 2004.

[RBR02] S.E. Raasch, N.L. Binkert, and S.K. Reinhardt. "A Scalable Instruction Queue Design Using Dependence Chains." In *ISCA*, 2002.

[RMS98] A. Roth, A. Moshovos, and G.S. Sohi. "Dependence Based Prefetching for Linked Data Structures." In *ASPLOS*, 1998.

[RNA12] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann. "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge." In *IEEE Micro*, 2012.

[SCH15] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud. "Long Term Parking (LTP): Criticality-aware Resource Allocation in OOO Processors." In *MICRO*, 2015.

[SJ01] P. Shivakumar and N.P. Jouppi. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model." Technical report, Compaq Computer Corporation, 2001.

[SJL01] S.T. Srinivasan, R.D. Ju, A.R. Lebeck, and C. Wilkerson. "Locality vs. Criticality." In *ISCA*, 2001.

[SRA13] S. Srinivasan, R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu. "A Study on Polymorphing Superscalar Processor Dynamically to Improve Power Efficiency." In *ISVLSI*, 2013.

[SSJ09] D. Shelepov, J. Saez, S. Jeffery, A. Federova, N. Perez, Z. Huang, S. Blagodurov, and V. Kumar. "HASS: A Scheduler for Heterogeneous Multicore Systems." In *Operating Systems Review*, 2009.

[VSG10] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryskin, J. Lugo-Martinez, S. Swanson, and M.B. Taylor. "Conservation Cores: Reducing the Energy of Mature Computations." In *ASPLOS*, 2010.

[ZS00] C. Zilles and G. Sohi. "Understanding the Backward Slices of Performance Degrading Instructions." In *ISCA*, 2000.