4-23-2012

A Hardware and Software Integrated Approach for Adaptive Thread Management in Multicore Multithreaded Microprocessors

Lichen Weng
Florida International University, lichen.weng@fiu.edu

DOI: 10.25148/etd.FI12071106

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A HARDWARE AND SOFTWARE INTEGRATED APPROACH FOR ADAPTIVE THREAD MANAGEMENT IN MULTICORE MULTITHREADED MICROPROCESSORS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

by

Lichen Weng

2012

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Lichen Weng, and entitled A Hardware and Software Integrated Approach for Adaptive Thread Management in Multicore Multithreaded Microprocessors, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Malek Adjouadi

Gang Quan

Raju Rangaswami

Chen Liu, Major Professor

Date of Defense: April 23, 2012

The dissertation of Lichen Weng is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2012

DEDICATION

To my parents.

ACKNOWLEDGMENTS

First and foremost I offer my sincerest gratitude to my supervisor, Dr. Chen Liu, who has supported me throughout my doctoral study with his patience and knowledge. Dr. Liu has taught me in tremendous topics, which led me from a fresh graduate to a qualified doctor. Meanwhile, I attribute the level of my doctoral degree not only to my major professor, but also other committee members: Drs. Malek Adjouadi, Gang Quan and Raju Rangaswami. Without the valuable and great guidance from the committee, the dissertation would not have been completed.

Furthermore, honest appreciations are delivered to the faculty in the Department of Electrical and Computer Engineering, the School of Computing and Information Sciences and the Telecommunications and Information Technology Institute, for their inspiring teaching and informative lectures. I do honor the fine and prompt aid from the friendly staff in our department in various events.

Last but not least, I am absolutely grateful for the gorgeous help and constructive suggestions from my colleagues and friends. Especially in the Computer Architecture and Microprocessor Engineering Laboratory, we have had so much fun and so many fruitful collaborations.

ABSTRACT OF THE DISSERTATION

A HARDWARE AND SOFTWARE INTEGRATED APPROACH FOR ADAPTIVE THREAD MANAGEMENT IN MULTICORE MULTITHREADED MICROPROCESSORS

by

Lichen Weng

Florida International University, 2012

Miami, Florida

Professor Chen Liu, Major Professor

The Multicore Multithreaded Microprocessor maximizes parallelism on a chip for the optimal system performance, such that its popularity is growing rapidly in high-performance computing. It increases the complexity in resource distribution on a chip by leading it to two directions: isolation and unification. On one hand, multiple cores are implemented to deliver the computation and memory accessing resources to more than one thread at the same time. Nevertheless, it limits the threads' access to resources in different cores, even if extensively demanded. On the other hand, simultaneous multithreaded architectures unify the domestic execution resources together for concurrently running threads. In such an environment, threads are greatly affected by the inter-thread interference. Moreover, the impacts of the complicated distribution are enlarged by variation in workload behaviors. As a result, the microprocessor requires an adaptive management scheme to schedule threads throughout different cores and coordinate them within cores.

In this study, an adaptive thread management scheme was proposed, integrating both hardware and software approaches. The instruction fetch policy at the hardware level took the responsibility by prioritizing domestic threads, while the Operating System scheduler at the software level was used to pair threads dynami-

cally to multiple cores. The tie between them was the proposed online linear model, which was dynamically constructed for every thread based on data misses by the regression algorithm. Consequently, the hardware part of the proposed scheme proactively granted higher priority to the threads with less predicted long-latency loads, expecting they would better utilize the shared execution resources. Meanwhile, the software part was invoked by such a model upon significant changes in the execution phases and paired threads with different demands to the same core to minimize competition on the chip. The proposed scheme was compared to its peer designs and overall 43% speedup was achieved by the integrated approach over the combination of two baseline policies in hardware and software, respectively. The overhead was examined carefully regarding power, area, storage and latency, as well as the relationship between the overhead and the performance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

The Average Baseline Weighted IPC	abwIPC
The Adaptive Thread Management Scheme	ATMS
The Normalized Average IPC	avgIPC
Blended	BD
Computation Intensive	CI
Cycle Per Instruction	CPI
The Data Gating policy	DG
The Dynamic Scheduling Policy	dMIX
The Data Cache Warn policy	DWarn
The Scheduling Evaluation Overhead	EO
First Come First Serve	FCFS
Front Side Bus	FSB
The Hardware Assisted Scheduling Policy	HASP
Instruction Level Parallelism	ILP
Instruction Per Cycle	IPC
Instruction Set Architecture	ISA
Job Level Parallelism	JLP
Last Level Cache	LLC
Long-Latency Load	LLL
Memory Intensive	MI
The Microprocessor without Interlocked Pipeline Stages	MIPS
The Static Mix-Scheduling Policy	sMIX
Multicore Multithreaded MicroProcessor	MMMP
Memory Management Unit	MMU
The Model construction Overhead	MO

The Mono-Scheduling Policy	Mono
Misses Per Kilo Instructions	MPKI
Misses Per Million Instructions	MPMI
Multicore Single-threaded MicroProcessor	MSMP
The Online Linear Model	OLM
The Ordinary Least Square Regression	OLS
Operating System	OS
The Phase detection Overhead	PO
Prediction Expectation	PE
Program Counter	PC
The Regression-based Algorithm to Prioritize Threads	RAPT
Reduced Instruction Set Computing	RISC
Re-Order Buffer	ROB
Single-core Multithreaded MicroProcessor	SMMP
Simultaneous Multithreading	SMT
The Sampling Overhead	SO
The Sampling Period	SP
Single-core Single-threaded MicroProcessor	SSMP
Standard Deviation	StDev
Thread Level Parallelism	TLP
The Thread migration Overhead	TO
Windows Size	WS

CHAPTER 1

INTRODUCTION

High performance is greatly pursued in the microprocessor design and nowadays parallelism plays an essential role. The Multicore Multithreaded MicroProcessor (MMMP) employs different levels of parallelism to generate more throughput. Via implementing multiple cores in a processor, the Thread-Level Parallelism (TLP) is well utilized [1]. Further in the same core, the implementation of multiple threads is able to maximize on-chip parallelism, which is especially true for the Simultaneous Multithreading (SMT) architectures [2]. The SMT is defined as fully shared execution resources by several concurrently running threads in the same core, such that both TLP and Instruction-Level Parallelism (ILP) are utilized [3]. However, "Memory Wall" issues [4] still exit in such architectures, and may introduce great impacts on thread performance and system throughput. Moreover, due to the complexity of resource allocation and the variation in workload behaviors, it becomes more difficult to manage the shared resources in the MMMP. Thus more efforts, even innovative approaches, are required to manage the MMMP for better utilization of the resources and thus higher performance.

1.1 The Motivation

The MMMP increases the complexity in resource distribution, and thus the difficulty in resource management. In particular, there are two opposite scenarios in distributing the on-chip resources in the MMMP: isolation and unification. On one hand, since several physical cores are implemented on the chip, the resources are isolated within different cores. Although it is used to explore the TLP, it makes a thread unable to access the resource in other cores. This issue is especially undesired

when certain resources are idle in one core, but highly utilized in another core. It reduces the probability to schedule in a nature or heuristic way, such that resources are utilized comprehensively. On the other hand, threads in the same core live on the domestic resources jointly. As a result, the thread behavior is not independent any more, but rather has universal impacts on domestic threads and even threads in other cores. Thread performance may be degraded because of severe competition for the same resource, as well as inappropriate resource allocation, *e.g.*, ignorance of threads' resource demands. Therefore, it requires well-defined management scheme in the complex architecture to optimally utilize the hardware resources [2].

The complexity in hardware is not the only motivation, but workload variation also prompts a novel design. The phase behavior has been studied by dozens of researchers, which indicates workload behavior is somewhere between chaotic and periodic [5]. Phases exist in most workloads with granularity from thousands of instructions to millions of instructions. Within a phase, workload may exhibit similar or stable behaviors, with respect to retired instructions, memory accesses and resource demands. There are a few decent designs that explore phase behaviors for better system performance, but most of them result from a fixed doctrine or a universal assumption. In order to better make use of certainty in the phase behaviors, we advocate monitoring phase changes in a more active way, such that the system resources are allocated adaptively in observance of the varying demands.

1.2 Scope of the Study

Among different levels of resource management schemes, this study is focused on two: the hardware scheduling policy that is executed at the architectural level, and the software scheduling policy that usually happens at the Operating System (OS) level. Hardware scheduling policies take the responsibility to manage the

fully shared execution resource within a core, while their approaches may involve prioritization at the fetch stage based on memory accesses and thread behaviors. Hence the instruction fetch policy is a critical entry to allocate resources among simultaneous threads, which exactly belongs to the spectrum of this study. On the other hand, a fundamental objective of software scheduling policies is to increase system throughput in high-performance computing, while load balancing and power and thermal control might also be embedded. This study focuses on higher performance by scheduling, and thus it expects that the proposed design to overcome the boundary among different cores. In summary, both policies share the similar objective to better utilize system resources and the effectiveness of their decisions are highly dependent on workload behaviors. However, they may have the followed dissimilarities:

- **Target**: A software scheduling policy mainly manages threads across multiple cores while a hardware scheduling policy adjusts threads and their instructions in the same core;
- **Granularity**: A software scheduling policy's decisions are usually valid for millions of CPU cycles while a hardware scheduling policy may evaluate every clock cycle;
- **Accessibility**: A software scheduling policy is designed from a top-down view over multiple cores while a hardware scheduling policy is often embedded in computer architecture within every core.

Therefore, the two scheduling policies are not mutually exclusive, but rather have the potential to form a symbiotic relationship. In such a relationship, they are expected to work collaboratively to overcome the natural gap between hardware and software for optimal resource allocation.

Consequently, this study proposes an Adaptive Thread Management Scheme (ATMS) in the MMMP, and the merit lies in integration and adaptability. The integration refers to the active collaboration between hardware (the architectural level) and software (the OS level). It includes, first of all, the mutually beneficial relationship between the OS and the computer architecture. The software scheduling policy provides the architectural level with a proper working environment based on universal hardware resources and thread demands. Meanwhile, the thread demands are collected and partially analyzed at the architectural level and passed to the OS level for even better scheduling. Secondly, it also refers to the cooperation to reinforce management goals at different levels. The software scheduling policy is able to utilize multiple cores to explore TLP for better performance, while it relies on the hardware scheduling policy to further maximize TLP and ILP. The SMT scheduling policy continues the management scheme to fully utilize the shared resource at the pipeline level.

On the other hand, in account for the variation in thread behavior, adaptability is explored through constructing the online model without *a priori* knowledge, such that the ATMS is capable of accessing the critical demands for better resource allocation, in spite of the varying thread behaviors. As they change, the proposed scheme samples the new phases and the periodic information is extracted mathematically. Therefore, the varying behaviors along the execution of a certain workload and various behaviors among different workloads are better summarized to guide both hardware and software scheduling policies. As a result, adaptability is reached in the proposed design.

1.3 Significance of the Study

The MMMP is widely adopted as a critical trend in the design of the next-generation microprocessor. There have been some initial products in the commercial market already, including the Intel® Core™processor family [6, 7], the IBM® Power7 [8] and the Intel® Xeon™[9]. Even though such designs are able to achieve higher performance than previous generations, their resource efficiency might not be optimized until well-designed scheduling policy is implemented [3]. It means there is more difficulty and thus larger potential for better utilization of the hardware resources in the MMMP. Consequently, the proposed scheme makes the MMMP fully utilize the existing hardware resources, and adaptively distribute the shared resources according to the thread demands. Hence, the contributions of this research fall into four aspects:

1. Constructed the online linear model to estimate threads' resource demand for the architectural level and the OS level with no *a priori* knowledge;
2. Prioritized the threads proactively in a core such that the shared resources in the SMT core were utilized efficiently;
3. Scheduled threads adaptively according to their resource usage in a sustainable approach;
4. Integrated the hardware and software scheduling policies to further fulfill the responsibility of resource management.

As far as the overall system performance is concerned, the proposed scheme will optimize the resource allocation in the MMMP. As mentioned above, both isolation and unification exist in resource distribution in the MMMP. The ATMS takes into consideration the overall hardware resources and thread demands, such

that the thread scheduling is optimized by dynamic manipulation. Though the software scheduling policy is at the OS level, the resource demands are collected and processed directly at the architectural level, which provides the software part with better observation. Furthermore, scalability is a critical challenge as the number of cores and threads increases, so the OS scheduling policy has to transmit partial management goals to the architectural level. In such a case, the ATMS strives to use multiple levels of efforts to achieve the management objective in a complex hardware environment. As a result, the hardware and software scheduling policies work jointly as the ATMS exploits both TLP and ILP for optimal system performance.

From the perspective of the threads, they are more likely to obtain the demanded resources and to be less affected by neighbor threads in the complicated environment. In the MMMP, the resources are isolated in different cores, meaning there are merely limited resources shared by domestic threads in a single core. When the proposed design pairs threads that demand different resources in the core, the contention for the same resource is minimized. Hence they are better satisfied by the limited resource budget through the proposed scheme. Furthermore, the threads sharing pipeline resources are less affected by Long-Latency Loads (LLL). When some threads suffer from the LLL, other threads are able to send more instructions to utilize the shared execution resources, which has minimal impact on the waiting threads.

1.4 The Dissertation Organization

The background information has been introduced in this chapter to show the motivation and contribution of this study in an intuitive tone. In Chapter 2, theoretical perspective is taken to examine the related research. It first introduces the hard-

ware architecture that the ATMS will be implemented in, and then reviews existing scheduling policies in the SMT environment and in the multicore environment.

The ATMS is introduced by addressing three technical topics in Chapters 3, 4 and 5, respectively:

1. **Online Linear Model (OLM)**: It is constructed during execution and updated continuously, such that it ensures adaptability of the proposed scheme.
2. **Regression-based Algorithm to Prioritize Threads (RAPT)**: The hardware scheduling policy utilizes the OLM at the architectural level to predict future thread demands in prioritizing simultaneous threads.
3. **Hardware-assisted Scheduling Policy (HASP)**: The software scheduling policy explores adaptability in the OLM through pairing threads dynamically across multiple cores at the OS level.

After the hardware and software scheduling policies, the aggregation of different components is proposed in Chapter 6. Their performance, *i.e.*, system throughput and average thread performance, is presented with appropriate overheads integrated, such that the proposed designs are validated comprehensively. Eventually, Chapter 7 concludes this dissertation and recommends some interesting future work.

CHAPTER 2

THEORETICAL PERSPECTIVE AND LITERATURE REVIEW

The related studies are reviewed comprehensively in this chapter. There are several theoretical aspects that should be clarified before a new scheme is proposed:

1. Architectural development of the MMMP
2. Multithreaded scheduling policies
3. Multicore scheduling policies
4. The phase behavior

The first topic in Chapter 2.1 is the hardware architecture that this study is focused on, while Chapter 2.2 and 2.3 belong to hardware and software levels respectively. You will find they are usually designed separately by other researchers, but will be collaborating closely in the proposed design, which is the major innovation of this study. Chapter 2.4 is about the resource demands of the workloads.

2.1 Hardware Architecture of the MMMP

The MMMP is composed of two architectural designs: the multicore architecture and the multithreaded architecture. The former one designs several physical cores on the same chip, such that threads in different cores can explore TLP greatly. The latter one employs the Thread Context (TC) to implement logical cores in the same physical core. They share most execution resources in the same core to maximize the parallelism on the chip. An example of studied two-core four-threaded microprocessor is shown in Figure 2.1.

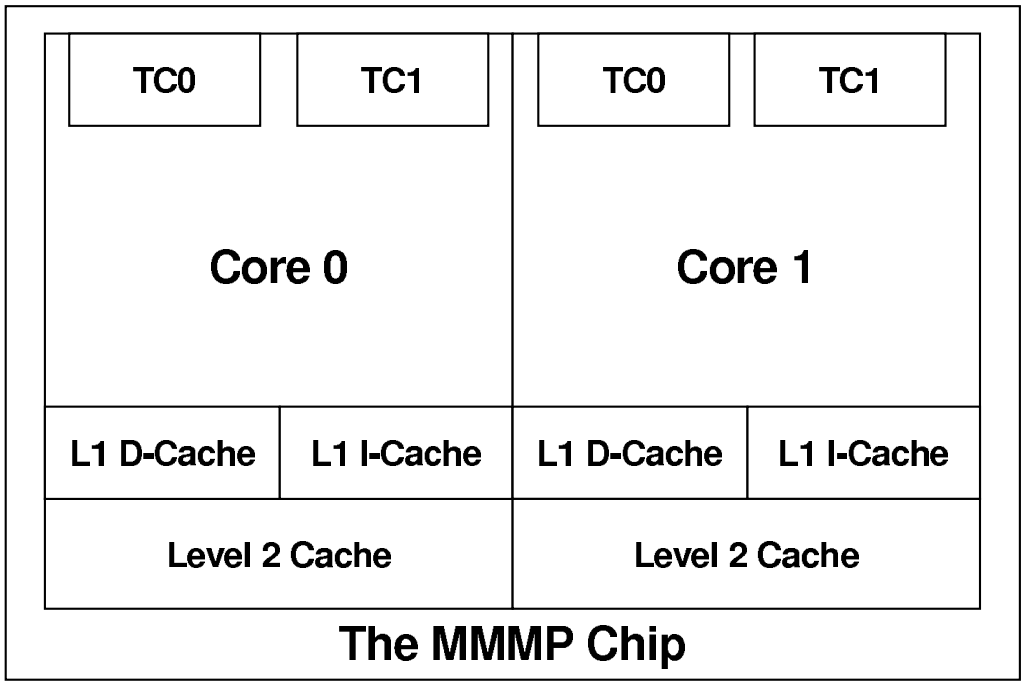


Figure 2.1: An example of two-core four-threaded microprocessor

2.1.1 The SMT Architecture

Furthermore, the SMT was first proposed as a multi-streamed superscalar processor by Yamamoto *et al.* [10]. The authors validated the idea mainly via mathematical analysis, though system performance was obtained from emulation. They did not compare the proposed design with other similar architectures, but rather focused on the difference between the simulated results and the predicted results. Tullsen *et al.* [11] proposed the architecture named SMT, in which several threads compete for each of the issue slots in each cycle. The difference was studied among the SMT, fine-grain multithreaded, single-issue and dual-issue designs. In general, the multi-threaded architecture is able to minimize the vertical waste, while the SMT further reduces the horizontal waste in microprocessors [12]. Assuming two simultaneous threads T0 and T1 are implemented in the core, the example of minimized horizon-

tal and vertical waste from the single-threaded architecture to the SMT architecture is shown in Figure 2.2.

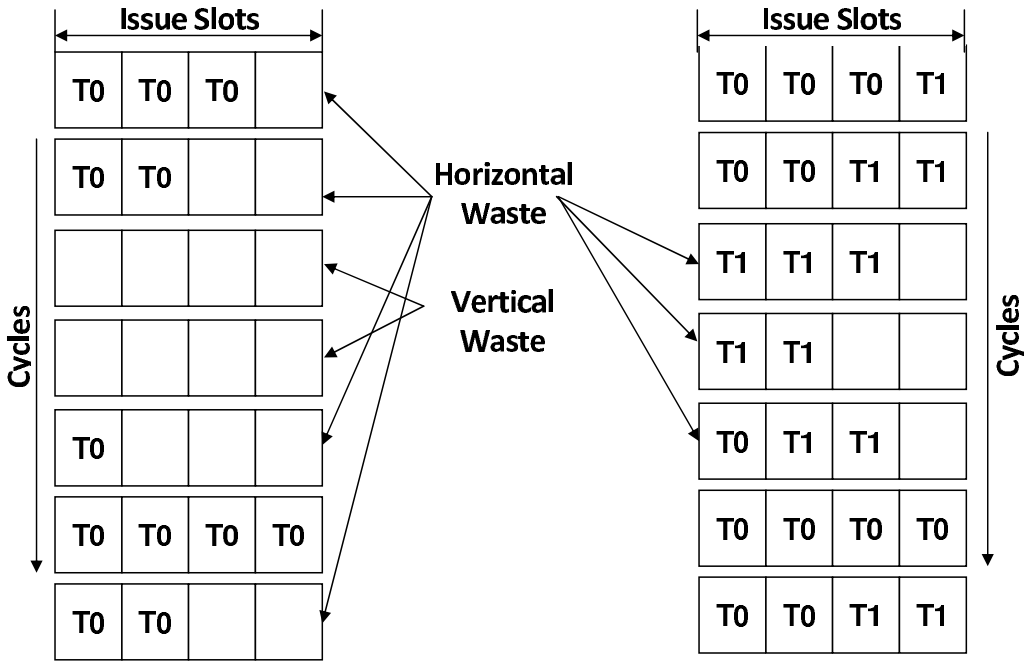


Figure 2.2: Minimization of the horizontal and vertical waste by the SMT

Given an SMT core that is able to fetch instructions from multiple threads and its critical ability to issue and execute instructions from multiple threads at every clock cycle [13, 2], the computer architecture requires a policy that defines the fetched sources. Practically, the policy is noted as an instruction fetch policy, which concludes the priorities of domestic threads in the fetch stage for the fetch engine. It is of great importance to the performance of the SMT core, because it defines the instructions that utilize the shared resources in the pipeline. Therefore, it is expected to allocate the shared resources in the SMT core to those instructions and threads that have the ability to better utilize them, such that the system throughput is maximized and so is the overall performance. As a result, instruction fetch

policies are widely studied as a convenient and effective scheduling policy in the SMT environment, which will be the focus at the hardware level in this study.

2.1.2 The Multicore Architecture

The multicore architecture is mainly focused on Job-Level Parallelism (JLP) and TLP via executing several threads in different cores on the same chip [14]. Its design comes with two flavors: the heterogeneous architecture and the homogeneous architecture. There are several different cores on the chip in a heterogeneous design. For example, in IBM® Cell Broadband Engine™[15], the threads are processed by one Power Processing Element (PPE) and then distributed to several Synergistic Processing Elements (SPE). The performance improvement may be achieved when the parallel threads are executed by multiple SPEs, while sequential threads by the the PPE running at a higher frequency [1]. On the other hand, a unique example of the homogeneous multicore microprocessor is the Intel® Single Chip Cloud Computer [16], on which 48 identical cores are implemented. It does not employ a separate core to manage the threads but may rely on the multicore scheduling policy in the OS to fully utilize the shared resources in cores.

Considering multiple cores on a chip, the resources are isolated among different cores. Without a thread management scheme, threads are able to access merely resources within the same core. Meanwhile, the threads in a core jointly share the local resources, such that the utilization of the resources are defined by the thread scheduling. Some resources may be highly demanded in a core, while rarely used in another core. From the perspective of threads, their resource demands could not be met because of the resource isolation, even though there are in total enough hardware resources on the chip. As a result, a multicore scheduling policy is needed

to manage the threads across different cores, and it ought to take into consideration the resource unification by the SMT architecture as well.

2.2 Scheduling in the SMT Architecture

The instruction fetch policy was originally set as Round-Robin by Yamamoto *et al.* [10], meaning all threads were fetched alternatively. In order to improve the performance of the SMT architectures, Tullsen *et al.* [17] proposed the ICOUNT to assign the priority according to the in-flight instructions. It assumed that the threads with fewer instructions in front-end stages in the pipeline were able to retire more instructions than other threads. As to the system throughput, those threads were expected to committee more instructions in the future and thus were favored in the fetch stage. However, it was argued that the ICOUNT might attach too much importance to high throughput thread, and Tullsen *et al.* realized that the Long-Latency Load (LLL) was the major obstacle to full utilization of the shared resources [17]. As a result, the LLL will be explained in Chapter 2.2.1, and then the hardware scheduling policies that were related to the LLL will be discussed. After that, the policies based on machine learning will be reviewed as well. Moreover, some other policies were proposed for parallel workloads, which have their own characteristics compared to multiprogramming workloads.

2.2.1 The Long-Latency Load

The LLL refers to a miss in the last level cache (LLC), and is caused by the "Memory Wall" [18]. LLC misses lead a thread to wait for the data from lower memory hierarchy, which costs much more time than most operations in the pipeline and data from caches. Empirically the latency is increased from several CPU cycles for the L1 cache to hundreds of CPU cycles for main memory. Due to dependence among

instructions, the thread suffering from the LLL will eventually stall. Consequently, it denies the instruction fetch policies that have biased favor on high-throughput threads, because of the "Memory Wall" and limit ILP [19]. Moreover, the dependent instructions are still floating in the pipeline, occupying the shared resources such as ReOrder Buffer (ROB) and Instruction Queue (IQ). These resources are limited in the SMT architecture and may be demanded by other threads for immediate advancement. As a result, most SMT scheduling policies after the ICOUNT are related to the LLL issue.

2.2.2 Instruction Fetch Policies for the LLC

Given the speed of the memory system develops slower than that of processors, it is less likely that this issue can be solved before long [20]. Therefore, several instruction fetch polices were proposed to deal with the LLL in the SMT.

The Stall and the Flush [21] were both based on the LLL, *i.e.*, L2 cache misses, which were recognized by the system some time after a load. The Stall policy stopped the thread experiencing any L2 cache miss from further fetching, because such a thread was less likely to use more resources efficiently. Nevertheless, the resources being occupied by such a thread could not be released until the data come back. More aggressively than the Stall, the Flush expelled the instructions from such thread to release the occupied resources. Hence, those resources would be available for other threads. It ensured the utilization of shared resources, but required extra overhead for flushing and re-fetching. The essential difference between the Stall and the Flush was whether to release the occupied resources or not. Actually they might lead to different performance depending on the resource budget in the SMT. Consequently, Cazorla *et al.* [22] proposed to improve the Stall and the Flush, such that at least one thread was active in the pipeline. In particular, the oldest stalled

thread was resumed when the final active thread was suspended due to a new LLC miss. They further proposed the Flush++ to enjoy the advantages of both the Stall and the Flush. The Flush++ executed the improved Stall when the system resource was relatively enough, *i.e.*, there were up to four threads in the processor, while followed the improved Flush when the resources were limited, *i.e.*, more than four threads.

2.2.3 Proactive Instruction Fetch Policies

Considering it may be late to act upon a LLC miss, some other instruction fetch policies would rather depend on a L1 cache miss, such that they would respond to the LLL in advance. They were expected to minimize the inefficient occupancy on the shared resources. El-Moursy *et al.* [23] proposed the Data Miss Gating (DG) to observe the L1 cache miss, and stopped a thread from fetching when it had n outstanding L1 cache misses. Although they studied several values, $n = 1$ has been widely discussed. It resulted in an instruction fetch policy that suspended a thread when there was unsolved L1 cache miss associated with it, which was obviously different from previous work.

Actually, the overhead here to stop a thread with any outstanding L1 cache miss was still high, because the basic concept of the SMT was to utilize more instructions from different threads. In the DG, it might happen that most threads were stopped, such that a lot of resources were idle due to lack of instructions. Therefore, Cazorla *et al.* [24] proposed the Data Cache Warn (DWarn) to adjust the priority without stalling. The DWarn was also based on L1 cache misses, but it reduced the threads' priority with unsolved L1 cache misses in the fetch stage. Consequently, such threads were fetched when other threads without any unsolved L1 cache miss could not satisfy the fetch width. The DWarn strived to keep the fetch width as full as

possible, such that the shared resources were utilized by more instructions. Overall, it generated more system throughput than other instruction fetch polices according to their study. In summary, the proactive instruction fetch policies relied on the relationship between L1 and L2 data misses, but there is rarely a precise description about it. Given the execution phases in workloads behaviors, a better observation on the relationship between L1 and L2 data misses, and thus the workload behaviors, would benefit the instruction fetch policies in the STM architectures.

2.2.4 More SMT Scheduling Policies

Furthermore, several studies explored more design spaces in the architectural level as well as in the software level. They were not typical instruction fetch policies, and usually involved the OS in algorithmic computation. Therefore, they might face challenges: high overhead and low adaptability. The overhead is mainly introduced by submitting information to the OS, interrupting the OS and calculating algorithmic parameters; while the disadvantage in adaptability is due to the failure to consider execution phases. Consequently, this study ought to target at an instruction fetch policy that is equipped with low overhead and high adaptability.

Cazorla *et al.* [24] proposed to collect the information of registers and classify instructions with respect to integer, floating-point and load/store units, such that the resource usage was summarized. They assumed the threads without any outstanding L1 cache miss hardly needed any memory accessing resources, while those experiencing L1 cache misses could not perform better even if they were provided with extra computation resources. Therefore, threads fell into different groups with pre-defined resource quotas and fetch throttling was employed to enforce such quotas. On the other hand, Wang *et al.* [25] evaluated thread efficiency via monitoring resource entries and committed instructions. They introduced the Committed In-

struction Per Resource Entry (CIPRE) to indicate the resource usage efficiency. Consequently, the threads that were capable of retiring more instructions with less resource entries were favored by the system, and it would fetch more instructions from them to improve the overall resource efficiency and thus system throughput, at the expense of hardware counters and the CIPRE computation by the OS.

Choi *et al.* [26] proposed to achieve the optimal performance via Hill-climbing. The system estimated the impacts of different resource distributions via actually executing them for one epoch. The best observation was employed for future epochs and gradually the optimal resource distribution was realized in the SMT environment. Obviously this scheduling policy introduced considerable hardware overhead in monitoring resource and evaluating performance in the OS.

Recently, as in-order execution is employed again by some modern processors, *e.g.*, the Single Chip Cloud Computer by Intel® [16], research on the combination of in-order and out-of-order execution was done by Wang *et al.* [27]. They selected the dependent instructions of a cache miss and arranged them in an individual queue, which was closely related to the instruction queue. When the data came back and such instructions were ready, they were executed in-order in the pipeline. This design reduced instruction window occupancy rate and could be implemented together with other fetch policies.

2.2.5 SMT Scheduling for Parallel Programmes

Originally the multithreaded workloads used by Tullsen *et al.* were composed of multiple independent threads, *e.g.*, benchmarks from SPEC95, which we call multiprogramming. The communication among threads increases rapidly when parallel programmes are employed *e.g.*, Splash [28], SPECjbb [29] and RUBiS [30]. Some scheduling efforts were finished at the granularity of instructions, and they are ad-

dressed here. Later contexts will discuss about the multicore scheduling policies for parallel programmes.

Long *et al.* [31] proposed to detect the Single Program Multiple Data (SPMD) portion in the benchmark, such that it requires only one instruction fetch for different threads. If the identical instructions required the same input, the execution was limited to one time while the results were duplicated to different threads. This design utilized preliminary analysis for less work in fetching and execution. Meanwhile, Cheng *et al.* [32] proposed to arrange memory accessing instructions according to the system capability. They justified the thread resource demands via Misses Per Kilo Instructions (MPKI). Assuming redundant memory accesses at the same time were not efficient, the system tried to schedule threads such that memory accesses were under a threshold. The threshold was called Memory Task Limit in the system, which was managed to achieve better performance. Bhattacharjee *et al.* [33] investigated thread criticality by the weighted summation of L1 and L2 misses, such that future thread behaviors were predicted based on the history information. The voltage and frequency were scaled in accordance with the thread criticality predictor (TCP), such that the power consumption is reduced, instead of improved system throughput.

Similarly in other designs about the parallel programmes in the multicore architecture, *e.g.*, [34, 35, 36, 37, 38, 39, 40], there seemed to be more potential due to more communications among threads than a single program. It easily overwhelms the cost to migrate threads across different cores, such that improvement is virtually deterministic for parallel programmes [37]. However, to find out the critical sections in parallel programmes, *a priori* knowledge is sometime required, *e.g.*, [34], which reduced the adaptability greatly. Some other designs, *e.g.*, [36, 37, 38, 39], were highly dependent on history information, and failed to adopt a statistical view

about workload behaviors. Moreover, Kokku *et al.* [35] and Cai *et al.* [40] mainly targeted at less power consumption by network processors.

In summary, they all agreed that the critical sections ought to be recognized and improved in the scheduling policy. Moreover, reducing the communication time within parallel programmes was convenient for performance improvement, especially when compilers were employed for *a priori* knowledge, *e.g.*, [41, 31, 40]. However, possible contribution from the instruction fetch policy was underestimated, and the mutual impacts among different programmes, similar to the inter-thread interference for multi-programming workloads, were not addressed well.

2.3 Multicore Scheduling in the OS

There are many OS scheduling policies proposed for parallel architectures, and the majority of them reside in the OS at the software level. The underlying hardware architecture may be either homogeneous or heterogeneous, but they are not mutually exclusive, because homogenous cores exist partially in a heterogeneous architecture. For example, IBM® Cell Broadband Engine™[15] has identical SPEs. Consequently, the research on homogeneous architectures in this study is eligible for both areas. Furthermore, parallel architectures are extensively employed in clusters or cloud computing. They might not be exactly the same as a Chip MultiProcessor (CMP), in terms of more traffic through I/O interfaces and network communications, but their results are valuable in designing a multicore scheduling policy in the OS. In other words, the proposed schemes will be easily converted to a scheduling policy for cloud computing.

2.3.1 Homogeneous Microprocessors

The multicore scheduling policies are widely studied to pursue various goals, and high performance is a major one. Especially in the MMMP, the resources are distributed across different cores, so the way in which the threads are scheduled defines how the resources are utilized [2].

In the environment of the Single-core Multithreaded Microprocessor (SMMP) with the SMT, Snavely *et al.* [42] proposed the symbiotic scheduling policy to mix jobs with different priorities together, such that the system throughput was increased due to multithreading and co-scheduling. Furthermore, scheduling based on cache usage in the Multicore Single-threaded Microprocessor (MSMP) was studied in [43, 44, 45]. They modified the scheduling policy in the Linux kernel, and the cache usage was balanced among private caches of different cores. However, they did not discuss the unique characteristic of the MMMP with the SMT, that potential for better performance could be explored through fully utilizing the execution resources by several concurrent threads.

Zhuravlev *et al.* [46] concluded that the dominant factor in the MMMP is the contention for resource lower than LLCs, *e.g.*, the DRAM controller, the Front Side Bus (FSB) and prefetch requests. It was found that better performance could be achieved by pairing threads with different demands for such resources [47], compared to scheduling threads with similar demands to the same core. More system throughput was generated by the proposed scheduling policy, but it did not conduct real-time migration, so the policy was a static dispatching policy with *a priori* knowledge. Radojkovic *et al.* executed a large quantity of task assignments among all possible combinations, such that the best observed one was statistically within the top performance group [48]. Their design spent 2 hours on executing every 5000 assignments, and every new set of threads needed such a process. Other studies such

as [49, 50, 46] conducted online manipulation for better scheduling, but fixed epochs with respect to CPU cycles, *e.g.*, 100 million CPU cycles, were employed in their designs. It could not adapt to the various workload behaviors during execution.

2.3.2 Heterogenous Microprocessors

Aside from scheduling within the homogeneous domain, the coordination throughout multiple heterogenous cores also helps improve system performance. Fundamentally, the Amdahl's Law [51] pointed out two sources for performance improvement: the serial part and the parallel part. Hence, a more powerful core, *e.g.*, higher frequency and voltage, can be used to execute the serial part, while exploring TLP by several less power cores [1, 52, 53, 54] for the parallel part. This approach is able to speedup both parts, such that the overall system throughput is increased. However, they did not address the resource sharing in the heterogeneous cores, so they would further need the scheduling policies for homogeneous cores to manage multiple identical cores. Furthermore, the scenario to identify the serial part and parallel part is a great challenge. Eyerman *et al.* [55, 56] proposed an off-line analysis tool to examine the Cycle Per Instruction (CPI) breakdowns for the parallel applications, and then use the results to estimate threads' demands during execution. In order to work in the multithreaded environment, a thread was sampled when it was running alone and then comprehensively running with other threads, such that a better scheduling was concluded [57]. Even though they specified hardware counters for their design, the execution of a thread alone and with other threads should not be a prerequisite for a design.

As a result, even though we see a convenient objective in the scheduling policy in heterogeneous microprocessors, the necessary information is rarely available during

execution. In order to develop an adaptive scheduling policy, a better observation on the various workloads behaviors remains an open question.

2.3.3 Allocation of Memory Resources

Some researchers proposed to manage the shared resources by coordinating accesses to the memory hierarchy, especially the cache [58, 59, 60, 61]. They spent many efforts on monitoring memory accesses by different threads, so the scalability of their designs might not be promising in larger systems. Moreover, because real memory accesses happen when an instruction is ready to retire, such approaches could not take the action proactively enough to prevent the inefficient occupancy on the shared resource in the pipeline. Studies such as [62, 63] proposed complex algorithm to allocate the cache resources to different threads, but their biased scheduling in the cache system may affect user experience and thus reduce Quality of Service (QoS). It was studied by Liu *et al.* [2] that to evenly divide the cache among corresponding threads results in superior performance given its relatively low complexity and overhead.

2.3.4 Scheduling Policies in Cluster

Even though they do not share exactly the same architecture with multicore microprocessors, scheduling policies in many-node clusters are meaningful and useful. They shared the similar concept with our study that resource demands ought to be considered in scheduling. Weinberg *et al.* [64] proposed the symbiotic space-sharing in a many-node supercomputer. They argued that a single job on the node could not fully utilize the resources, so it was better in terms of resource efficiency to co-schedule some other jobs. As a result, they utilized the idle resources in nodes via executing background jobs at a lower priority than the primary jobs in the same

node. Sodan *et al.* [65] proposed to schedule jobs according to their resource demands, which are CPU-bound, disk-bound and network-bound. Even though they mentioned the SMT architecture, in their cluster each node was equipped with independent hierarchical memory system. The difference between the MMMP and their cluster is the resources lower than the LLC, such as the DRAM controller, the FSB and prefecth requests. Moreover, Frachtenberg *et al.* [66] proposed to classify processes into several categories with descending priorities for scheduling in a many-node system. The categorization was in accordance with thread's synchronization requirement and CPU utilization. Although they provided some hints to classify the processes dynamically, their criteria were mainly based on the CPU time and communication time in the network, which was not applied to the shared resources in the SMT.

2.3.5 Thread Replacement

Compared to a static dispatching policy, a dynamic multicore scheduling policy that moves threads across different domains is better at ensuring the scheduling objective [46]. Therefore, most multicore scheduling policies, such as [42, 43, 46, 64, 65], involved dynamic thread migration. Essentially, the total improvement to the system is the difference between the suspending time during migration and the reduced execution time by migrating, so the overhead introduced by migrating threads plays a critical role in defining the final performance.

The overhead conceptually results from rescheduling the process to another hardware thread on another core, manipulating the page table and warming up a new cache in a homogeneous multicore architecture [67]. If there is a heterogeneous architecture, binary translation and state transformation are further required to execute the task in a new Instruction Set Architecture (ISA) [68]. There are many

researchers who conducted the studies on easier and more efficient migrations, such as [68, 69, 70, 71], but they were focused on the heterogeneous architectures, and were unable to reduce the overhead to completely zero.

Moreover, DeVuyst *et al.* [68] found that the unit overhead of migration is highly dependent on the underlying architecture, the characteristic of the victim threads and the difference between the source and the destined environment. Consequently, we would like to emphasize the followed points in this study:

1. This study is focused on higher performance at a given overhead, while it does not fall into our scope to reduce the unit overhead.
2. This study is based on the homogeneous architectures and thus the overhead is due to rescheduling and warming up.
3. The performance of the proposed schemes will be discussed with the overhead properly considered.

2.4 The Shared Resources and Workload Behaviors

In the followed context, we will first of all examine the shared resources. There have been several approaches to monitor the resources, such that the utilization was summarized. The second part is about the workload behaviors. The related studies have spent great efforts on analyzing the demands in different phases along execution, and among different workloads. The results paly an essential role in scheduling instructions and thread in the multithreaded architecture and the multicore architecture.

2.4.1 The Shared Resources

It provides better abstraction to divide the shared resources in the MMMP into two categories: computation resources and memory accessing resources [66]. The former one involves execution resources in the pipeline as well as the high-level caches. State-of-the-art design of fast cache is at similar speed with the CPU clock, meaning its latency is around $1 - 2$ CPU cycles. Therefore, the accesses to high-level cache belong to usage of computation resources. On the other hand, lower memory hierarchical systems are memory accessing resources, which include low-level memory hierarchy, the DRAM controller, the FSB and prefetch requests. In quantity, Zhu *et al.* [72] provided the Cycle Per Instruction (CPI) portions to express the usage. The overall CPI was decomposed into computing, L1, L2, L3 and main memory accessing. The sum of the first two parameters suggested the usage of computation resources, while others were considered as memory accessing resources. Sharing the same concept, the cache miss rate was used as the metric by Cazorla *et al.* [73].

2.4.2 Demands along Execution

From the perspective of threads, their resource demands can be described in such a categorization. Take cache miss rate as an example [73]: the threads with miss rate no less than 1% are considered to mainly utilize memory accessing resources, such that their performance is Memory Intensive (MI). On the contrary, the threads with cache miss rate less than 1% belong to the Computation Intensive (CI) category. This categorization is widely used in constructing multi-programming workloads. Depending on the benchmark category, a multiprogramming workload may be pure, *i.e.*, CI or MI, or blended, *i.e.*, BD. The similar result is available from summarizing misses over retired instructions, *e.g.*, Misses Per Kilo (1024) Instructions (MPKI)

in [32, 74, 45] and Misses Per Million (1024×1024) Instructions (MPMI). It follows the natural pace of the thread, rather than any specific execution environment. Thus, the MPKI or the MPMI are better at telling the thread resource demands in spite of the architectural specification. Considering execution phases [5], thread demands in terms of the MPKI or the MPMI partially shows phase behaviors along the execution of a single workload.

Duesterwald *et al.* studied the prediction along the execution of a single thread. Their designs were focused on the correlation among different performance metrics, *e.g.*, IPC and cache misses, and had no consideration for inter-thread interference [75]. Other categorizations were proposed as Colors [76] and Animals [77], which had more categories for the threads and thus more information was represented by the categorization. Furthermore, there were designs focused on the demands of threads [43, 77, 78, 79, 80], but their designs required *a priori* knowledge, and thus were not qualified for a real-time approach. For example, Chen *et al.* [80] involved deep profiling in the source code, searching for instructional dependency, data locality, instruction mixture and control flows.

2.4.3 Demands among Workloads

Pereira *et al.* [81] proposed to dynamically identify phase based on traces of the workloads. It obviously required analysis before the workloads were actually executed on the machine. Inspired by the Branch Predictor in the computer architecture, other studies [82, 83, 84] employed history patterns at different levels, *e.g.*, global or local, to predict the execution phases. The basic concept of their approaches was not adaptive enough for the execution phases, because the answer of a basic predictor is simply True or False. In order to further understand the workload demands, their approaches were associated with more hardware resources, such

as phase information tables, though its consistency with the prediction remains an open question.

The term Critical Section is mostly applied to parallel programmes, which is the serial part in the execution and defines the total execution time. Hollingworth [85] was the first to insert segments to the source code and monitor the program during execution to identify the critical sections. Recent studies such as Age-based approach [86] and Bottleneck Identification and Scheduling (BIS) [87] were also assisted by the source code, the library and/or the compiler in differentiating the applications. Even though prediction is proposed by Fields *et al.* [88, 89], their predictor was Program Counter(PC)-indexed and trace-based. Some other studies, *e.g.*, [90, 91], obtained the representative information of the underlying architecture, and then estimated the performance of the applications. Their employment of the *a priori* knowledge hardly helps apply their designs to various environments.

Ebrahimi *et al.* [41] identified the serial parts in the parallel applications, and increased their priorities in the memory scheduler, such that the application moves faster to the parallel parts that may be executed by multiple threads and/or cores. Cai *et al.* [40] relied on the hint instructions inserted into the source code, and thus the execution of a loop would trigger new scheduling decisions. However, they attached little importance to inter-program interference, which is more universal in both parallel workloads and multi-programmed workloads.

Therefore, inter-thread interference further increases the difficulty in utilizing execution phases for scheduling policies. Most of current studies rely on analysis in advance in order to predict during the actual execution. When *a priori* knowledge is not available, the prediction is shrunk to PC-based, so that the detailed characteristics of the execution phases cannot be easily unveiled. On the other hand, the focus on the critical section in parallel programmes underestimates the importance

of inter-thread interference, such that an adaptive and practical approach to identify execution phases is high desired.

2.5 Summary of the Related Work

In this chapter, we first of all examined the underling hardware in this study, which was the multicore multithreaded architecture. It introduced two distinguished impacts on the resources in an MMMP: unification and isolation. It meant the resources were shared by domestic threads within a core, but isolated among different cores. Meanwhile, the varying behavior of various workloads were spotted by a lot of researchers, and most of them would like to better use the execution phases to guide the scheduling in the MMMP. As a result, we recognized two major problems in the studied area: complexity in resource allocation and variation in workload behaviors.

In order to provide a solution to the problems, integration and adaptability will be proposed in this study. In this chapter, we have reviewed many related studies, and to our best knowledge, their designs were separated by the hardware and the software, or there has been no collaborative design. On the other hand, the existing scenarios to identify execution phases were either associated with off-line analysis, or weak at detailed information of the coming phases. Therefore, the proposed policies in this study will be an integrated approach, that employs both hardware and software efforts to cope with the complicated resource allocation in the MMMP; while they are also adaptive in identifying execution phases and telling more details about the phases.

CHAPTER 3

THE ONLINE LINEAR MODEL

Let us start the design from observing workloads of the MMMP. In general, phase behaviors are seen in most workloads at different granularity, but the repeated behavior is somewhere between ideally periodic and totally chaotic. Memory accesses are major phase behaviors and they show the essential demands of thread in scheduling policies [5, 46]. In practice, it is motivated by the observation in [47], in which a correlation coefficient of -0.4492 was obtained between L1 and L2 cache miss rates. Such coefficient hints an interesting relationship between L1 and L2 cache misses, which is neither linear, *e.g.*, coefficient = 1, nor unrelated, *e.g.*, coefficient = 0. It summons further investigation for better description, especially from a statistical perspective.

As a result, the Online Linear Model (OLM) regression is proposed in this chapter in an effort to better investigate into the relationship and the phase behavior. The proposed design relies on the Ordinary Least Square (OLS) Regression to construct the online model. Though there are a few mathematical approaches available in constructing models, efficiency is not guaranteed empirically for a sophisticated implementation. Hence, the estimation is started from the OLS regression in Chapter 3.1. Furthermore, the OLM is designed in Chapter 3.2, and its hardware engines are built to accommodate its features in Chapter 3.3. Optimization is conducted to reduce latency and complexity, while other overheads, such as power, area and storage, are considered as well. Summary of the OLM is written in Chapter 3.4.

3.1 The OLS Regression

Given that a random variable Y is a function depending only on another random variable X and their relationship is linear, they can be expressed as:

$$Y = \beta X + \alpha + \varepsilon \tag{3.1}$$

where ε is Gaussian distribution with zero mean and variance σ^2, while β and α can be evaluated as $\hat{\beta}$ and $\hat{\alpha}$ through the regression [92]. Therefore $\hat{\beta}$ and $\hat{\alpha}$ are evaluated:

$$\hat{\beta} = [\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})][\sum_{i=1}^{n}(x_i - \bar{x})^2]^{-1} \tag{3.2}$$

$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x} \tag{3.3}$$

where $\bar{x}$ and $\bar{y}$ are the mean values for X and Y respectively, x_i is for misses in the L1 DCache, y_i refers to the data requests that miss in L2 cache and n stands for the number of samples. The OLS regression is the best among all unbiased estimators in the sense of having the smallest variance [92, 93].

Furthermore, X and Y are both one-dimension matrices, or vectors, so the simple regression can be used to calculate the estimators:

$$\hat{\beta} = \frac{L_{xy}}{L_{xx}} = [\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}][\sum_{i=1}^{n} x_i^2 - n\bar{x}^2]^{-1} \tag{3.4}$$

And $\hat{\alpha}$ still uses Equation 3.3. ε is omitted from our model to achieve moderate simplicity in computation. The beauty of simple regression is to update factors accumulatively, rather than to re-calculate them completely, and thus it is feasible to reduce the hardware overhead greatly.

In reality, perfect linear relationship hardly exists, especially between L1 and L2 cache misses along execution, so significance test on the linear relationship may be used to determine the accuracy of the regression. The total sum of squares (S_T)

is composed of the residual sum of squares (S_e) and the regression sum of squares (S_R). As a result, the difference between the estimated values ($\hat{Y}$) and the original values (Y) is decomposed into two parts: one resulting from the linear relationship, *i.e.*, S_R, and one caused by other factors, *i.e.*, S_e.

$$F = \frac{S_R}{S_e/(n-2)} = \frac{\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2/(n-2)} \tag{3.5}$$

Given the hypothesis that there exists linear relationship, when the variation mainly results from the linear relationship, rather than other factors, the linearity hypothesis is true. Hence, the F value of the regression is used to test the significance level of the model quantitatively [92]. With the required significance level ρ, the above hypothesis holds when $F \gg \mathcal{F}_\rho(1, n-2)$ is true, where $\mathcal{F}_\rho(1, n-2)$ is the F distribution upon ρ, and n is the number of elements in Y.

3.2 Construction of the OLM

Two modules are needed to construct OLM:

1. **Sampling**: It collects cache misses information and interrupts other modules upon newly completed samples.
2. **Regression**: It performs an iteration of regression as soon as new samples are available. It also predicts according to the updated sample and the current model.

3.2.1 The Sampling Module

L1 and L2 data misses are collected in this module, while samples are formed for every Sampling Period (SP). The model is focused on the relationship between L1 DCache misses and the consequential L2 cache misses. Even if the L2 cache is

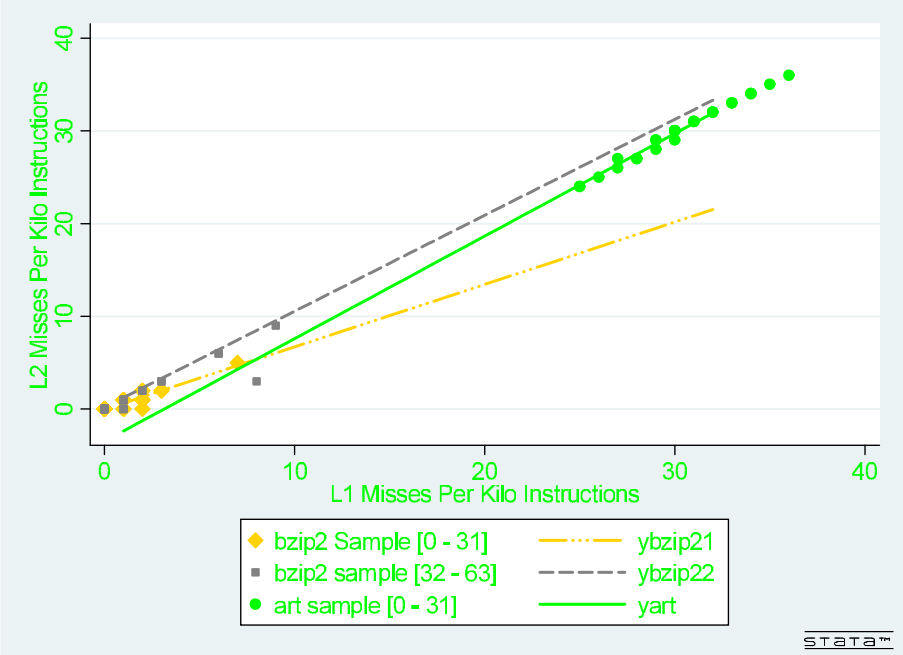


Figure 3.1: The Online Linear Model of cache misses

unified, we track the data request misses only. Here the Misses Per Kilo Instructions (MPKI) [32, 74] or the MPMI might be employed, where the SP is 1 kilo (1024) or 1 million (1024^2) instructions. We feel it is a better metric than miss rate in terms of quantitatively describing the pressure on the cache system caused by the thread. Two SPEC CPU2000 benchmarks [94]: *bzip2* and *art*, are taken as an example. Their MPKIs are collected in some arbitrary segments, which are shown in Figure 3.1.

3.2.2 The Regression Module

Given X as the L1 Data Cache (DCache) MPKI and Y as the data MPKI in L2 cache, the OLS regression is conducted using Equations 3.2 and 3.3 in this module. The number of samples employed in regression is denoted as the Window Size (WS) in this study, and {SP, WS} will be used to specify OLM's configuration. Prediction

is fulfilled using newly updated L1 data misses and estimators:

$$\hat{y} = \hat{\beta}x + \hat{\alpha} \tag{3.6}$$

where $\hat{\beta}$ and $\hat{\alpha}$ are estimators from the OLS regression, x is the current L1 MPKI, while $\hat{y}$ is the predicted value, *i.e.*, the L2 MPKI. Let's stick to the example in Figure 3.1, in which 32 samples are recorded in a row, *i.e.*, {1K, 32}. Consequently, there are three groups of consecutive samples, which are shown as the dots in Figure 3.1.

First of all, it is valid to employ the OLS regression for linear model, because they are proven by the significance test, *i.e.*, F test based on Equation 3.5. In detail, the *art* model has $F_{art} = 405.31$, while *bzip2* models have $F_{bzip2[0-31]} = 179.09$ and $F_{bzip2[32-63]} = 179.36$, which are all greatly larger than the corresponding F distribution, *i.e.*, $\mathcal{F}_{0.01}(1, 30) = 7.56$.

Furthermore, the linear model should be able to cope with diverse behaviors from different threads and respond to the phase changes within a single thread along execution as well. In this example, *bzip2* and *art* have different linear models, which are extracted by the OLS regression with different scopes and offsets. This shows one model does not fit all threads, and hence each thread needs to be evaluated separately. Furthermore, because *bzip2* changes its linear model in the second group of samples, *i.e.*, from the dotted to the dashed, its L2 MPKI would be different along execution, even if the L1 MPKI remain the same. This varying relationship in a single thread indicates that using a fixed model for a single thread along execution is still not good enough, but an on-line model that could self-adapt in real-time would be more favorable.

3.3 Hardware Implementation

In order to construct the OLM, two hardware engines are designed in computer architecture for their corresponding modules: **Sampling** engine and **Regression** engine. The engines are dedicated to the OLM and thus do not interfere with regular thread execution, *i.e.*, usually do not increase execution time of workloads. Nevertheless, the native regression is computation intensive by nature and thus not efficient, so better designs are explored through simple regression and some other optimizing methods. They lead to two different designs: the OLMn based on the native OLS regression and the optimized solution OLM. Such optimization will be inherited by other components as the default policy, while those policies without optimization, *i.e.*, based on the native algorithm, are referred by the same postfix "n", *e.g.*, the OLMn. Considering the overwhelming overhead in the OLMn, we would focus on the optimized version, while details of native implementation are omitted.

3.3.1 The Sampling Engine

The **Sampling** engine would be similar to that in the DWarn [24], which collects the cache information constantly. There are thread-specific counters for each of the two-level caches, which are increased by one as a new miss happens within an SP but are reset at the beginning of a new SP. And updating or resetting is supposed to take no more than one CPU cycle. Nevertheless, because the engine is an independent hardware, its latency has no interference with downstream functions.

3.3.2 The Regression Engine

The **Regression** engine is responsible for some logic operations, so logic units are designed for our proposed scheme in Table 3.1, where the initial "i" defines an integer

Table 3.1: Architecture units in Regression engine

Unit	iALU	iShftr.	iMul.	fALU	fMul.	fDiv.
OLM	1	1	1	N/A	N/A	N/A
OLMn	2	2	2	1	1	1
Purpose	$\pm$	$\times$ & $\div$	$\times$	$\pm$	$\times$	$\div$

unit and "f" is for FP units. Control logic and data path are omitted here, because they are highly dependent on the customized implementations, *e.g.*, FPGA. The updating of regression engine and the linear model is triggered by a new sample, which is served on a First-Come First-Serve (FCFS) basis. Time to read samples from counters and registers does not incur extra computation latency here because it happens in parallel with the regression, so the **Regression** engine contributes most latency of the OLM construction. Consequently, our efforts to OLM's hardware will be focused on the **Regression** engine.

An essential advantage of the OLM over the OLMn is to employ simple regression, *i.e.*, Equations 3.3 and 3.4, so optimization in the **Regression** engine targets at: cumulative updating and simpler computing.

Firstly in our optimization, updating in the OLM is cumulative to reduce overall overhead. Given Equation 3.4, its numerator and denominator are both enlarged to ensure precision in integer division:

$$\hat{\beta} = \frac{n \times L_{xy}}{n \times L_{xx}} = [n\sum_{i=1}^{n} x_i y_i - (n\bar{x})(n\bar{y})][n\sum_{i=1}^{n} x_i^2 - (n\bar{x})(n\bar{x})]^{-1} \tag{3.7}$$

Assuming the current samples for X and Y are from 0 to 31, and now we have new samples x_{32} and y_{32}:

$$n(\sum x_i y_i)' = n(\sum x_i y_i - x_0 \times y_0 + x_{32} \times y_{32}) \tag{3.8}$$

$$(n\bar{x}')(n\bar{y}') = (\sum x)'(\sum y)' = [\sum x - x_0 + x_{32}] \times [\sum y - y_0 + y_{32}] \tag{3.9}$$

$$n(\sum x_i^2)' = n(\sum x_i^2 - x_0 \times x_0 + x_{32} \times x_{32}) \tag{3.10}$$

$$(n\bar{x})'(n\bar{x})' = (\sum x)'(\sum x)' = [\sum x - x_0 + x_{32}] \times [\sum x - x_0 + x_{32}] \tag{3.11}$$

where prime, *e.g.*, $(\bar{x})'$, refers to the newly updated value and n is the WS in our scheme. Because we choose a WS as multiple of 2, multiplication and division involving n is actually shifting.

Secondly, the division in Equation 3.4 is replaced by shifting and rounding, such that there is neither FP data nor division in OLM:

$L_{xy}[i] \leftarrow L_{xy}[i] \times 1024$ {Empirically magnify to keep details}
if $2^{m+1} > L_{xx} \geq 2^m$ {Round to the nearest multiple of 2} **then**
  if $(2^{m+1} - L_{xx}) \geq (L_{xx} - 2^m)$ **then**
    $L_{xx} \leftarrow 2^m$
  else
    $L_{xx} \leftarrow 2^{m+1}$
  end if
end if
$\hat{\beta} \leftarrow \frac{L_{xy}}{L_{xx}}$ {Shift right by m or $m+1$ to replace the division}
$\bar{y} \leftarrow \bar{y} \times 1024$ {Align $\bar{y}$ with magnified $\hat{\beta}$}
$\hat{\alpha} \leftarrow \bar{y} - \hat{\beta} \times \bar{x}$

3.3.3 The Hardware Overhead

Table 3.2: Storage for OLM

Unit	Counter	Register	Register
OLM	16×TN	16×WS×TN	32×2×TN
OLMn	20×TN	20×WS×TN	32×2×TN
Purpose	MPKI	Samples	$\hat{\beta}$ and $\hat{\alpha}$

Since we are monitoring the MPKI in the **Sampling** engine, 10-bit counters would be enough to store samples. However, as the MPKI for most benchmarks

studied are under 150 [74], we employ 8-bit saturating counters to reduce the hardware overhead even further. Similarly when the MPMI is employed, 20 bits are certainly enough, while 16-bit saturating counters are implemented as a balanced design. The counters and registers in the **Sampling** engine are specified in Table 3.2, where TN means the number of threads. In the table, the capacity of counters and registers is evaluated by the number of bits.

After the optimization, latency in **Regression** engine is 79 cycles. The latency is not dependent on the WS any more due to cumulative updating, except for a WS that is not power of 2. Those irregular WSs would raise the total time by involving real multiplication instead of shifting, but it can easily be avoided at the design stage, *e.g.*, it will not hurt to choose 32 rather than 29 or 37 for the WS. The detailed breakdown is shown in the left part of Table 3.3, with the latency for architecture units from Hennessy *et al.* [18]. Although it is greatly improved to 26% of the OLMn, shown in the right part of Table 3.3, we ought to account for the latency as realistically as possible. We assume that new model cannot be available until one SP after the update of a new sample. Empirically, a single thread normally has a sustained Instruction Per Cycle (IPC) around 1. In this case, with the SP set to 1024, it actually sets aside more than 1024 cycles in order for the computation associated with the regression to complete, which to our knowledge is adequate.

In addition to the storage and latency, power and area should be considered in the hardware design. Therefore, this study need to further examine these two in overhead evaluation, and it will follow the approach in similar hardware-based machine learning algorithm [95]: to scale existing units to our specific design. Hickmann *et al.* [96] implemented a 64-bit fixed-point multiplier at 110nm CMOS technology, which area was estimated as 0.65mm^2. Linearly scaled down to 32-bit and 65nm, our integer multiplier's area is 0.113mm^2. Given the integer multiplier in the **Regres-**

Table 3.3: Latency in Regression engine

OLM		OLMn	
Factor	Cycles	Factor	Cycles
$n\sum x_i y_i$	17	$\bar{x}$ & $\bar{y}$	4
$(n\bar{x})(n\bar{y})$	11	$(x_i - \bar{x})$ & $(y_i - \bar{y})$	32
$n\sum x_i^2$	17	$(x_i - \bar{x})(y_i - \bar{y})$	–
$(n\bar{x})(n\bar{x})$	7	& $(x_i - \bar{x})(x_i - \bar{x})$	224
$\hat{\beta}$	9	$\hat{\beta}$	24
$\hat{\alpha}$	10	$\hat{\alpha}$	11
Prediction	10	Prediction	11
Total	79	Total	306

sion engine is the most complex unit, the OLM's functional units are implemented at most in an area of 0.339mm^2, which is 0.17% of a 200 mm^2 chip. On the other hand, given the FPU in IBM Power6 spends 0.56W/mm^2 at 1.1V and 4GHz [97]. The functional units in the **Regression** engine would consume up to 0.19W at full utilization. Please note above estimation applies the data of the multiplier to the integer ALU and the shifter and ignores our design of updating upon interruption. Hence, compared with some similar hardware schemes and other software schemes, the OLM is certainly promising in terms of hardware overhead even though it is regression-based.

3.4 Summary of the OLM

In this chapter, we have traveled through the setup of the OLM, which is embedded in the architecture level to analyze workload behaviors. Linearity is the target of the model because of the natural of the regression algorithm, such that L1 and L2 data misses of every workload are sampled and then used to compute the estimators, *i.e.*, $\hat{\beta}$ and $\hat{\alpha}$. They work together to describe the linear relationship between L1 and L2 data misses, and cope with the varying and various phase behaviors. In particular, the model is able to provide two sets of information to other parts in the system:

the estimators and the predicted LLL, *i.e.*, $\hat{y}$. The detailed role that OLM plays in the whole design will be addressed clearly in later chapters.

CHAPTER 4

THE REGRESSION-BASED ALGORITHM TO PRIORITIZE THREADS

In the previous chapter, the OLS regression is employed to build the OLM, which describes the relationship of misses within two-level caches adaptively. Via the linear model, the system is able to observe and predict the thread behavior, which reflects its resource demands as well. Resource management in both multicore and multi-threaded environment will benefit from such a model. Considering the hardware scheduling policies in the SMT architecture as discussed in Chapter 2, some of them take the action at the occurrence of L1 data misses, expecting it should be better at controlling the LLL in the pipeline, but they do not have an accurate observation on the relationship between L1 data misses and the LLL. Other resource management schemes employ OS in conducting their algorithms, such that they introduce great overhead to the system by interrupting OS and transmitting data. The goal of this chapter is to utilize the statistical model from the OLM to guide the instruction fetch policy in the SMT architecture, while hardware overhead is minimized to its best efforts.

As a result, the Regression-based Algorithm to Prioritize Threads (RAPT) is proposed, in an effort to manage the shared execution resources for minimal contention and optimal performance. In Chapter 4.1, three modules will be explained for the RAPT, which take the responsibility including building up models and prioritizing multiple threads. Hardware engines are implemented for the modules, and two of them are exactly inherited from the OLM to explore its merits. Overview of the RAPT's overheads, including those introduced by the extra engine, are summarized in Chapter 4.2. The basic experimental methodology is explained in Chapter 4.3. Performance is compared to other similar instruction fetch policy to validate

RAPT in Chapter 4.5.1, while the sensitivity analysis is done in Chapter 4.6. Brief summary is drawn at the end of this chapter.

4.1 The Three-Module Design

To explore the critical and variant relationship between L1 and L2 data misses for better resource management in SMT processors, RAPT is proposed with three modules:

- **Sampling**: L1 and L2 data misses, *e.g.*, MPKI, are collected for regression.
- **Regression**: The linear model is constructed by the OLS regression, such that future L2 MPKI are predicted according to current L1 data misses.
- **Prioritization**: Higher priority is assigned to the thread(s) with smaller predicted L2 MPKI.

Then, the priority is submitted to fetch engine, which is responsible for really fetching instructions in the SMT architecture. The first two modules actually set up a linear model for every thread, and they coincide the two modules in the OLM. In another word, RAPT utilizes the adaptive models through embedding the OLM in its design. Hence, the **Sampling** and **Regression** modules in RAPT will follow the explanation in Chapter 3.3.1 and Chapter 3.3.2, respectively.

Regarding **Prioritization** module, it grants priority to the domestic threads in accordance with their predicted L2 MPKI from the previous module. In particular, threads with more L2 MPKI are assigned low priority in the engine, such that their instructions have lower probability to enter the pipeline. The motivation here is to reduce the occurrence of the LLL in the pipeline, which occupy the shared resources inefficiently. This module only defines the priority among different threads for fetching, and is widely adopted by researchers, *e.g.*, [17, 21, 22, 23], but it is emphasized

here that interpolation is employed instead of complete sorting. Upstream modules will notify the changed values, which are then put into the priority queue after several rounds of comparison. In such a case, the complexity of this module is up to $O(n)$, and number of threads is quite limited in most studies. Therefore, latency of the **Prioritization** module is not the major concern in this design.

4.1.1 The Inherited Engines

According to the three modules, RAPT scheme physically is composed of three engines: **Sampling**, **Regression** and **Prioritization**. RAPT is proposed closely connected with the OLM, and thus the first two engines are actually inherited from the OLM in Chapter 3, where revision is not required in RAPT. Similarly, the RAPT and the RAPTn both exist in the study due to the OLM and the OLMn, while RAPT is certainly the focus. Therefore, only the **Prioritization** engine need be further designed here.

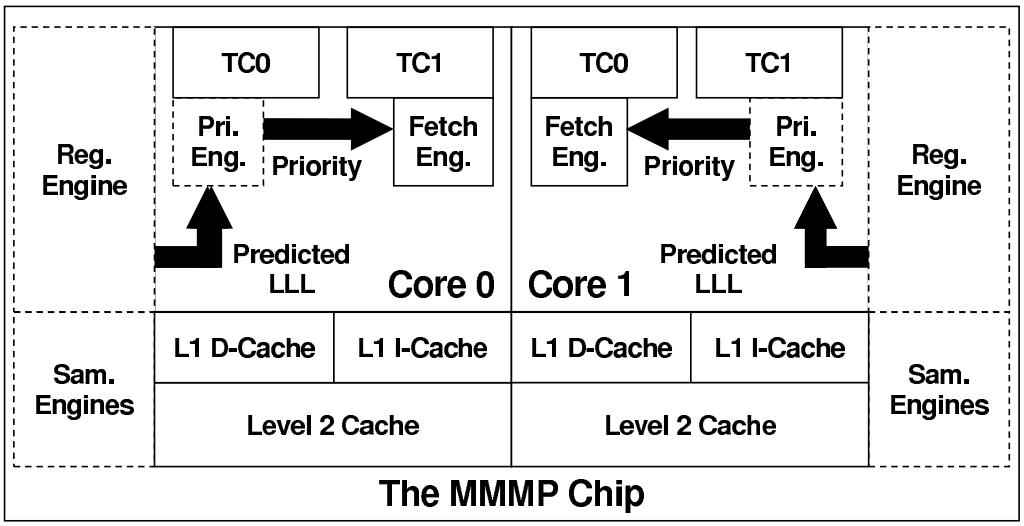


Figure 4.1: Designed hardware engines for RAPT. The Sampling (Sam.) engines are duplicated to every thread while two others are associated to a core. The Regression (Reg.) engine is shared by threads due to asynchronous updating, while the unique Prioritization (Pri.) engine concludes priorities for all domestic threads.

4.1.2 The Prioritization Engine

The **Prioritization** engine is a new engine, which takes results from **Regression** engine and then compares them for the priority at the fetch stage. The basic logic unit needed for interpolation is a comparator, which is able to indicate the larger input between two. It is an integer unit with width of 32 bits, such that the updated predicted values are compared with an sorted queue for prioritization.

In summary, a two-core four-threaded microprocessor is amended to illustrate the overview of the proposed architecture in Figure 4.1. Every thread is equipped with a **Sampling** engine, while the **Regression** and **Prioritization** engines are shared by all domestic threads. We do not claim fetch logic as the major contribution of our research, which has already been studied in other fetch policies, *e.g.*, [11, 22]. The scenario in the fetch engine is to fetch from higher priority to lower priority. It increases expected instructions in the pipeline and ensures the utilization of hardware resources. Regarding the fairness in the RAPT, as more instructions are fetched from the threads with higher priority, their real-time L1 data misses may increase. As a result, their predicted L2 MPKI may grow even if there is no change in their estimators. Therefore, it is less likely for them to stick to higher priority in future fetching, and thus the fairness among threads is ensured.

4.2 Summary of the Hardware Overhead

The total overhead of the RAPT is specified here as an independent scheme. Although it integrates some engines from the OLM, their overheads belong to the RAPT and should be considered in evaluating RAPT's performance. This action will validate the RAPT as a completed and reusable scheme for SMT professors, and the overheads are listed in Table 4.1, where TN is the number of threads. To

Table 4.1: Total RAPT overhead

Engine	Item	Specification
Sampling	Counter	$2 \times 16 \times TN$ bits
	Register	$2 \times 16 \times WS \times TN$ bits
Regression	ALU	Integer
	Shifter	Integer
	Multiplier	Integer
	Power	0.190w
	Area	0.339mm^2
	Register	$2 \times 32 \times TN$ bits
	Latency	79 cycles/model
Prioritization	Comparator	Integer
	Power	0.063w
	Area	0.113mm^2
	Latency	$O(TN)$

our knowledge, we believe such a table reflects RAPT's theoretical overhead comprehensively, but data path is not reflected in the table. We acknowledge that data path consumes power, occupies area and needs intermediate registers, but such evaluation has covered the dominant overheads so far. Nevertheless it requires further implementation, *e.g.*, on FPGA, to unveil more details, which might not fall into the scope of this study.

4.3 Experimental Methodology

In order to examine the performance of the proposed schemes, an architectural simulator Super ESCalar (SESC) [98] is employed to implement the design. It is able to provide various information for our analysis, and we are mainly focused on the system throughout. In this chapter, the simulator and the performance measurement metrics are explained, which are valid for the rest of the manuscript, unless specified.

4.3.1 The Architectural Simulators

The motivation of a simulator is to reduce the manufacturing cost and speed up the related research. Fundamentally, architectural simulators are software executed on a hardware platform. They emulate the functionality of the proposed architecture through executing some representative workloads and provide various information from the simulation. The major goal of a cycle-accurate simulator is to summarize the execution time of the simulated workloads in terms of CPU cycles, while other data, such as cache misses and branch prediction, might be available as well. In particular, the SESC simulator is an open-source software in C++ and runs in a Unix/Linux environment, which certainly belongs to such a category [99].

Overall, the SESC simulator has two parts: an emulator and a timing model: the instructions from the workloads are executed by the emulator to generate the necessary patterns of the program, and then the generated patterns are submitted to the timing model for timing evaluation. The emulator in the SESC is MINT [100], which is an emulator for the Microprocessor without Interlocked Pipeline Stages (MIPS) Instruction Set Architecture (ISA)[4]. Hence, the basic architecture in the SESC is essentially derived from the Reduced Instruction Set Computing (RISC) [101, 102]. The SESC simulator follows a five-stage-design: Fetch, Decode, Issue, Execution and Retirement. The timing model is composed of virtually a lot of parameters in the architecture. For example, the latencies to different levels in the memory hierarchy. As a result, the target architecture is implemented by modifying the emulator and the parameters in the timing model, such that the impacts could be shown in the final report.

Given there had been many architectural simulators, the motivation to develop the SESC simulator was to implement an understandable environment for the parallel architectures [103]. It attached much importance to fidelity, performance, modi-

fiability and feasibility. Together they address these topics: the difference between simulation and real implementation, the speed of simulation, the difficulty to revise the architecture and the capability to accommodate a proposed design. Hence it provides a good support for most stakeholders in architectural research. Furthermore, the superscalar out-of-order pipeline is well implemented in the SESC simulator, which is critical in recent development and might be less emphasized by some other simulators. Such a capability matches our focus on the multicore multithreaded architectures, so the SESC simulator is a good fit for the research in this study.

4.3.2 The Workload Organization

Given the simulator is considered virtually as the proposed microprocessor, the basic way to validate it is to run some workloads and to compare its performance with reference architectures. It is not practical to run the full combinations workloads, and thus benchmarks are provided to represent the dominant situation in the expected environment [4]. There are usually two ways to organize the multithreaded workloads: parallel workloads and multi-programming workloads. Parallel workloads, such as Splash [28], SPECjbb [29] and RUBiS [30], are able to generate multiple threads with communications among the threads. Multi-programming workloads are composed of independent threads from different benchmarks, so there is no inter-thread synchronization in these workloads. The inter-thread communications may offer great potential for performance improvement, since the latency could be greatly reduced by clustering those threads.

Our work, however, is focused on the multi-programming workloads, which face more challenging issues in speeding up the whole system compared to the parallel workloads. As a result, workloads employed in this work are of multiple SPEC CPU2000 [94] benchmarks, *e.g.*, 2, 4, 6 and 8, and they are listed in Table 4.2.

Their type may be either Floating Point (FP) or Integer (INT), and their category are specified by Cazorla *et al.* in [22] based on their miss ratios in the LLC. In particular, the benchmarks with a miss ratio less than 1% are considered as CI, while other benchmarks are MI. Due to the "Memory Wall" issue, CI benchmarks might have more throughput than MI, but there is limited ILP in the long run [19]. Given the utilization of multiple levels of parallelism in the MMMP, it is against such a nature to have any biased preference solely defined by statistical throughput. In other words, a natively biased approach will not lead to satisfactory performance [17].

Table 4.2: The SPEC CPU2000 benchmarks employed

Benchmark	Type	Category	Benchmark	Type	Category
301.apsi	FP	CI	*164.gzip*	INT	CI
179.art	FP	MI	*181.mcf*	INT	MI
256.bzip2	INT	CI	*197.parser*	INT	MI
186.crafty	INT	CI	*171.swim*	FP	MI
183.equake	FP	MI	*300.twolf*	INT	MI
176.gcc	INT	CI	*168.wupwise*	FP	CI

4.3.3 The Performance Measurement

The average Instruction Per Cycle ($avgIPC$) used in [94] is one critical method to measure the overall system throughput, which is defined as the total number of instructions executed over the time elapsed. The formula for $avgIPC$ is:

$$avgIPC = \frac{1}{N}\sum_{i=1}^{N} IPC_i \tag{4.1}$$

where i is thread identity and N is the number of threads.

Nevertheless, in justifying the performance of the new architecture for a multiprogramming workload, Sazeides *et al.* [104] proposed the Average Baseline Weighted

IPC ($abwIPC$). It is calculated as the average thread improvement in the new architecture over the old architecture, which is a good reference to observe the average thread performance. The formula of $abwIPC$ is:

$$abwIPC = \frac{1}{N}\sum_{i=1}^{N}\frac{IPC_{new,i}}{IPC_{baseline,i}} \tag{4.2}$$

where i is thread identity and N is the number of threads as well. Moreover, due to its consideration of the improvement in every thread, $abwIPC$ is good at telling a biased approach in the scheduling policy. Assuming there is a scheduling that achieves better performance by favoring high-throughput threads, the performance of other threads are harmed, so such a scheme cannot show promising result in terms of $abwIPC$.

Therefore, fairness of the proposed scheme is well indicated by such two metrics. Although it is a common practice in architectural studies to look at the variance among the threads, the average variances of thread IPCs will be presented for further reference. As a result, these metrics will be used to measure the performance of the studied policies in the followed contexts.

4.4 Experimental Results

We would like to present readers the experimental results as soon as a design is finished, *i.e.*, RAPT and HASP. They will be compared with their peer policies to show their achievement by performance results. It is a careful approach to examine them separately, because it will be difficult to find any similar scheme for comparison with the ATMS. Hence, the aforementioned simulation methodology and performance measurement is adopted for the followed analysis.

4.5 Implementation Details

RAPT employs the SP = 1024 instructions and the WS = 32 samples, which is denoted as {1K, 32}. With the configuration in Table 4.3, the architecture now is augmented with the SMT ability, *i.e.*, two threads, four threads and six threads. Overall the main memory latency is to model 100 nanoseconds in an experimental processor running at 5GHz minus bus latency. The reason that we do not examine more than six threads is that performance degradation due to severe cache conflicts may prevent SMT processors from being equipped with a great quantity of threads. Instead, the SMT may be combined with multicore architectures to achieve less confliction and better parallelism. This opinion is also well studied in [22, 105, 106].

The multi-programming workloads are listed in Table 4.4. The workloads might consist of only CI or MI threads, such that they are CI or MI workloads respectively, while BD workloads have 50% CI threads and 50% MI threads.

The early simulation points are employed [107], and every thread is simulated for 100 million instructions in the representative regions, which is enough for major phase shifts at granularity of millions of instructions level [5]. The selected instructions are considered representative for the whole benchmark, such that the results would tell the execution of the workloads, no matter the actual duration of the execution in the future [107, 108]. Hence, it overcomes the limitation of simulation speed in an architectural simulator. The similar simulation methodology, *e.g.*, benchmarks, multi-programming workloads and/or representative regions, is also adopted by other researchers, such as [17, 21, 22, 23, 24, 25, 50].

The ICOUNT [17] policy is used as the baseline scheme. Several other fetch policies that utilize the cache miss for prioritization and decision-making are employed for comparison, which include the STALL [21], DG [23] and DWarn [24]. We try

Table 4.3: The baseline parameters

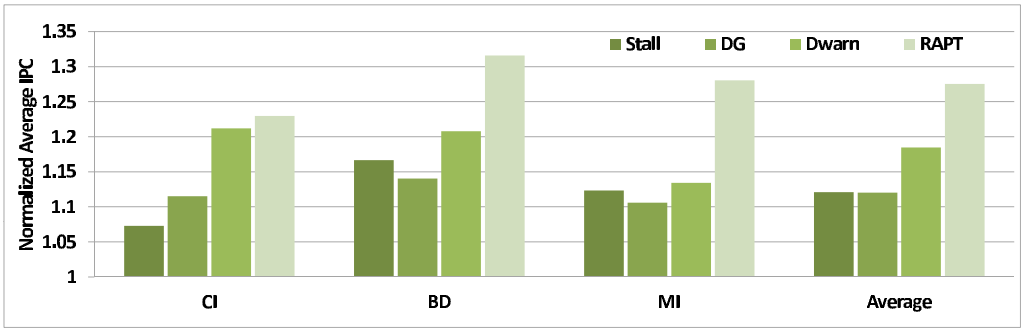
Parameter	Value
IF/IR Width	8/9
ReOrder Buffer Size	320 entries
Inst. Window	160 INT, 64 FP
Function Unit	3 Ld/St , 5 FP Mul/Div
	3 INT Mul/Div
L1 DCache	32KB, 4-way
L1 ICache	32KB, 4-way
L1 Cache Hit	2 cycles
L2 Cache Hit	9 cycles
L2 Cache	512KB, 8-way asso.
Main Memory Hit	469 cycles

Table 4.4: The workloads in simulation

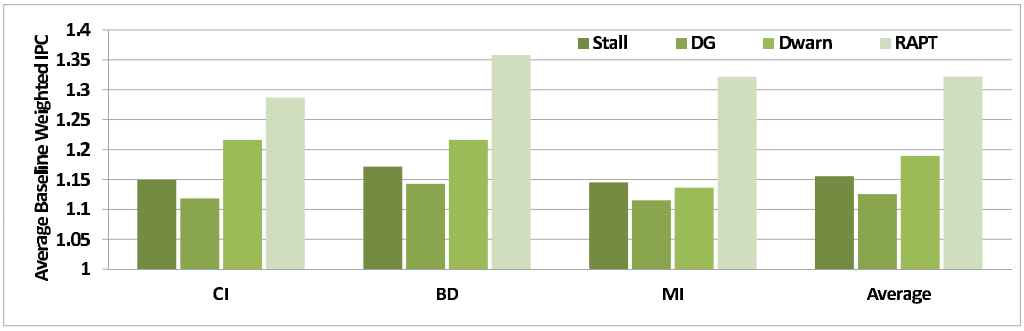
Thread #	Cty	Benchmark List
two	CI	*gcc, crafty*
		gzip, bzip2
	BD	*gcc, parser*
		gzip, twolf
	MI	*parser, twolf*
		mcf, twolf
Four	CI	*wupwise, apsi, gzip, gcc*
		gzip, bzip2, crafty, gcc
	BD	*gcc, bzip2, swim, art*
		gzip, twolf, bzip2, mcf
	MI	*swim, art, equake, mcf*
		mcf, twolf, equake, parser
Six	CI	*wupwise, apsi, gzip, gcc, crafty, bzip2*
	BD	*gzip, wupwise, crafty, mcf, twolf, parser*
		gcc, bzip2, apsi, swim, art, equake
	MI	*swim, art, equake, mcf, parser, twolf*

our best to construct these policies and we believe they provide similar results as they were originally presented. For example, the STALL in our simulation generates almost the same performance improvement over the ICOUNT in two-threaded and four-threaded workloads.

4.5.1 Performance Achievement



(a)



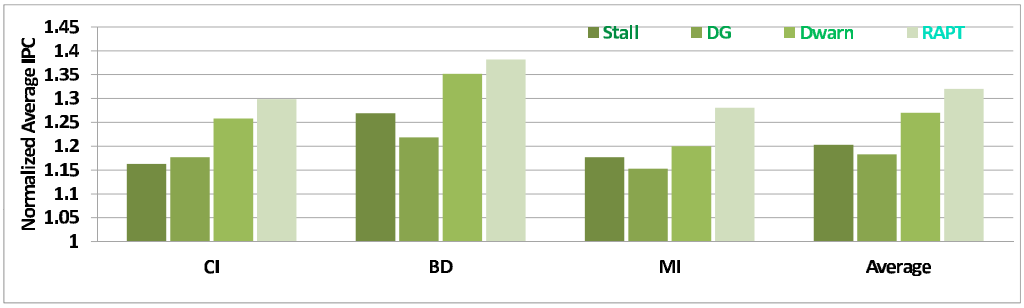
(b)

Figure 4.2: Overall performance improvement (a) $avgIPC$, (b) $abwIPC$

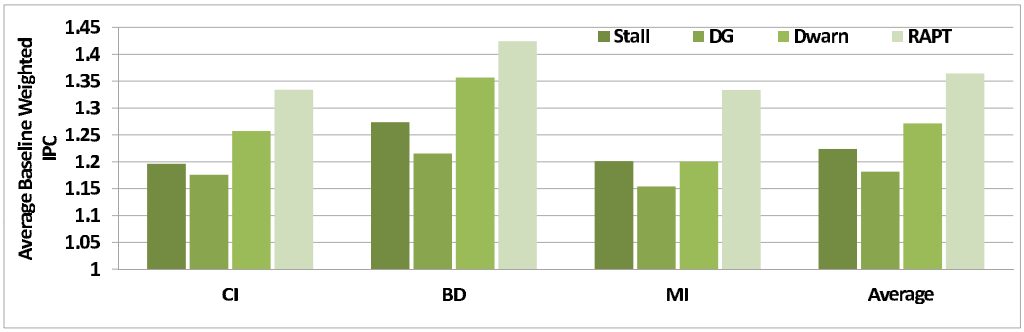
The overall performance improvement is shown in Figure 4.2. The $avgIPC$ of other policies are normalized to that of the ICOUNT, such that there are only five policies shown in the figures. In different configurations, the average results are presented if there are more than one workload belonging to the same category, *i.e.*, CI, BD and MI.

Overall, RAPT improves the performance over the ICOUNT by 28% with respect to $avgIPC$ and 32% with respect to $abwIPC$ as shown in Figure 4.2(a) and 4.2(b), which are obviously better than any other polices in the study. This is because the RAPT takes the correct action with proper timing. It does not follow a fixed doctrine, but rather extracts the real-time relationship between L1 and L2 MPKI from a statistical point of view. It then makes decision adaptively. Whenever

the thread changes its execution phases, RAPT is able to catch any slight change and update the model correspondingly. Thus the most suitable decision would be made properly. As a result of the appropriate prioritization, the RAPT is able to better reduce inefficient occupancy and contention for the shared resources, and thus perform better than other policies.



(a)



(b)

Figure 4.3: The performance in two-threaded workloads (a) $avgIPC$, (b) $abwIPC$

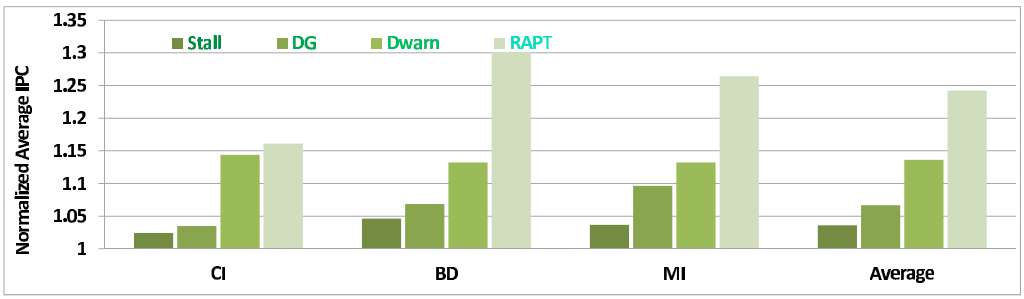
With the RAPT, the SMT processor will fetch fewer instructions from the thread with more expected L2 MPKI. Because when a load misses in the L2 cache, the thread itself can rarely progress anyway. Hence, no noticeable performance reduction is introduced. Correspondingly, the RAPT assigns higher priority to threads with less expected L2 MPKI, which are still able to move forward in the processor. Consequently, the RAPT improves average thread performance greatly. That is why we observe more improvement in $abwIPC$ than in $avgIPC$.

The RAPT exceeds the STALL by 13.7% in $avgIPC$ and 14.4% in $abwIPC$ on average. When L2 data miss happens, in the system a lot of shared resources are occupied by the thread with little throughput. Even though STALL may work well to prevent that thread from further introducing more LLLs into the system, it rarely helps current situation. On the contrary, the RAPT acts before the L2 data miss actually happens by reducing the priority of those threads with more expected L2 MPKI. Therefore, it is able to minimize their influence in advance, rather than patch up after it happens.

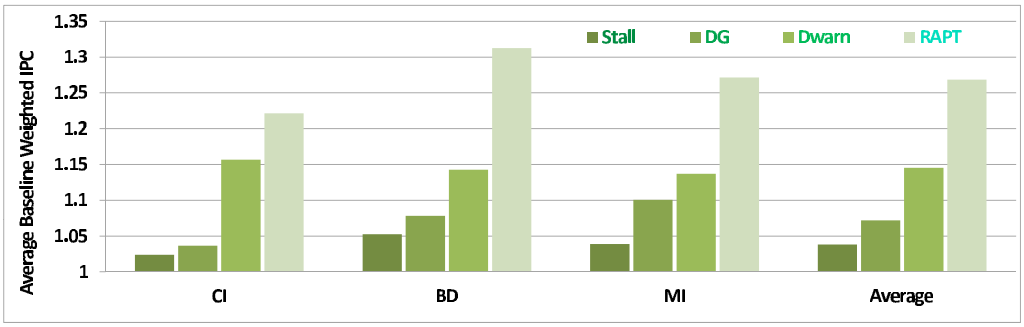
DG takes action more aggressively than STALL. It gates the thread with unsolved L1 cache miss. Nevertheless, the models between L1 and L2 MPKI may be different among threads and along the execution of a single thread, such that gating in DG might be too aggressive in some cases. Hence, we do not take this approach when we design RAPT. Instead RAPT adjusts its priority assignment according to forecast L2 MPKI. As a result, the RAPT outperforms the DG by 14% in terms of $avgIPC$, and 17% in terms of $abwIPC$.

Sharing the same concept that L1 data miss might lead to L2 data miss, the DWarn and RAPT both take action as L1 data miss happens. the DWarn takes a heuristic approach, which actually undermines a complicated relationship. RAPT does not advocate a constant linear model, because applications vary and their cache behaviors change too. Hence, RAPT sets up the real-time model between L1 and L2 MPKI adaptively, and obtains the estimated L2 MPKI. Consequently, the prioritization matches the execution phases better than those based on heuristic assumption. As a result, the RAPT outperforms the DWarn by 8% in terms of $avgIPC$ and 11% in terms of $abwIPC$.

Regarding the performance with different numbers of threads, the RAPT keeps similar improvement over the ICOUNT as shown in Figure 4.3 – 4.5 for the multi-



(a)

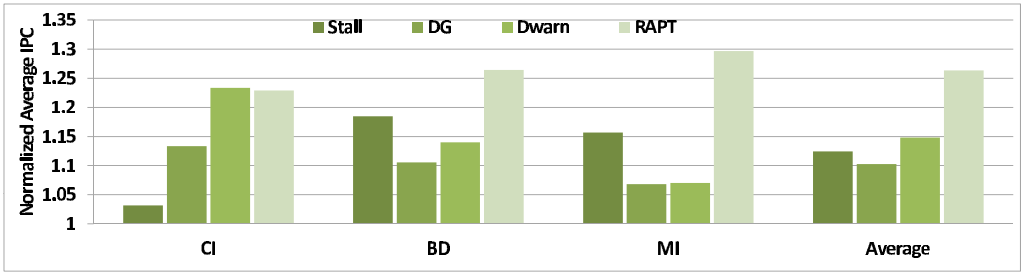


(b)

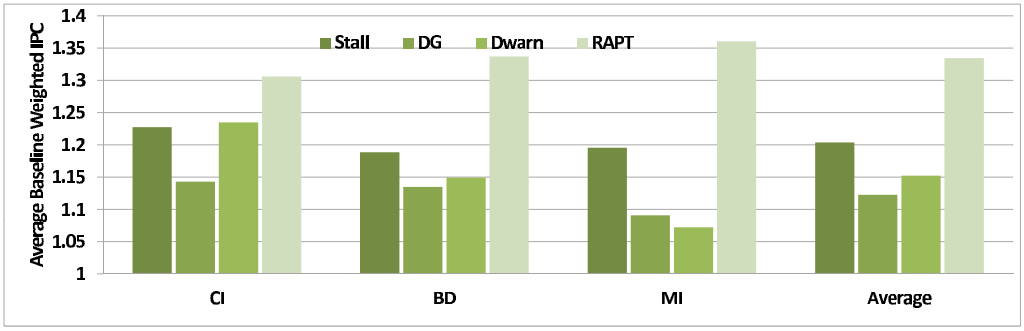
Figure 4.4: The performance in four-threaded workloads (a) $avgIPC$, (b) $abwIPC$

programming workloads. It is because the improvement is naturally from the cache behavior of benchmarks. As long as there exists significant linearity in the cache behavior, the RAPT is able to build the statistical model adaptively and prioritizes the threads correspondingly. Moreover, we note that BD workloads provide best overall results over CI and MI workloads. We assume it is because minimal resource contention is achieved when there is diversity in terms of resource demands in the processor [46, 47]. The RAPT is able to coordinate thread execution according to their resource demands adaptively, such that the shared resources are better utilized with less contention for them. Nevertheless, in CI and MI workloads, such potential is relatively small because threads may show similar demands statistically, so RAPT does not produce more throughput in CI and MI workloads than in BD workloads.

Fairness of the RAPT is validated by its superior $abwIPC$, which is the average improvement of performance in the new architecture over the baseline architecture.



(a)



(b)

Figure 4.5: The performance in six-threaded workloads (a) $avgIPC$, (b) $abwIPC$

We advocate that it suggests a critical aspect of the fairness in performance, which is the fairness of the improvement. However, regarding the variances of the IPCs, it is naturally defined by the difference among different threads, *i.e.*, threads may have differently sustainable pace in the execution. Moreover, it is also closely associated with $avgIPC$, meaning a large $avgIPC$ is sometimes accompanied by a large variance. Even though they are not so indicative as the $abwIPC$, the average variances of the IPCs are presented in Table 4.5, where TN is the number of threads.

Table 4.5: The Average Variances of the IPCs

TN	ICOUNT	Stall	DG	DWarn	RAPT
2	0.00067	0.0070	0.00041	0.028	0.023
4	0.00015	0.00015	0.00019	0.00055	0.0037
6	0.000063	0.0079	0.00023	0.00012	0.0025

One interesting phenomenon we observe is that as the number of threads in the workload increases, the performance of fetch policies based on L1 cache miss such as the DG and the DWarn get worse even than the STALL for MI workloads. This is because for the SMT architecture, associated with the high cache access demand from the MI workload, the ratio of cache conflict increases significantly as the number of threads increases, especially for the small-size L1 cache. That is why those fetch policies based on L1 cache miss will not be able to indicate the LLLs as effectively as in the two-thread workload scenario for MI workloads. On the other hand, the STALL, which is solely based on L2 cache miss, outperforms them. The performance of the RAPT is not interfered by the blurred L1 cache misses because of its adaptive nature and still demonstrates its superiority.

4.5.2 Prediction Expectation of RAPT

In order to examine the linear model in our proposed policy, we collect the Prediction Expectation (PE) of every thread in execution, which is calculated as predicted L2 MPKI over sampled L2 MPKI.

$$PE = \frac{\hat{y}}{y} = \frac{\beta \times x + \alpha}{y} \tag{4.3}$$

To have better observation on the overall situation, we calculate the PE if and only if there is a new complete sample, instead of in every CPU cycle. Therefore, given 100 million instructions are simulated, there are approximately 97656 PE values in record for every thread in every workload. We gather the data specified to benchmarks, in spite of different workloads, and the average value and the Standard Deviation (StDev) are calculated. Furthermore, there is also the average PE over all the benchmarks, shown in Figure 4.6, where the overall StDev is for the mean values of different benchmarks, rather than the average of all deviations.

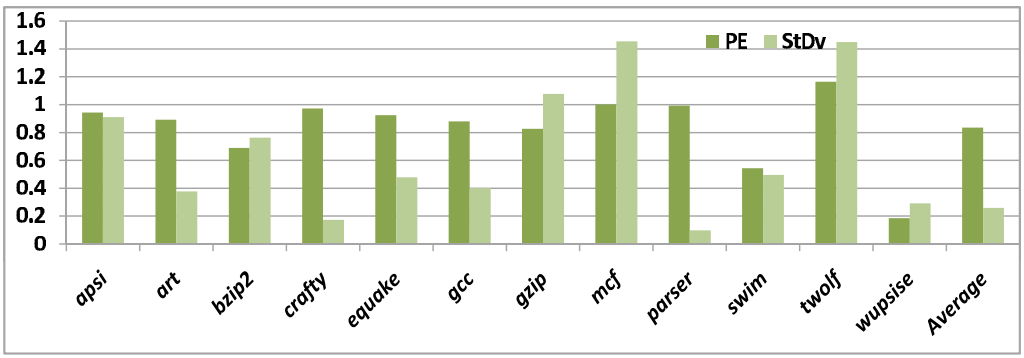


Figure 4.6: The prediction expectation of the studied benchmarks

On average, benchmarks have the PE = 0.83, while the StDev among the averages of benchmarks is 0.258. Hence, the linearity does exist between L1 and L2 data misses, and is significant enough to support the prioritization in our proposed policy. Considering the way to calculate PE, the predicted values are mostly smaller than real sampled values. In our opinion, this trend is helpful in terms of the fairness among multiple threads. This point also explains more improvement in average thread performance over other studied policies. On the other hand, given the StDev among benchmarks, different benchmarks may have various confidence levels in their respective OLS regression models. The OLS regression is of higher confidence at prediction for some benchmarks, but lower confidence for some others. Regarding the reason for the observed PE and StDev, we assume they are defined by the memory accessing patterns in different benchmarks. Meanwhile, the concurrent threads also affect the model of each other, because they totally share the memory resources in the the SMT processor. For example, *gcc* has different PE when executed with *bzip2* and *twolf* in two-threaded workloads.

4.6 Sensitivity Analysis

We first of all compare the RAPT with the RAPTn to see to how much valuable information we lost in simplifying the implementation. In addition, the sensitivity analysis is conducted regarding the impacts on RAPT's performance from different algorithmic parameters and cache system configurations.

4.6.1 The Comparison between the RAPT and the RAPTn

Generally speaking, RAPT shares the same concept with the RAPTn, but its hardware implementation is designed considering the tradeoff between hardware overhead and performance. On average, the RAPTn achieves 0.6% more overall system throughput, *i.e.*, $avgIPC$, while 2.2% more average thread improvement, *i.e.*, $abwIPC$, than RAPT. The detailed performance normalized over the ICOUNT [17] is shown in Figure 4.7. Considering RAPT is able to reduce the architecture complexity greatly, improve the computation latency to 26% and save power consumption by updating when necessary, RAPT of course has better efficiency than the RAPTn.

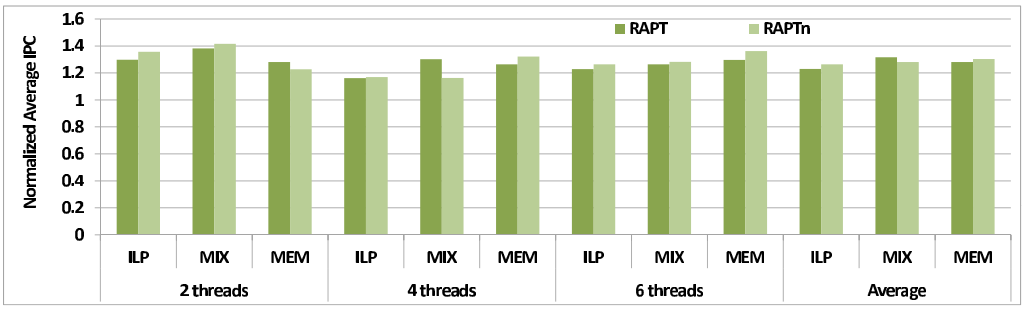
The accuracy of the RAPT is reduced mainly in the division in calculating $\hat{\beta}$ and $\hat{\alpha}$, which is substituted with rounding and shifting. It may result in less accurate predicted L2 data misses, and thus the prioritization is not based on the perfect observation. However, because of the carefully designed approximation process, essential details are kept in the RAPT, such that the performance reduction is very mild compared with the RAPTn, and it is still superior with respect to other studied policies.

On the other hand, it is interesting to see that RAPT does not incur monotonically performance reduction. Actually it even achieves better results in some workloads, *e.g.*, two-threaded MI, four-threaded BD, six-threaded CI and six-threaded

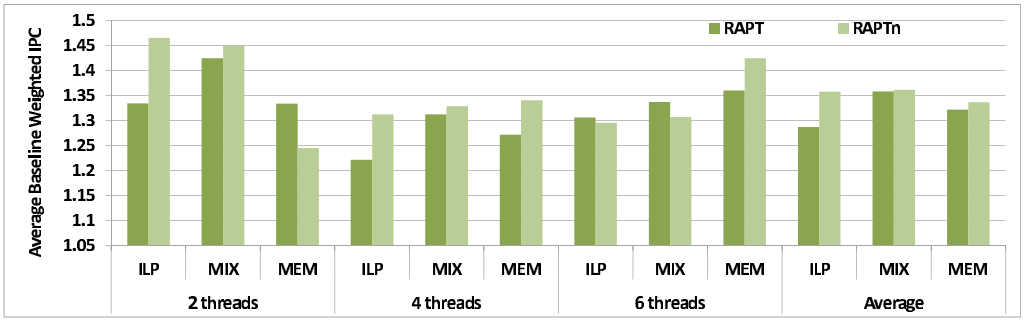
MI, in terms of either $avgIPC$ or $abwIPC$. In this case it proves the assumption that the relationship between L1 and L2 data misses is quite complicated for a linear regression. The goal of our proposed policy is to design an indicator of this relationship based on the limited information at hardware level, rather than to describe it in its completeness, which would require a far more rigorous model with no practical value in terms of hardware implementability. Our indicative model strives to improve system throughput and efficiency, but might not be able to achieve the same level of accuracy in every case. Given our approximation process, the significance of linearity in threads and the variant execution phases [5], it does not surprise us that sometimes RAPT even outperforms the RAPTn. However, the RAPTn is hardly able to produce better improvement in both overall system throughput and average thread improvement, *i.e.*, $avgIPC$ and $abwIPC$ respectively. In summary, the overall performance achievement is already defined by the OLS regression and the linearity between L1 and L2 MPKI, and our optimization in the RAPT has better performance improvement to hardware overhead ratio.

4.6.2 The Algorithmic Configurations

We go through various combinations of the SP and WS using the four-threaded BD workload composed of *gcc*, *bzip2*, *swim* and *art*. This workload introduces mediocre pressure on the system resources, compared to other two-threaded and six-threaded workloads. And its category of BD leads to a comprehensive test on both computation resources and memory accessing resources. Meanwhile, it includes both integer and FP benchmarks to utilize various resources in the processors. Finally, its performance improvement is moderate among all workloads, suggesting it is a good indicator to the average situation. We conduct the simulation over 16 configurations of the SP and WS to observe the system performance. The SP is set as 256, 512,



(a)



(b)

Figure 4.7: The Different Performances between the RAPT and the RAPTn (a) $avgIPC$, (b) $abwIPC$

1024 or 2048 instructions, while the WS as 16, 32, 64, 128 samples. The $avgIPC$ is normalized to the baseline configuration {SP, WS} = {256, 16}, and so for the $abwIPC$, which are shown in Figure 4.8.

Regarding the SP, it defines how the history information is sampled along execution. The SP = 1024 instructions may be a critical threshold in the RAPT. When the SP $\geq$ 1024, the performance is improved a little bit from the smaller-SP cases. When the SP is small, there is limited information covered and linearity might not be significant enough for a confident model. On the contrary, when samples are collected over longer period (*e.g.*, more than 1K), variability is minimized, which leads to better model. Moreover, increasing the size of the SP does not introduce overhead to our scheme, but rather makes $\hat{\beta}$ and $\hat{\alpha}$ change less frequently. Overall, we feel it is necessary to set the SP as at least 1024 instructions. Given a certain

SP, we would observe the cache behavior through a larger window, *i.e.*, a larger WS, but the WS does not play a role so important as the SP. The difference among various WS configurations is around 1% on average, when they have the same SP. Although there is rarely any monotonic relationship between performance and the WS, the WS = 32 appears to be the optimal choice across all configurations, given it generates performance stably better than the baseline configuration in Figure 4.8, and the larger the WS is, the more overhead there is.

Overall, the product of the SP and the WS means how much history information that RAPT takes into consideration in the OLS regression. The performance varies in studied configurations, but the difference is not linearly increasing as the covered instructions grow. It means the useful information in samples is not necessarily accumulated and passed to prioritization in proportional to the SP and WS. For example, when we compare {256, 128}, {512, 64}, {1024, 32}, and {2048, 16}, they all utilize the information over past 32K instructions for the OLS regression, but the performance of RAPT varies. Please note that the bottleneck here might not be only the accuracy of RAPT, but the linearity existing between the L1 and L2 MPKI also plays an essential role. Even though there are phases for applications, there is hardly any universal period for the phases, meaning a fixed SP and/or WS might not be suitable for every phase in every application.

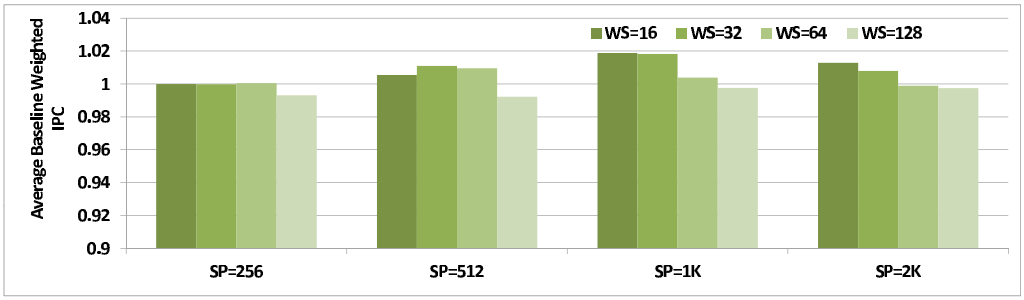
4.6.3 The Cache Configurations

In order to explore the sensitivity of our proposed scheme to different cache configurations, we studied three parameters, *i.e.*, the L1 DCache size, the L2 cache size and the L2 cache associativity, with algorithmic parameters set to {1024, 32}. They are represented by three binary bits, where 0 represents the smaller value employed in this study and 1 means the larger value as shown in Table 4.6. And the results

Table 4.6: The cache configurations

Configuration Number	L1 cache	L2 cache	L2 association
000	16KB	256KB	4-way
001	16KB	256KB	8-way
010	16KB	512KB	4-way
011	16KB	512KB	8-way
100	32KB	256KB	4-way
101	32KB	256KB	8-way
110	32KB	512KB	4-way
111	32KB	512KB	8-way

(a)

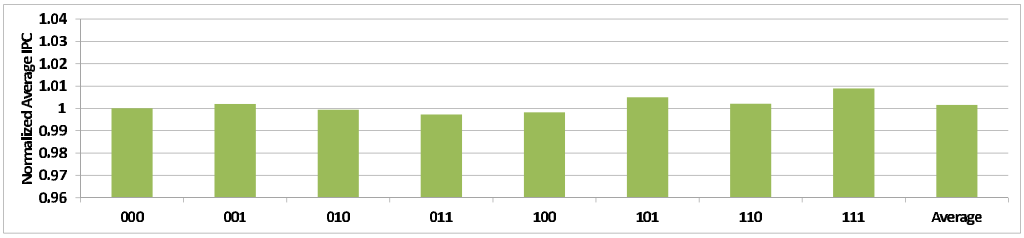


(b)

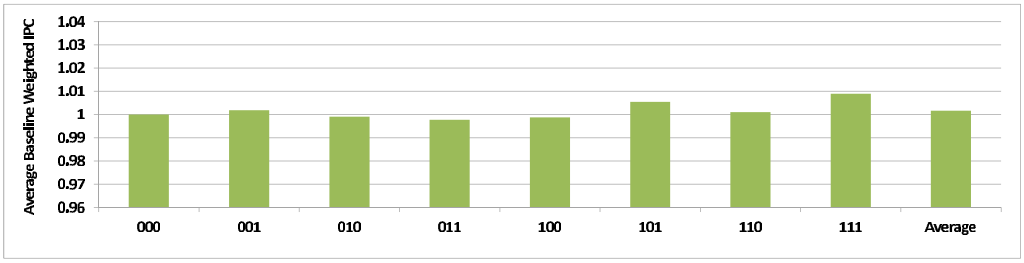
Figure 4.8: Performance of the RAPT with varying {SP, WS} (a) $avgIPC$, (b) $abwIPC$

in terms of $avgIPC$ and $abwIPC$ are shown in Figure 4.9, where the RAPT in 000 configuration are considered as the baseline for themselves respectively.

Overall, more cache resources ensure better performance. Naturally, the cache system is designed to reduce the average access delay via utilizing the temporal locality and spatial locality. By implementing more cache resources, the probability of the LLL is reduced. Consequently, the average load latency is decreased, and



(a)



(b)

Figure 4.9: Different performances in different configurations (a) $avgIPC$, (b) $abwIPC$

thus the system resources are utilized more efficiently in the configurations with more cache resources.

However, RAPT is not highly sensitive to the cache configurations. Even though the performance changes as above analysis in theory, the difference is relatively small. Comparing the case 111 with 000, nearly 1% of improvement in both $avgIPC$ and $abwIPC$ is observed for the RAPT. Other performance improvement over the baseline configuration is smaller than such data. It could result from three factors: first of all, our proposed scheme relies on the adaptively built model, which is able to cope with changes in execution. The linear model strives to handle most variation actively, so the scheme is robust with respect to the cache configuration. Secondly, the configuration does not change greatly in such simulation. Although they double in different cases, the capacity is still around similar level, *e.g.*, a couple of dozen KB or half MB. Thirdly, the approximation of division and elimination of FP is more effective than cache configurations in defining the prediction accuracy.

4.7 Summary of the RAPT

As the system resources, *i.e.*, computation resources and memory resources are totally shared by the concurrently running threads in the same core in the SMT architecture, both parallelism and efficiency are improved significantly. Nevertheless, resource contention imposes great impacts on system performance and ought to be managed by hardware scheduling policies well. Most instruction fetch policies consider long-latency load as a major obstacle towards better performance, and thus they spend great efforts on alleviating its negative impact on the system. However, they either are too late to effectively prevent the influence of cache miss, or fail to precisely describe the relationship between L1 and L2 cache misses, especially as it changes along the execution.

To explore the critical and variant relationship between L1 and L2 data misses for better resource management in the SMT processors, we proposed RAPT as a three-module decision-making scheme:

- **Sampling**: L1 and L2 data misses, *i.e.*, the MPKI, are collected for regression.
- **Regression**: The linear model is constructed by the OLS regression, such that the future L2 MPKI are predicted according to current L1 data misses.
- **Prioritization**: Higher priority is assigned to the thread(s) with a smaller predicted L2 MPKI.

Then, the fetch engine fetches according to the priorities from RAPT. Considering the RAPTn based on the native regression algorithm, *i.e.*, the OLMn, introduces great hardware overhead, we focused our study on optimized implementation RAPT, and the optimization includes simple regression, cumulative computation, elimination of FP, approximation of division and updating when necessary. Eventu-

ally RAPT introduces overhead, *i.e.*, computation latency, storage space, area and power, no more than similar hardware-based machine learning schemes.

Because RAPT minimizes the negative effect of LLLs, not only the overall system throughput but also the average thread improvement were optimized. Especially as a result of improved resource efficiency, RAPT is able to generate more system throughput in terms of increased $avgIPC$. It was better than all other policies in our study: it outperformed the ICOUNT by 28%, exceeded the STALL by 14%, the DG by 14%, and the DWarn by 8%. Because the thread with the LLL can barely move forward while other threads that are able to better utilize the shared resources can progress, RAPT improved the average thread performance in terms of $abwIPC$ over the ICOUNT by 32%, over the STALL by 14%, over the DG by 17% and over the DWarn by 11%. About 1.5% better performance could be achieved by the RAPTn based on the OLMn, but its overhead is obviously more than RAPT we were focused on. The sensitivity analysis confirms the configuration {1024, 32} of RAPT as a balanced one among all 16 configurations considering the performance difference, hardware overhead and confidence factors. Nevertheless, the RAPT is not highly sensitive to neither algorithmic configuration nor cache parameters, because the optimization shadows their impacts partially. Overall, RAPT has an adaptive nature as phase behaviors vary widely in applications.

CHAPTER 5

THE HARDWARE-ASSISTED SCHEDULING POLICY

Along with the argumentation in parallelism on the chip, comes an increase in the complexity and thus difficulty of hardware resource management. On one hand, since several physical cores are implemented on the chip, the resources are isolated between different cores. It makes a thread on one core not able to access the resources in other cores. This issue is especially undesired when the resource is idle in one core, but highly demanded in another core. On the other hand, threads in the same core share the local resources jointly when employing the SMT architectures [17]. As a result, the thread behavior is not independent any more, but rather has mutual impact on each other, which we refer to as "inter-thread interference". Thread performance will be degraded due to severe competition for the same resource, as well as inappropriate resource allocation that despises thread's demand [47]. Even though we have proposed the hardware scheduling policy RAPT in previous chapter, the ability to coordinate threads throughout multiple cores to reduce competition remains blank so far. Therefore, it requires well defined software scheduling policy in such a complicated architecture in order to optimally utilize the hardware resources [2].

As discussed in Chapter 2, OS scheduling policies try to pair threads according to their demands, and the dominant factor impairing the performance of MMMP is the resources lower than the LLC. Most of the previous studies, however, either do not consider phase changes at all, or even when they do, they commonly fall into passive and static approaches. Such approaches may be challenged by the execution phases [5], where workload demands are strongly correlated with execution phases, rather than constant or CPU cycles. Such characteristic implies that the thread scheduling based on resource demands should be synchronized with thread execution phases.

As a result, the Hardware Assisted Scheduling Policy (HASP) is proposed in this chapter, which is supported by the computer architecture, *i.e.*, the OLM, to monitor phase changes in workloads, and then to schedule threads for a collaborative pattern.

The rest of this chapter is organized as follows: The proposed design is improved from a static dispatching policy in Chapter 5.1 to a dynamic scheduling policy in Chapter 5.2. Then it is equipped with the ability to detect phase changes through the OLM in Chapter 5.3. Scalability is addressed in Chapter 5.4. We will examine the performance improvement and discuss the evaluated overhead within the relative chapters. Eventually summary of the HASP is finished in Chapter 5.5.

5.1 Static Mix-Scheduling

We start optimizing thread mapping pattern from introducing a static Mix-Scheduling policy (sMIX), which defines the scheduling at dispatching stage, and conducts no manipulation during execution [47]. It is assumed in the MMMP architecture there are multiple LLC domains and several threads share the same LLC. The basic concept of the sMIX is to distribute LLC misses evenly across different LLC domains, but it relies on off-line analysis to divide threads to different groups. LLC miss ratios are obtained ahead of execution, such that benchmarks may be Memory-Intensive (MI) or Computation-Intensive (CI) as shown in Table 4.2. According to their categories, benchmarks are sent to a two-core four-threaded MMMP as in Table 5.1. However, instead of the proposed sMIX, threads with similar demands, *i.e.*, the same category, may be scheduled to the same core, which forms the Mono-Scheduling (Mono) policy as shown in Table 5.2.

Table 5.1: Threads scheduling in the Mix-Scheduling

Core	C0	C1
Thread	MI, CI	MI, CI
WL1	*equake, gzip*	*mcf, gcc*
WL2	*equake, gzip*	*twolf, gcc*
WL3	*parser, gzip*	*twolf, gcc*
WL4	*mcf, gcc*	*parser, gzip*
WL5	*mcf, gcc*	*twolf, gzip*
WL6	*equake, bzip2*	*twolf, gcc*
WL7	*equake, bzip2*	*parser, gcc*
WL8	*equake, gzip*	*mcf, bzip2*
WL9	*equake, gcc*	*mcf, bzip2*
WL10	*parser, gzip*	*twolf, bzip2*
WL11	*mcf, bzip2*	*twolf, gcc*
WL12	*mcf, bzip2*	*parser, gcc*
WL13	*mcf, bzip2*	*parser, gzip*
WL14	*mcf, bzip2*	*twolf, gzip*
WL15	*equake, gzip*	*mcf, crafty*
WL16	*equake, gzip*	*twolf, crafty*
WL17	*mcf, crafty*	*twolf, gcc*
WL18	*mcf, crafty*	*parser, gzip*
WL19	*mcf, crafty*	*twolf, gzip*
WL20	*parser, gzip*	*twolf, crafty*
WL21	*equake, gcc*	*mcf, crafty*
WL22	*mcf, crafty*	*twolf, bzip2*
WL23	*parser, bzip2*	*twolf, crafty*

5.1.1 Experimental Methodology

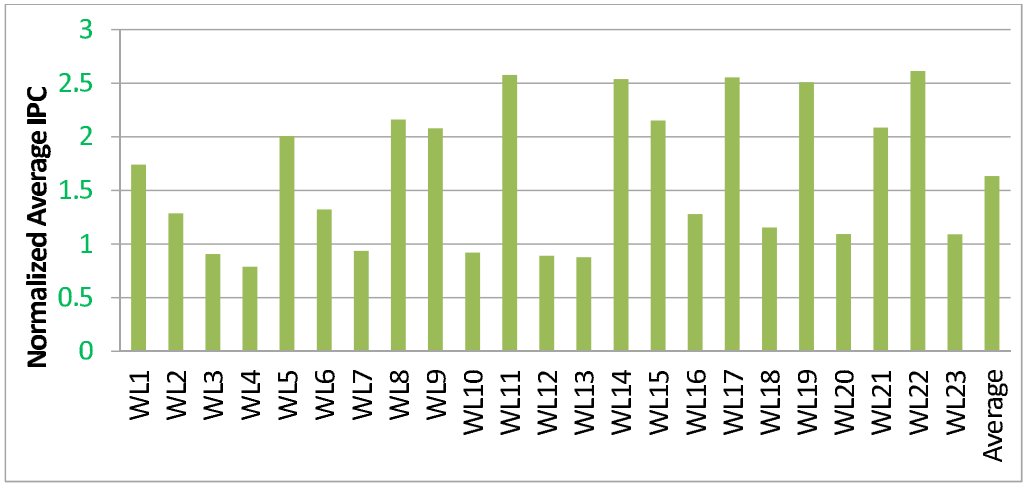
The proposed scheduling policies are implemented in the SESC with the similar configuration in Chapter 4.3. The MMMP now is equipped with two identical cores on the chip with the two-way SMT for each core. Workloads in Tables 5.1 and 5.2 are simulated for our analysis. The first two benchmarks will be sent to core 0 while the other two to core 1, and every thread will be simulated for 100 million instructions in the early simulation points [107], The ICOUNT [17] policy is used as the instruction fetch policy in the SMT environment.

Table 5.2: Thread scheduling in the Mono-Scheduling

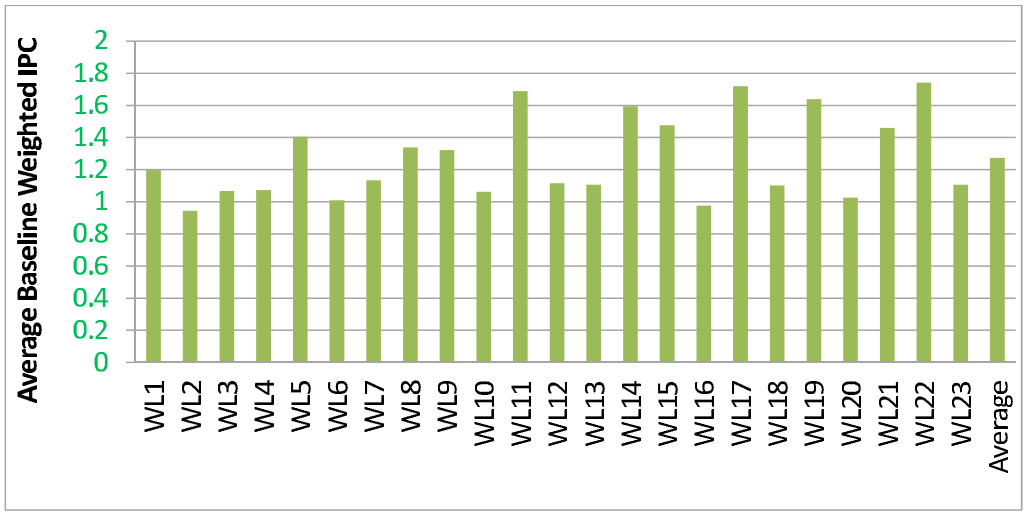
Core	C0	C1
Thread	MI, MI	CI, CI
WL1	*equake, mcf*	*gzip, gcc*
WL2	*equake, twolf*	*gzip, gcc*
WL3	*parser, twolf*	*gzip, gcc*
WL4	*mcf, parser*	*gzip, gcc*
WL5	*mcf, twolf*	*gzip, gcc*
WL6	*equake, twolf*	*gcc, bzip2*
WL7	*equake, parser*	*gcc, bzip2*
WL8	*equake, mcf*	*gzip, bzip2*
WL9	*equake, mcf*	*gcc, bzip2*
WL10	*parser, twolf*	*gzip, bzip2*
WL11	*mcf, twolf*	*gcc, bzip2*
WL12	*mcf, parser*	*gcc, bzip2*
WL13	*mcf, parser*	*gzip, bzip2*
WL14	*mcf, twolf*	*gzip, bzip2*
WL15	*equake, mcf*	*gzip, crafty*
WL16	*equake, twolf*	*gzip, crafty*
WL17	*mcf, twolf*	*gcc, crafty*
WL18	*mcf, parser*	*gzip, crafty*
WL19	*mcf, twolf*	*gzip, crafty*
WL20	*parser, twolf*	*gzip, crafty*
WL21	*equake, mcf*	*gcc, crafty*
WL22	*mcf, twolf*	*crafty, bzip2*
WL23	*parser, twolf*	*crafty, bzip2*

Because the Mono tries to mimic an arbitrary scheduling pattern that is supposed to lead to the worst performance, it shows the necessity to schedule threads based on their demands to compare the sMIX with the Mono. The performance is shown in Figure 5.1, where the baseline is the Mono. As a result, $avgIPC$ increases in the sMIX by 63% from the Mono, indicating the improved overall system throughput; and the sMIX further shows 27% growth in terms of $abwIPC$ over the baseline, expressing the better average thread improvement.

The dominant factor here causing such improvement is the diversity in different cores, *i.e.*, less inter-thread interference. By paring threads according their cate-



(a)



(b)

Figure 5.1: Improvement in the sMIX over the Mono (a) $avgIPC$, (b) $abwIPC$

gorization, threads in the same core mostly request different shared resources in the sMIX. From the perspective of the MMMP, more computation resources and memory accessing resources are utilized by different threads in the sMIX than in the Mono. In other words, idle resources are reduced by the sMIX, such that the overall system throughput is increased. On the other hand, threads experience less contention for the shared resources in the sMIX than in the Mono. Due to the diversity among domestic threads, it is less likely for them to compete for the same resource,

e.g., the MMU, the FSB and prefecth requests. Consequently, inter-thread interference is minimized by the static dispatching scheduling policy. On the contrary, the Mono scheduling policy increases the severity of contention among domestic resources by sending threads requesting similar resources to the same core. It explains the average thread improvement by the sMIX.

The sMIX is a static scheme because once the threads are dispatched, the scheduling is fixed till completion, so there are two issues unsolved for the sMIX: accessibility of off-line analysis and phase behaviors. The sMIX is highly dependent on beforehand analysis, so it cannot be applied to unknown workloads. Meanwhile, the sMIX is not addressing phase behaviors during execution. The categorization is only an indicator on average for a thread, while the thread may change its demands greatly during execution across different phases. It results in the probability that actual scheduling violates the objective for some time. To address such two issues, we feel it is necessary to implement a dynamic scheduling policy above the sMIX.

5.2 Dynamic Mix-Scheduling

Based on the sMIX, we convert it to a dynamic scheduling policy, referred as the dMIX. Since the Mono is discussed conceptually and defeated by the sMIX in the simulation, the followed context will be focused on the sMIX and later policies merely.

5.2.1 The sMIX and the dMIX

Clearly the sMIX does not adapt to the phase behavior of the threads and requires *a priori* knowledge about the threads. In an effort to improve upon it, we propose the followed changes:

1. Modify the objective to minimize the deviation of LLC misses among different LLC domains, instead of pairing threads according to pre-defined category.
2. Employ misses over certain amount of instructions instead of miss ratio.
3. Use epochs to conduct online evaluation instead of off-line.
4. Perform runtime migration of threads instead of static mapping pattern.

As a result, the sMIX is converted to the dMIX, a dynamic scheduling policy driven by CPU epochs. At the end of every epoch, the OS scheduler processes the sampled misses for every thread, and thus every LLC would have a summation of MPMI or MPKI of all domestic threads. The dMIX conducts the search to find out the optimal scheduling, in the sense of having the smallest difference in terms of total misses across all LLC domains. In other word, the difference, or Standard Deviation ($StDev$), of cache misses across LLC domains should be minimized.

For example, assuming there are four threads on two LLC domains, with each LLC supports two threads. At the end of an epoch, MPMI of all threads are obtained and evaluated by the OS scheduler. Here arbitrary numbers of MPMI are used for illustration purpose only, which are shown in Table 5.3. As we can see, there are three possible scheduling patterns. However, Pattern 2 has the least $StDev$ and thus it will be the new scheduling decision and the dMIX will perform thread migration if necessary. If migration is necessary to achieve Pattern 2, threads with less MPKI or MPMI will be chosen to move. It saves the efforts to warm new caches, and thus it introduces less disruption to normal thread execution.

Table 5.3: The objective in the dMIX

	Core 0	Core 1	$StDev$
Pattern 0	1, 2	3, 4	2
Pattern 1	1, 3	2, 4	1
Pattern 2	1, 4	2, 3	0

5.2.2 Throughput of the dMIX

Here multi-programming workloads in Table 5.4 are following the concept in the sMIX, *i.e.*, pairing per category. The workloads in Table 5.4 is the dispatching pattern of the sMIX, and they are simulated here in accordance with the description in Chapter 5.1.1. In order to obtain experimental results to advance the multicore scheduling policy, the workloads WL24, WL25 and WL26 are employed to study the dMIX.

Table 5.4: The workloads in the dMIX

Core	C0	C1
Thread	MI, CI	MI, CI
WL24	*twolf, apsi*	*art, bzip2*
WL25	*parser, crafty*	*equake, gcc*
WL26	*swim, gzip*	*mcf, wupwise*
WL27	*twolf, bzip2*	*apsi, art*
WL28	*parser, gcc*	*equake, crafty*
WL29	*swim, wupwise*	*mcf, gzip*

On the contrary, the workloads WL24, WL25 and WL26 will be the initial allocation in the dMIX. The difference is that the dMIX will reevaluate the thread scheduling at every epoch and make dynamic migration if necessary. Here we vary the epoch ranges among 10, 50, 100 and 200 million CPU cycles, which are adequate to observe major phase changes at millions of instructions [5]. Performance improvement by the dMIX in terms of $avgIPC$ and $abwIPC$ is shown in Figure 5.2, where the results are normalized to those of the sMIX. On average, the dMIX achieves 19% better performance than the sMIX. Given that the sMIX even requires knowledge about the threads beforehand and the dMIX does not, the dMIX justifies a promising direction for thread scheduling. Moreover, by comparing the dMIX-K (the dMIX based on MPKI) with the dMIX-M (the dMIX based on MPMI), it is found that the dMIX-M is able to improve system performance more than the

dMIX-K. It means MPMI is better at expressing thread behaviors in the long run. Therefore, our schemes will be focused on MPMI from now on.

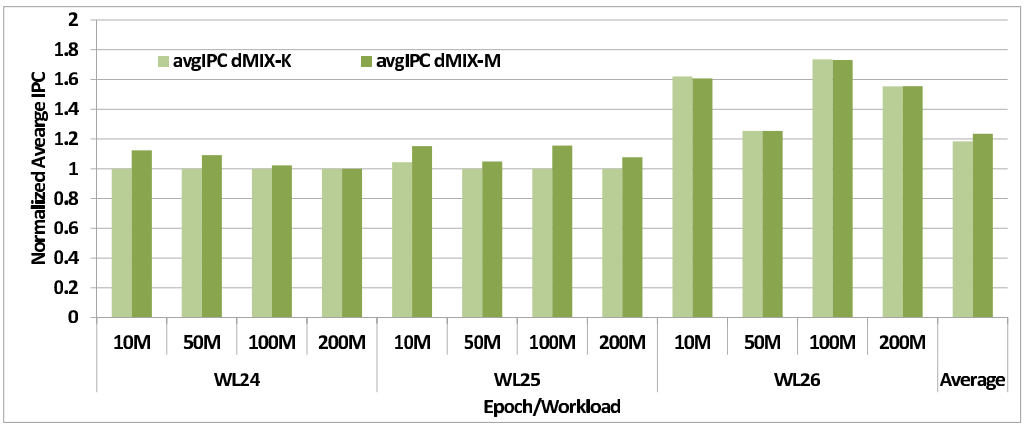
Although dynamic scheduling policy is able to generate better throughput than the sMIX, we feel the size of the epoch is still a predefined fixed value that will not fit the different needs from various workloads. For example, for the WL24 the best performance is achieved when the epoch is 10 million CPU cycles, while the WL25 has the best performance when the epoch is 100 million CPU cycles. It results from the nature of phase behaviors that phase duration varies greatly across different benchmarks. Therefore, a better scheduling option would be based on phases, rather than fixed CPU cycles, such that the thread management is synchronized to execution phases.

5.3 The Phase Triggered Scheduling Policy

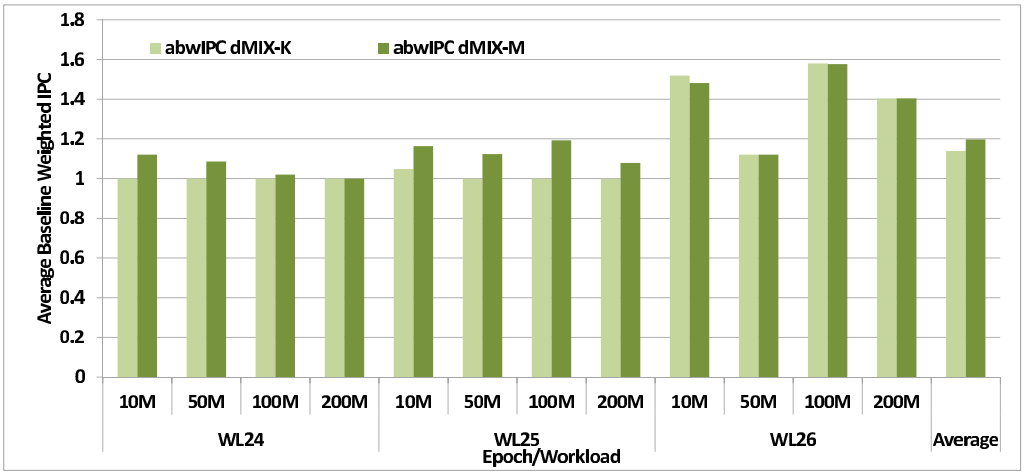
In an effort to better conduct the dynamic scheduling policy, we propose the Hardware Assisted Scheduling Policy (HASP) here to employ hardware components to observe phase behaviors, and then evaluate the scheduling in the OS upon recognized phase changes. This is a novel approach in the sense of having both hardware and software parts in the design, and in total there are five modules:

1. **Sample**: Sample L1 and L2 data misses for every thread
2. **Model**: Construct thread models by the OLS regression
3. **Phase**: Monitor the models to identify phase changes
4. **Pattern**: Evaluate the scheduling after the changes
5. **Thread**: Migrate threads if necessary

Three modules **Sample**, **Model** and **Phase** are implemented by the OLM to assist the HASP in observing phase changes, while the rest are finished in the OS scheduler



(a)



(b)

Figure 5.2: Performance of the dMIX (a) $avgIPC$, (b) $abwIPC$

at the software level. The motivation to embed three of them in architecture is to save the efforts to interrupt the OS and normal execution and transmit data between the OS and the architecture. The **Pattern** module and the **Thread** module remain in the OS, while the latter one is considered default in most OS kernels.

5.3.1 The Sample Module

The **Sample** module is inherited from the OLM and supported by its corresponding hardware engine: the **Sampling** engine. The inheritance refers to not only the

conceptual design, but also the optimization in the OLM and RAPT. Hence the duplicate explanation is omitted here, but only modification is listed here. Furthermore, due to the comparison between the RAPT and the RAPTn, we are now confident about our optimization in the OLM, so there is no need to carry the HASPn with us any more.

There is hardly any essential change in the modules or engines, but we employ {1M, 32} as the regression parameter. It leads to the same WS, but the SP is enlarged to 1 million (1024 × 1024) instructions. Given a metric as MPMI, 20 bits are enough for the counters and registers storing the samples, while 16-bit designs are employed in our optimized design to reduce the overhead, considering most MPKI are under 150 ($2^8 = 256$) [74]. The counters and registers are listed in Table 5.5, where TN stands for Thread Number. The rightmost column lists registers for $\hat{\beta}$ and $\hat{\alpha}$, which are not available until the end of the **Model** module.

Table 5.5: The storage for the HASP

	Counter	Register	Register
HASP	32×TN	32×WS×TN	32×2×TN
Purpose	MPMI	Samples	$\hat{\beta}$ and $\hat{\alpha}$

5.3.2 The Model Module

The **Model** module is inherited from the OLM and supported by its corresponding hardware engine: the **Regression** engine. Strictly speaking, this module takes over the optimized design of the **Regression** engine, *i.e.*, the OLM. The **Model** module provides the design with two fundamental indicators: the model of thread behavior, *i.e.*, $\hat{\beta}$ and $\hat{\alpha}$, and evaluated LLLs, *i.e.*, $\hat{y}$. The former one comes from the regression algorithm and plays an important role in triggering pattern evaluation, while the

latter one may be used to define the scheduling. Mathematically it is calculated in Equation 3.6 in Chapter 4.

5.3.3 The Phase Module

This module monitors new models from previous module and identifies changes, and is supported by the **Prioritization** engine, which is designed naturally to compare values. In particular, the HASP recognizes phase changes by comparing new model parameters with the old. We calculate "first order derivative" for both estimators:

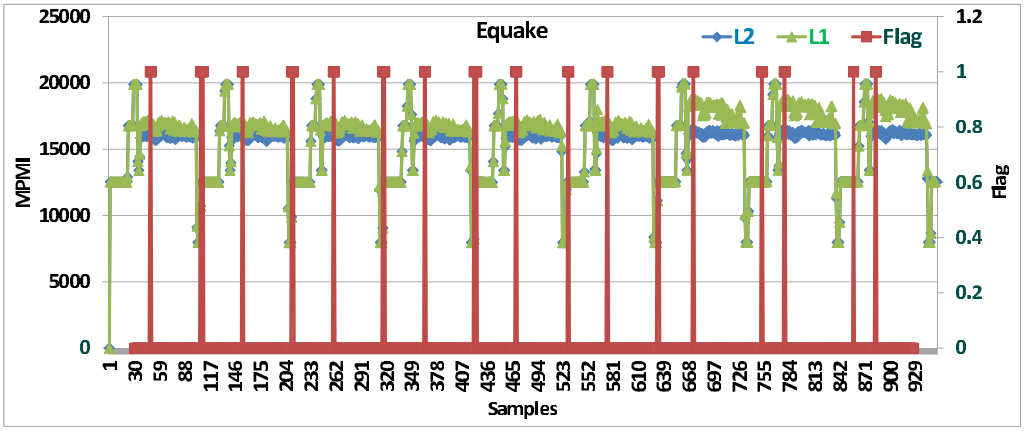
$$\Delta\hat{\beta} = |\frac{\hat{\beta}_{new} - \hat{\beta}_{old}}{\hat{\beta}_{old}}| \tag{5.1}$$

$$\Delta\hat{\alpha} = |\frac{\hat{\alpha}_{new} - \hat{\alpha}_{old}}{\hat{\alpha}_{old}}| \tag{5.2}$$

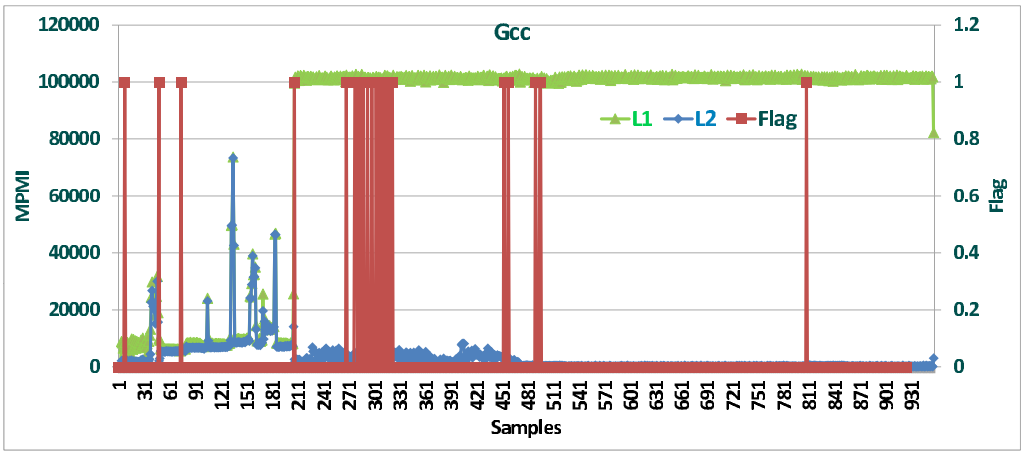
A phase change is identified when $\Delta\hat{\beta}$ and $\Delta\hat{\alpha}$ both exceed the threshold (δ).

$$newPhaseFlag = \Delta\hat{\beta} > \delta \; AND \; \Delta\hat{\alpha} > \delta \tag{5.3}$$

To illustrate how the **Phase** model works, we collected MPMI of two benchmarks: *equake* and *gcc* from their 1-billion representative regions [108]. Every 32 consecutive samples of L1 and L2 MPMI , *i.e.*, {1M, 32}, are processed to form a linear model, *i.e.*, $\hat{\beta}$ and $\hat{\alpha}$. The $newPhaseFlag$ is raised when Equation 5.3 is true, where δ is set to 1. The results are shown in Figure 5.3. We can see that our linear model closely captures the phase behavior of *equake*, where flags are able to indicate major phase changes. Nevertheless, the flags are not raised sharply at the beginning of a new phase but lag a little bit. This is because it takes more than one sample to accumulate the changes in $\hat{\beta}$ and $\hat{\alpha}$ in order to raise the flag. On the other hand, *gcc* is more challenging that its flags are raised irregularly. Around the 300th samples, flags are raised so often that it loses the purpose of phase detection. The options to cope with such irregularities will be discussed in the next section.



(a)



(b)

Figure 5.3: The model indicates Phase changes (a) *183.equake*, (b) *176.gcc*

5.3.4 The Pattern Module

The **Pattern** module is the one who evaluates thread scheduling as much as possible and choose to take action according to the objective. The objective, however, is kept the same as to evenly distribute LLC misses throughout LLC domains, as shown in Table 5.3. Nevertheless, there are now two factors that ought be discussed in this module, which are the evaluating timing and the migration skipping. Let's examine them separately.

Evaluating Timing

As a straightforward solution from flags raised in the **Model** module, every flag may trigger a round of evaluation, which is marked as "Immediate" approach. This approach is a good choice when flags perfectly indicate phase changes in execution, *e.g.*, *equake*; but it may not work for some other cases, *e.g.*, *gcc*. To compensate for this disadvantage, we propose a hybrid approach that employs the concept of both epoch and flag, which is marked as "Advanced". In this section, we are focused on the Advanced approach that is explained in the followed context.

Given we set an epoch as 200 million CPU cycles, it means the system will evaluate the thread scheduling at the end of every 200-million-cycle epoch, under the condition there is no flag raised in this epoch. If there is a flag raised from one thread, no immediate evaluation will be triggered; instead, next evaluation is advanced, *i.e.*, the rest of the current epoch is cut to 50%. For example, if the current epoch starts at t and at $t + 100$ million cycles there is a flag raised from thread 0, then the next evaluation timing will be advanced to $t + 150$ million cycles, instead of the original $t + 200$ million cycles. If there is another flag raised from thread 1 at $t + 110$ million cycles, the time to next evaluation will be reduced by another 50% to $t + 130$ million cycles, so on and so forth. Please note in each epoch, only the first flag from each thread is allowed to advance the evaluation time, other subsequent flags from the same thread will be ignored. The rationale behind is if flags from multiple threads are raised during one epoch, it indicates there is a growing demand to evaluate the scheduling. Consequently, the next evaluation is advanced by those flags. If all flags are from one thread, it might be its transient behavior. So more changes from more threads will lead to a much earlier evaluation in this design.

The pseudo-code to express such an idea to advance next evaluation by flags is shown here:

while evaTiming ≠ clockTicks **do**
  if newPhaseFlag[i] && firstChange[i] **then**
    evaTiming ← (evaTiming - clockTicks >> 1) + clockTicks
    firstChange[i] ← FALSE
  end if
end while
Evaluate(Scheduling)
evaTiming ← Epoch + clockTicks

where evaTiming is the next evaluating timing, clockTicks is the CPU clock, i is the thread identity, firstChange is to indicate the first change of a unique thread and Epoch is 200 million CPU cycles in the Advanced approach.

Migration Skipping

Another approach to minimize thread migration overhead is to deny unnecessary migrations, where the necessity is estimated as the changes in terms of $StDev$. It means if the OS scheduler sees too little change in LLC miss re-distribution, the corresponding migration is denied, even though it matches the objective of minimizing the $StDev$ across all LLC domains. In this way, the OS scheduler can ensure that the conducted migration would lead to significant improvement, that deserves the effort and overhead. In one word, the migration is skipped because of ignorable change in re-distributing LLC misses, when Equation 5.4 is true.

$$\frac{|StDev_{new} - StDev_{old}|}{StDev_{old}} \leq \theta \tag{5.4}$$

where $StDev$ is the standard deviation of LLC misses among different LLC domains, *new* means the anticipated value after migration, while *old* is the current value.

The **Thread** module is considered default in most OS kernels, so its implementation is omitted here. As a result, the example MMMP is amended to implement the proposed OS scheduling policy in Figure 5.4

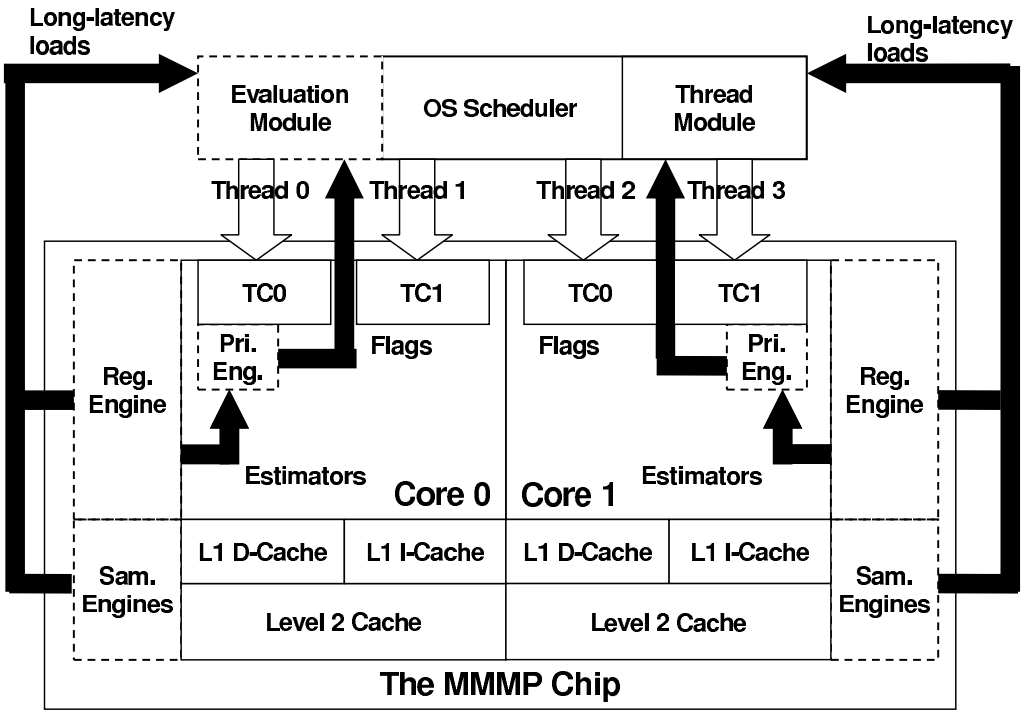


Figure 5.4: Model of the HASP in a two-core four-threaded MMMP

5.3.5 Performance Discussions

In total there are five overheads caused by five modules:

- The Sample reading overhead: SO
- The Model construction overhead: MO
- The Phase detection overhead: PO
- The Scheduling evaluation overhead: EO

- The Thread migration overhead: TO

The EO and the TO are applied to all dynamic scheduling policies, *i.e.*, dMIX and HASP, and thus the sMIX is excluded. The MO and the PO are exclusively for HASP, but they are finished by dedicated hardware components. Therefore, they have no interference with normal execution and do not increase the execution time of the threads. The SO is also for dynamic policies, but with different meanings: the dMIX reads all historic samples to get an overview in the past, while the HASP only reads a single average or predicted value.

Table 5.6: The adopted overheads in the HASP

Overhead	dMIX	HASP	Purpose
SO	6400/evaluation	100/evaluation	Sample reading
MO + PO	N/A	Hardware Supported	Model processing
EO	10K/evaluation	10K/evaluation	Pattern evaluation
TO	0:60M/pair/migration	0:60M/pair/migration	Thread migration

Upon a new evaluation, WS, *e.g.*, 32, samples are read from the architectural registers by the dMIX to conclude the past information. We assume on average a single sample costs 100 CPU cycles to interrupt the OS and transmit data, such that the total SO added to a thread in the dMIX is $100 \times 2\times$ WS per evaluation. Given the default configuration $WS = 32$, this number is around 6400 cycles per evaluation. However the HASP only reads a single value, so it spends 100 CPU cycles on the SO per evaluation. According to our design about the **Model** and **Phase** modules, we find that 100 CPU cycles are enough to finish the computation, so the MO and the PO are considered as 100 per model. And since we also need to generate the model 100 times ($100M/1M$) during the thread's lifetime, ten thousand cycles would be added to HASP if it were NOT supported by hardware.

The evaluation in a system with four threads to generate the scheduling pattern is not that time consuming, and this overhead can be partially covered by dedicating

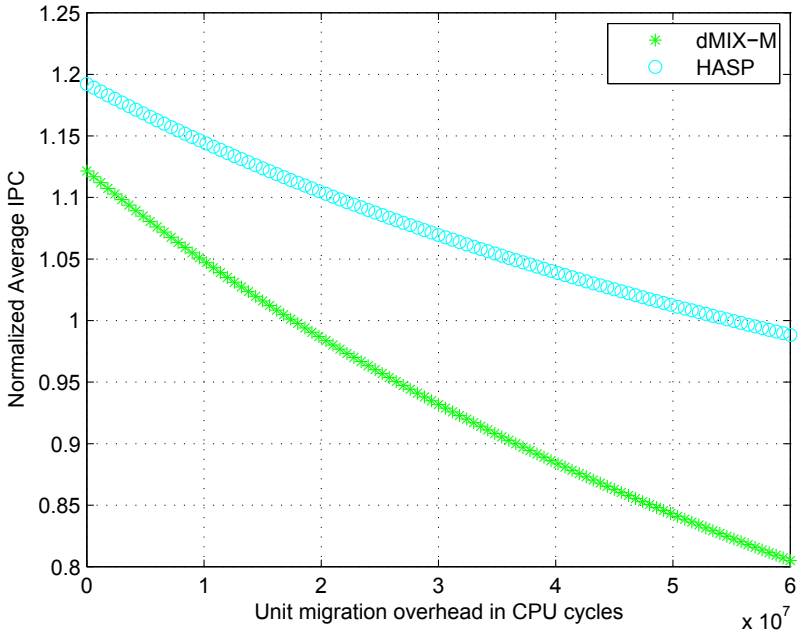
idle resources on the chip to perform the computation. Consequently 10000 cycles per evaluation for the EO is a solid estimation here. As far as the TO is concerned, it involves the time to swap threads between ready queues of different cores. Please note the effect to warm up new caches as a result of thread migration is well simulated by the simulator and has been taken into consideration already. Nevertheless, due to its inevitable impacts on the overall thread execution time by suspending and moving threads around, the TO is the major impact factor in our overhead analysis. We vary it from 0 to 60 million CPU cycles to examine its influence on the performance of our scheduling policies, and also provide a large safety margin to any unknown overhead.

All evaluation metrics are normalized to the sMIX, our baseline scheduling policy. The raw thread performance numbers are generated by the simulator, and then we manually add the overheads from Table 5.6 to each thread in order to calculate its performance. In this way, we feel such analysis overcomes the limitation of the simulator, and provides a comprehensive overview on performance considering various overheads.

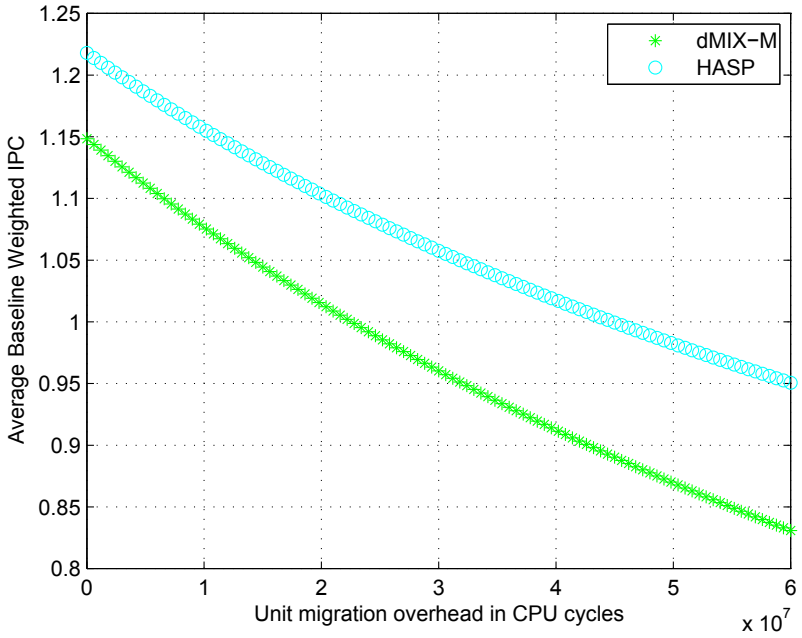
Coordinated Performance

Considering the dMIX-M is better than dMIX-K and the HASP has an SP of 1 million instructions, the six workloads in Table 5.4 are simulated to compare the studied policies: dMIX-M and HASP. Figure 5.5 presents the performance comparison of the HASP scheme defined in this section and the dMIX scheme from the previous section. θ is kept as 0.1 and δ as 1.0 for all workloads. Other simulation methodology remains the same as in Chapter 4.5.

Overall, the HASP approaches are able to achieve 19% improvement in $avgIPC$ and 22% in $abwIPC$ in raw performance when no overhead of any kinds is considered



(a)



(b)

Figure 5.5: Performance of the dynamic scheduling policies considering overheads (a) $avgIPC$, (b) $abwIPC$

at all, while dMIX scenarios improve over the baseline by 12% in $avgIPC$ and 15% in $abwIPC$. Figure 5.5 also shows the performance trend as migration overhead increases in the system, when all other overheads such as the SO, the MO, the PO and the EO are considered. On average all dynamic policies have shown better performance than the static approach sMIX when migration overhead is small. We can increase the migration overhead to as high as 60 million cycles, the HASP still outperforms the sMIX in terms of $avgIPC$ when the dMIX-M fails. The fairness of the HASP is justified by its superior $abwIPC$, as well as the average variance of IPCs: 0.0617 in the sMIX, 0.040 in the dMIX and 0.022 in the HASP. The above experimental results validate our dynamic scheduling policies in most tough environment in terms of performance and overhead.

Furthermore, the HASP has better sustainability than the dMIX policy. Its migrations are only 86.7% on average of the dMIX. It thus introduces less overhead to the final results, which is shown as the smaller slope in the Figure 5.5. Furthermore, because the HASP is based on the execution phases, the evaluations and migrations are closely associated with the committed instructions. Hence, the reported migrations show the frequency of such events with respect to the execution of the thread. The frequency ought to remain at a similar level, even if more instructions are executed. Moreover, the HASP has the ability to identify execution phases adaptively, and maintain a low level of overhead, which is much better than the dMIX policy. In summary, the HASP has superior scalability than the other studied policies, in spite of the limitation in the architectural simulator.

Sensitivity to the Threshold

In above simulations of the representative workloads, δ is available from 1.0, 0.5 to 0.1, such that the average performance is obtained across different workloads with

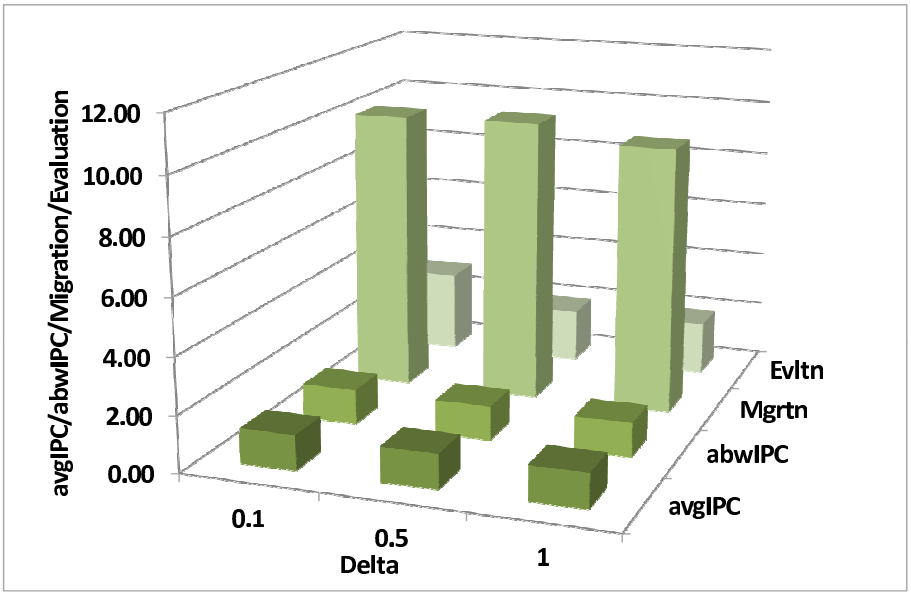


Figure 5.6: Delta defines overhead

respect to different δ. δ may lead phase detection to opposite directions: when a small number is adopted, frequent changes would trigger too many pattern evaluations, which might not be necessary since the phase is not categorically different yet. On the contrary, a large δ may fail to identify major phase changes, such that thread paring is not following threads well. Therefore in Figure 5.6, we present the times of migrations, the times of evaluations, $avgIPC$ and $abwIPC$. As expected, times of evaluation and migration are reduced by large δ. It means some changes in linear models are not recognized by the HASP, such that they do not trigger evaluation or migration. Meanwhile, the performance is not greatly harmed by less evaluation and migration. It means to evaluate and migrate only when significant changes happen in the system is necessary and adequate. In summary, we advocate $\delta \geq 1.0$, while larger number may be validated by specific workloads and overheads.

Another variable is θ that we used to reject some migrations when they do not lead to considerable change in the system. We change it among 0.0, 0.1, 0.2, 0.3, 0.4 and 0.5, where δ is kept 1 in the simulation. Overall what we see from

the simulation results is that migration is affected in several extreme cases. For example, when $\theta = 0.0$, a migration that did not happen in previous simulations now is conducted for workload WL25; while one migration is rejected when $\theta = 0.5$ for WL26. Such changes do not introduce consistent performance impacts on the system, and other simulations, *e.g.*, $\theta = 0.1, 0.2, 0.3$ or 0.4 have the similar behavior. The reason for so few change caused by θ is because it is a variable evaluated after δ. As δ is able to deny most little phase changes when it is appropriately designed, this leads to any changes that are submitted to θ to evaluate is probably significant enough, such that it is less likely for θ to deny a lot of migrations.

5.4 Scalability in the HASP

So far we have introduced two improvements in optimizing thread scheduling in the MMMP: the first one converted the scheduling policy from a static dispatching scenario to a dynamic paring policy; while the second one implemented the ability to identify phase changes in execution. The study now is focused on scalability: is our proposed policy able to scale to larger system capacity, *i.e.*, more cores and more threads?

5.4.1 Increasing Capacity

We anticipate the popularity of the MMMP as well as its increasing system capability in the foreseeable future. However, it does not necessarily mean we can arbitrarily increase both core and thread counts. Especially the multithreaded architecture itself may come to a saturation point in terms of performance when there are more than six threads [22, 105, 106]. In those extreme cases that dozens of threads are employed, there is rarely any performance benefits because of severe cache conflict-tion. Therefore we believe there will be only a moderate number of threads on one

core for the future MMMP. On the other hand, more rapid increment in core counts in the MMMP are considered feasible, *e.g.*, doubling the number of cores with each technology generation. Time complexity in searching for the optimal scheduling will increase exponentially if we want to evaluate all the threads across all cores, irrelevant to the kind of metric we employ. Ultimately, it merges to an NP-hard problem [48], and the time complexity is $O(n^u)$, where n is the total number of tasks and u is the number that an MMMP supports simultaneously [109, 110].

Luckily, Zhuravlev *et al.* [111] found it practically enough to concentrate upon merely two pairs of threads in a swapping-based policy, which are from four different LLC domains. Although it is feasible to do more, the performance improvement approaches saturation very quickly after more than two pairs. As a result, it is not necessary to perform exhaustive search in order for our proposed scheduling policy to scale up. All we need is to focus only on a limited number of threads across a limited number of cores.

5.4.2 Compromise to Scalability

The performance of the HASP was examined previously, and it showed the frequency of evaluations and migrations is sustainable along the execution in the HASP. Now the question here is about the HASP in a larger system, such as cluster or cloud computing. We will take out more designs for the HASP, so that the scalability is optimized in the most tough environment.

Based on the previous study, we will limit the migration to only two pairs of threads, so the evaluation overhead may be the only scalability issue in the HASP. This overhead will be increased with system capacity due to the scheduling policy has to access more counters and perform more computations. To adjust towards scalability, we propose to locate up to four cores on different LLCs in the pMIX:

two of which have most LLC misses, *i.e.*, heavy cores, while two others have the least LLC misses, *i.e.*, light cores. Then two pairs of threads are selected for replacement: the heaviest threads from heavy cores against the lightest threads from light cores.

The proposed policy locates all candidate threads first and then swaps them to evenly distribute LLC misses as shown in Figure 5.7. In Figure 5.7(a), four extreme cores are chosen based on their total LLC misses, which are summarized from all domestic threads. These four cores are significantly unbalanced and they generate different demands on resources at lower memory hierarchy (below the LLC). On the contrary, we assume that the unselected cores impose relatively moderate demands on the resources, such that their potential performance improvement might not deserve the thread migration. In an effort to reduce *StDev* of LLC misses across different LLC domains without introducing excessive evaluation overhead, four extreme threads are located on such four cores: two heaviest threads, *i.e.*, most LLC misses, on Core 0 and 1 in Figure 5.7(b), and similarly the two lightest threads on Core n and $n-1$. Then they are swapped among four cores to better distribute LLC misses. As a result in Figure 5.7(c), the four extreme cores now have similar LLC misses, such that *StDev* is reduced to its best efforts. In summary, assuming system has c cores and every core has t threads, complexity of evaluation in the HASP is in the order of $O(c \times t) + 4 \times O(c) + 4 \times O(t) \approx O(c^2)$ when $c >> t$.

Table 5.7: More workloads in the simulation

Workload	Benchmark List
WL30	*apsi, art, bzip2, twolf, crafty, equake, gcc, parser*
WL31	*apsi, twolf, bzip2, equake, gzip, wupwise, swim, art*
WL32	*crafty, mcf, gcc, wupwise, gcc, art, swim, twolf*
WL33	*apsi, parser, bzip2, mcf, crafty, wupwise, gcc, art*
WL34	*apsi, mcf, bzip2, wupwise, gzip, equake, swim, parser*
WL35	*crafty, twolf, gcc, equake, gzip, parser, swim, mcf*

5.4.3 More Designs in the HASP

Please recall that we have two approaches for the evaluating timing: Immediate and Advanced. The advanced approached was employed in the earlier simulations, because we assume that it is associated with more efficient operations and thus less overhead. Now the Immediate approach will be examined in this section.

Aside from the evaluating timing, the evaluated variable is another factor that might affect the performance of the HASP. The target of the OS scheduler here is to achieve minimal difference of LLC misses across different LLC domains. In our scheme, we deem LLC misses can be obtained from two sources: the past LLC miss value or the predicted future LLC miss value. The past LLC miss value is calculated based on regression algorithm, which is the average of sampled data mathematically represented as $\bar{y}$ in Equation 3.4. We know that such an average value is concluded from history information, and it covers past WS $\times$ SP instructions, *e.g.*, 32 million for the {1M, 32} configuration. On the other hand, the predicted LLC misses are calculated based on the newly constructed linear model as $\hat{y}$ in Equation 3.6, where x is newly sampled L1 data misses and $\hat{\beta}$ and $\hat{\alpha}$ are estimators. The predicted LLC misses can be employed because we assume current migration will lead to a pattern that is valid for some time in the future. Therefore, prediction may help anticipate workload behaviors.

As a result, there are four different implementations of the HASP, which are shown in Table 5.8. The listed derivatives here are all equipped with the ability to manipulate up to two pairs of threads in a larger system.

5.4.4 Scheduling in Larger Systems

More simulations are conducted using eight-threaded workloads in four-core configurations. The workloads are listed in Table 5.7. The overheads from Table 5.6

Table 5.8: Four derivatives of the HASP

Name	Evaluated variable	Evaluating timing
HASP-I	Average L2	Immediate
HASP-II	Predicted L2	Advanced
HASP-III	Average L2	Advanced
HASP-IV	Predicted L2	Immediate

are adopted again. Hence the overall performance comparison of different HASP schemes in a larger system (with 8 threads) is shown in Figure 5.8, where the baseline is still the sMIX.

Once again, all HASP schemes are able to achieve better performance than the static approach of the sMIX when migration overhead is small. Please note even when the migration overhead is zero on the leftmost in the figure, we still take the other overheads (SO, MO, PO, EO) into account for the evaluation. The average raw improvement (with no scheme overhead at all) across all four different HASP schemes is 8% in terms of $avgIPC$ and 13% in terms of $abwIPC$ over the sMIX. However, performance reduction is spotted here when compared with Figure 5.5. This is because in our adjustment towards scalability, we no longer perform exhaustive evaluation of all possible thread swapping anymore, in order to get reduced computation complexity. The price we pay is we cannot guarantee the optimal scheduling anymore, but a sub-optimal decision.

Secondly, the HASP-III is able to maintain the evaluations and the migrations at the similar level as in the smaller system. As the system becomes larger, the evaluations and migrations are kept roughly stable, so we will rarely see any dramatic increment in the scheme overhead. It validates the HASP for larger systems. Nevertheless, the major reason for the scalable overhead is because we delay the evaluation process in the Advanced approach, and limit the quantity of migrations when the system is large. Overall, the HASP-III tries to search for a balanced point

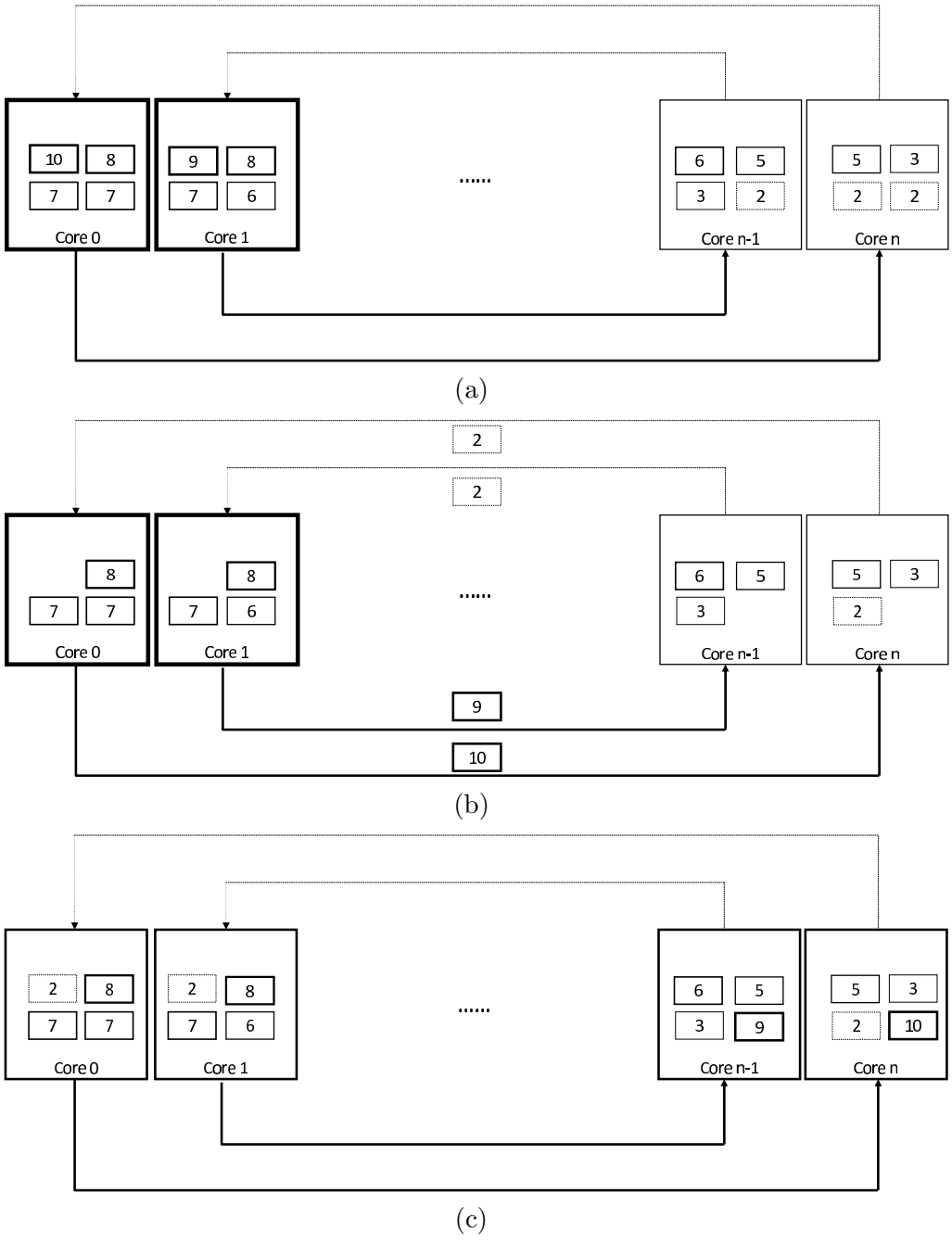
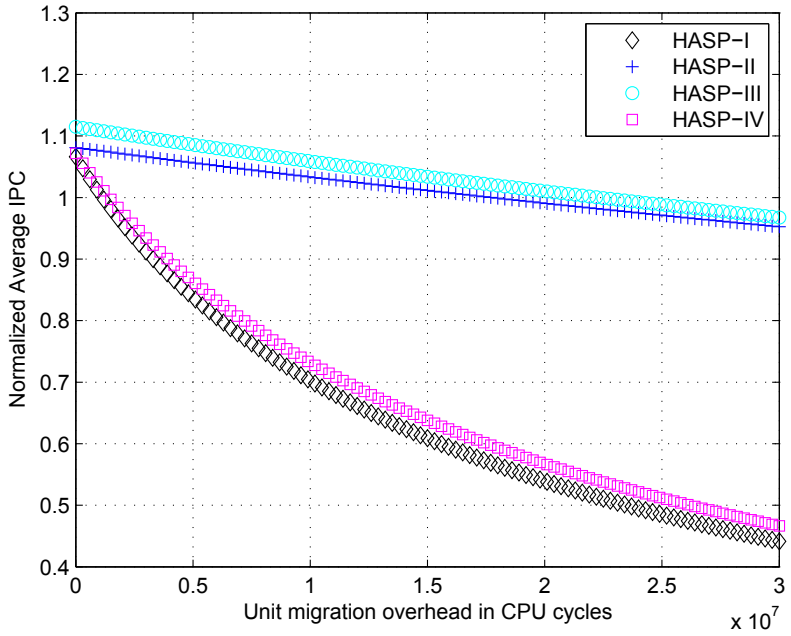
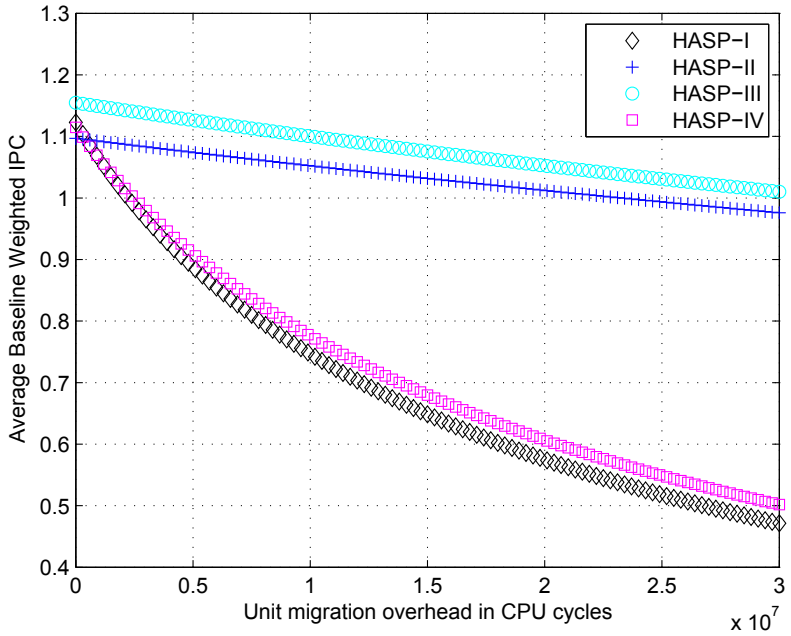


Figure 5.7: Scalable evaluation in the HASP. Numbers illustrate threads' LLC misses and line width corresponds to the heaviness: (a) Four extreme cores are chosen, (b) Four threads are selected for replacement, (c) The target environment is of evenly distributed LLLs across LLC domains



(a)



(b)

Figure 5.8: Performance of the HASP in larger systems (a) $avgIPC$, (b) $abwIPC$

between exhaustively following the phase changes and exponentially increasing the overhead. Hence, HASP-III shows a good rating for scalability with respect to longer execution and more threads.

Thirdly, the HASP-II and the HASP-III show strong resilience towards migration overhead, while the performance of HASP-I and HASP-IV drops sharply when the migration overhead increases. This is because the HASP-I and the HASP-IV evaluate upon every recognized phase change, which is associated with considerable system overhead. On average their evaluations are 25X over those of the HASP-II and the HASP-III, and hence the HASP-I and the HASP-IV suffers from more negative offset on Y-axis in Figure 5.8. Meanwhile, they also conduct 12X more migrations than the Advanced approaches, and thus show quick performance reduction as the migration overhead increases in the figure. As a result, the Advanced approaches only explore a small number of migrations in the execution, showing acceptable overheads in managing threads, but still are able to improve system performance. Among them, the HASP-III exhibits superior and sturdy performance even with increasing system capacity.

In summary, the Advanced approach and average L2 misses improve the HASP greatly in terms of the performance and the overhead. And the HASP provides satisfactory scalability in longer execution and larger systems. It stabilizes the evaluation and migration frequency such that longer execution will not increase the overhead exponentially. Meanwhile, it ensures the optimal scheduling when the system is small, and adopts a heuristic approach for the sake of scalability in larger systems.

5.5 Summary of the HASP

The emergence of MMMP introduces great challenges on how to efficiently and effectively manage resources for better throughput. On one hand, resources on different cores are used unevenly; on the other hand, multiple threads on each core introduce inter-thread interference, which impacts the performance of all. The dynamic phase behavior of each thread complicates the issue even more. In order to overcome these difficulties, MMMP requires a thread management scheme to fully utilize resources across different cores and to minimize competition among threads.

We believe an intelligent OS scheduling policy is capable of taking this responsibility at the system level, through pairing threads in accordance with the resource allocation and thread demands. In this chapter, we first proposed a static dispatching policy with no runtime manipulation. Then we demonstrated a dynamic scheduling policy driven by CPU epochs with runtime migration capability. Finally we introduced our regression-driven scheduling policy which is capable of capturing the phase changes of threads and scheduling them correspondingly. Our experimental results showed that the regression-driven policy clearly outperforms other policies due to its ability to capture the thread demands adaptively and then pair threads dynamically. The HASP exhibits superior performance and outstanding scalability in both small and large systems, and will be employed in designing the integrated approach. From now on, the HASP is used to refer the proposed software scheduling policy, which is a combination of HASP is Chapter 5.3 and HASP-III in Chapter 5.4.

CHAPTER 6

THE ADAPTIVE THREAD MANAGEMENT SCHEME

As stated previously, the MMMP faces two major challenges in resource management: complexity in resource allocation and variation in workload behaviors. The Multicore and multithreaded architectures introduce both unification and isolation to on-chip resources. In such an environment, the behavior of a single thread may have impacts on other threads and vice versa. The way in which threads are paired to different cores plays an essential role in utilizing hardware resources with minimal competition. On the other hand, varying workload behaviors make such paring not necessarily sustainable as time flows. Workloads may request various resources during different execution phases, such that thread scheduling should be updated correspondingly. As a solution in the MMMP, integration and adaptability are proposed to deal with such problems. We mainly explore adaptability through the OLM in previous chapters, while integration will be addressed here by aggregating both hardware and software scheduling policies together.

We have proposed the hardware scheduling policy RAPT, which works inside of cores to prioritize domestic threads; and the software scheduling policy HASP, which embeds in both the architecture and the OS to coordinate threads throughout multiple cores. In order to construct the hardware and software integrated approach, the restriction and modification will be explained in Chapter 6.1, such that all components are able to work in a harmonious environment. Finally in Chapter 6.2, performance will be presented to show the functionality and effectiveness of the ATMS.

6.1 Assembly of the ATMS

The OLM is the tie to connect the RAPT and the HASP tightly both theoretically and physically. It provides both scheduling policies with critical information for their evaluation. In particular, it predicts future LLC misses for the RAPT, such that thread priorities are decided based on the LLL. It makes the shared resources utilized more efficiently in spite of "Memory Wall". It also feeds the HASP with thread model estimators, which will indicate major phase shifts during execution. As soon as the shifts are recognized, the proposed OS scheduler evaluates mapping pattern in accordance with model information and replaces threads if necessary.

6.1.1 Synchronization of the RAPT and the HASP

Assuming the OLM is the base for both the RAPT and the HASP, they ought to utilize the OLM efficiently to their best efforts, but modifications in the OLM are mandatory to support them concurrently.

The MPMI

First of all, the MPMI is now the universal SP for both the RAPT and the HASP. The RAPT used to employ the MPKI in constructing the OLM while it now converts to MPMI. In another word, the SP is 1 million (1024×1024) instructions for both policies. It prevents the OLM from processing for two SP versions, ensures a synchronized design and saves hardware efforts. It further standardizes the data width for the most variables in the ATMS, which are now solely dependent on the WS. The reasons that the MPMI wins in the competition is:

1. The MPMI introduces slightly better performance to the HASP.
2. The MPMI reduces system overhead greatly in the ATMS.

3. The RAPT is still aware of thread behavior changes at a fine-grained level by updated L1 misses.

Consequently, the MPMI is adequate for the ATMS.

Hardware Support for the HASP

Secondly, the hardware engines support the HASP to their best. In particular, the **Sample** module is supported by the **Sampling** engine, and the **Model** module is executed by the **Regression** engine. Although some OS scheduling policies are designed in a pure software approach, we embed the HASP's ability in the hardware level to reduce its interference with normal thread execution. Therefore, more CPU resources are dedicated to normal execution, such that system performance is better ensured. Moreover, support for the HASP does not increase engines' workload, as the HASP acquires the same data as the RAPT, or the natural outcome of the engines. Therefore, it is an efficient way to support the HASP by hardware engines. However, software efforts are still critical components in our design, because they are better at overcoming core-boundaries for more comprehensive information. As a result, the HASP receives supports from both sides with justified efficiency and necessity.

The Upgraded Prioritization Engine

Thirdly, the **Prioritization** engine is upgraded to support the **Phase** module in the HASP. The engine originally was designed to compare the predicted L2 misses from the previous engine, so its capability is mostly composed of comparing integers. By increasing its capacity, the engine is expected to invoke the **Pattern** module if any change is recognized for any thread. Considering the instruction fetch policy has a tighter schedule than the OS scheduling policy, in the upgraded engine, higher pri-

ority, yet non-preemptive, is granted to the RAPT, *i.e.*, prioritization upon changes in the estimators or the predicted LLLs. It means if requests come from the RAPT and the HASP simultaneously, **Prioritization** engine will work for the RAPT first and then the HASP, otherwise FCFS. Please be advised, priority does not necessarily change every cycle, but rather the updating is triggered by a new L1 MPMI and/or new estimators.

6.1.2 Summary of the ATMS

There are in total three engines designed in the MMMP: **Sampling**, **Regression** and **Prioritization**. The **Sampling** engines are duplicated for every thread, while every core has one **Regression** engine and one **Prioritization** engine. Meanwhile, there are three modules with the same names as above engines in the RAPT, while the HASP has five modules: **Sample**, **Model**, **Phase**, **Pattern** and **Thread**. The RAPT stays in the computer architecture and its destiny is finished up to fetch engine, which is assumed as a default hardware component in SMT cores. Nevertheless, the HASP continues two extra modules in the OS and the OS has to monitor interruptions from different cores. It is obvious that modules in the OS are software efforts. In summary, we list the hardware engines and their corresponding tasks in Figure 6.1.

Furthermore, the total overhead is also provided here, which is specified to software or hardware. The software overhead refers to the time that interferes normal execution, and is measured by the complexity of the algorithm. The hardware overhead may include one or more aspects: area, storage capacity, architectural unit and/or latency. For example, storage usually is measured by the number of bits while area and power is omitted. The complete overheads are shown in Table 6.1, where TN is the number of threads, CN is the number of cores. Moreover, the

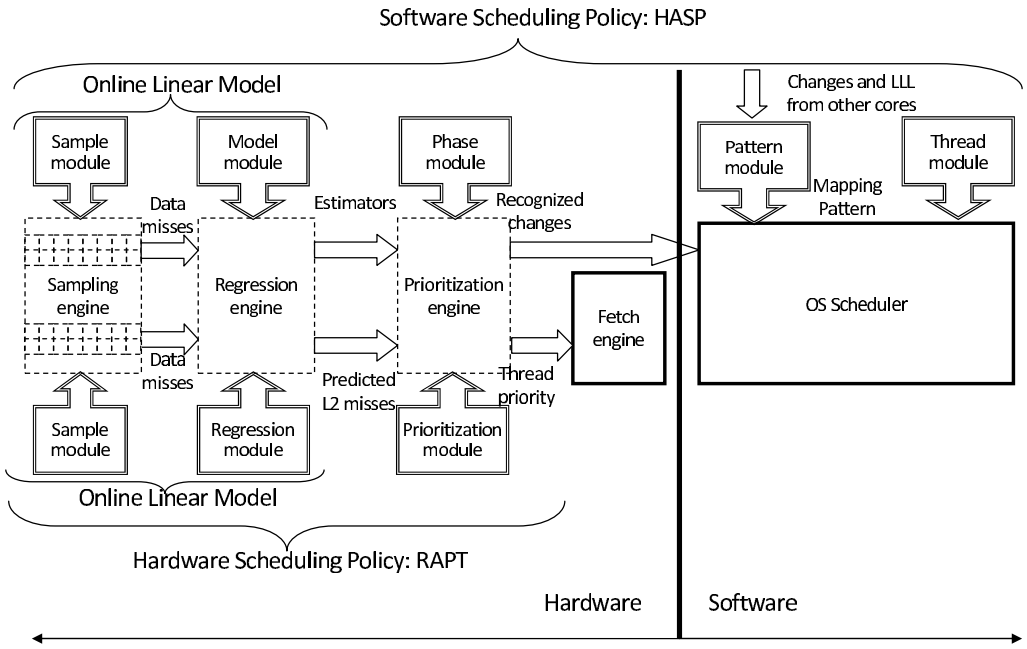


Figure 6.1: Three designed engines are in the dashed rectangles. Double-lined boxes indicate logic flow of the RAPT and the HASP. The OS scheduler and the fetch engine are considered default in the MMMP.

area, power and latency are specified to the logic units only, and the latency has no interference with CPU's normal execution. Anyway, the table tries its best to disclose the major information of the ATMS, while some details can be available after further hardware implementation and verification.

Table 6.1: Summary of overhead in the ATMS

Item	Specification
Counter	$16 \times 2 \times TN$ bits
Register	$16 \times 2 \times WS \times TN$
	$+32 \times 2 \times TN$ bits
ALU	Integer
Shifter	Integer
Multiplier	Integer
Comparator	Integer
Area	0.452mm^2
Power	0.253w
Latency	100
Complexity	$O(CN^2)$

An example of the two-core four-threaded microprocessor is shown in Figure 6.2 to illustrate the ATMS's architecture.

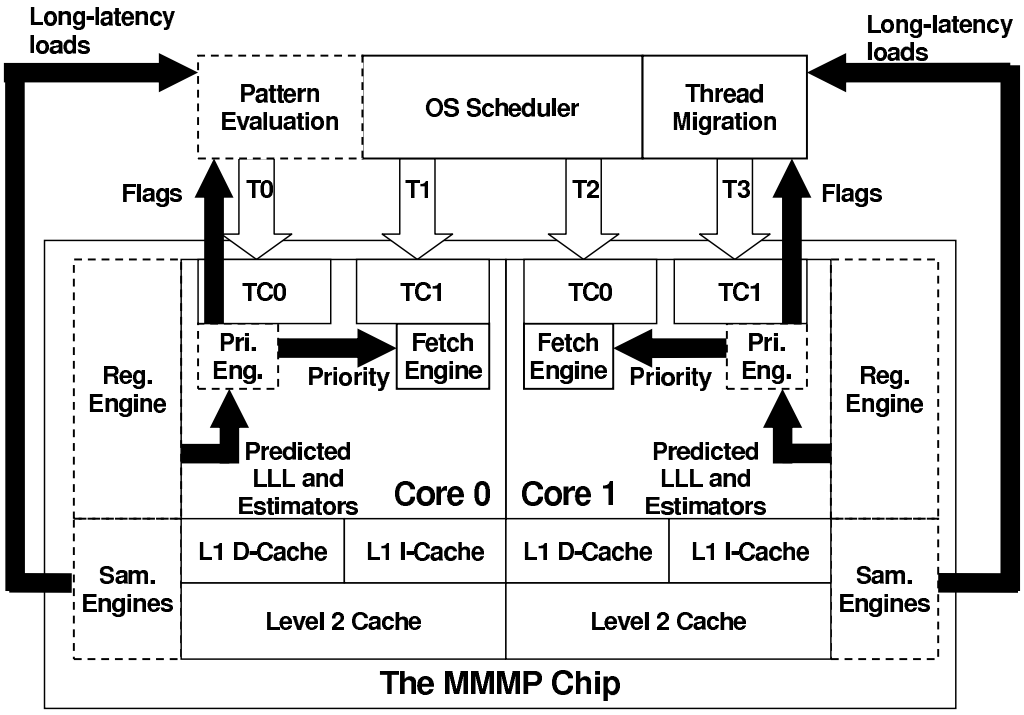


Figure 6.2: The ATMS in a tow-core four-threaded MMMP. The OLM employs Sampling (Sam.) and Regression (Reg.) engines to generate desired information. The RAPT uses an extra Prioritization (Pri.) engine to define priority for the SMT scheduler that fetches instructions. The OS scheduler is notified by Prioritization engine for phase changes and conducts evaluation and migration.

6.2 Performance Achievement

Experimental results are obtained via simulating workloads in Tables 5.4 and 5.7 in two-core and four-core configurations, respectively. Other methodologies maintain the same with that in Chapter 4.3. Configuration of the OLM in all policies are set to {1M, 32}, such that both the RAPT and the HASP are able to share its outputs in the ATMS. The epoch in the software scheduling policy is 200 million CPU cycles, and thus the HASP-III is employed in the ATMS when the system is

larger. Hence, $\delta = 1.0$ and $\theta = 0.1$ are still kept here. As the constant methodology in this study, both $avgIPC$ and $abwIPC$ are presented in Figure 6.3 for the two-core configuration and Figure 6.4 for the four-core configuration. The RAPT is not affected by the migration overhead, such that its performance is constant in all figures. The HASP is simulated here by enabling all hardware engines, but its associated instruction fetch policy for the SMT is the ICOUNT. The ATMS is the combination of the RAPT for the hardware part and the HASP for the software part, while the baseline here is the ICOUNT and the sMIX for the hardware and software scheduling, respectively.

On average, the ATMS is able to improve by 25% in $avgIPC$ and 43% in $abwIPC$ in the two-core configuration, and 21% in $avgIPC$ and 32% in $abwIPC$ in the four-core configuration; but the ability to maintain the scalability is well inherited from the HASP. This achievement is better than either independent policy. The major reason for such achievement is because of the collaboration between hardware and software. The software scheduling policy HASP tries to minimize competition in different cores, by pairing threads according to their demands, and migrating upon the changes in their behaviors. Consequently, it produces a collaborative environment in different cores by its best efforts. However, the HASP's efforts are limited by scalable evaluation designed in Chapter 5.4. It results in a scheduling that is not optimal under the condition of minimizing $StDev$, such that the hardware scheduling policy has to strive to improve system performance in an unideal environment. The RAPT coordinates threads in the same cores, such that the shared resources are more offered to efficient threads, in terms of number of possible LLLs.

In quantity, the improvement in the ATMS is not exactly equivalent to the summation of that in the RAPT and in the HASP, even though the difference is not significant. It is because that the HASP changes the thread scheduling in execution,

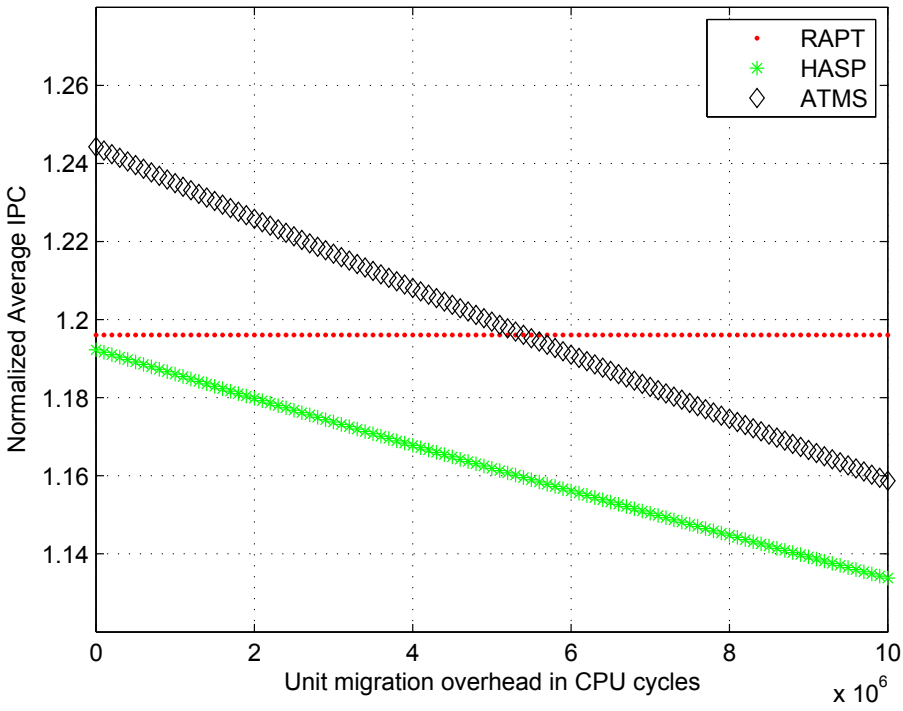
such that threads have different domestic neighbors in the HASP and in the ATMS. On the other hand, the RAPT changes the priority of threads within a core, such that accesses to caches and the main memory are changed. When LLLs are reduced by the RAPT, evaluation of thread mapping pattern is based on different memory accessing patterns, such that the HASP is not performing exactly the same in two cases. Considering the slight difference , we feel the trend is consistent enough to validate the ATMS.

An issue we notice here is the performance of RAPT is partially affected by the integration. There are two factors here: SP and system configuration. In Chapter 4, the MPKI was employed for the experiment, but here in ATMS it is the MPMI. From the perspective of OS, one million instructions are adequate to sample workload behaviors, but hardware scheduling policy may feel slow about such an SP, since it happens at cycle level. The simulated systems also have different configurations. The number of cores on the chip are increasing, but the resources lower than the LLC remain almost the same. It limits the improvement that the scheduling policies are able to make, by worsening the competition for the lower resources. They together affect the performance of the RAPT a little bit in a larger system.

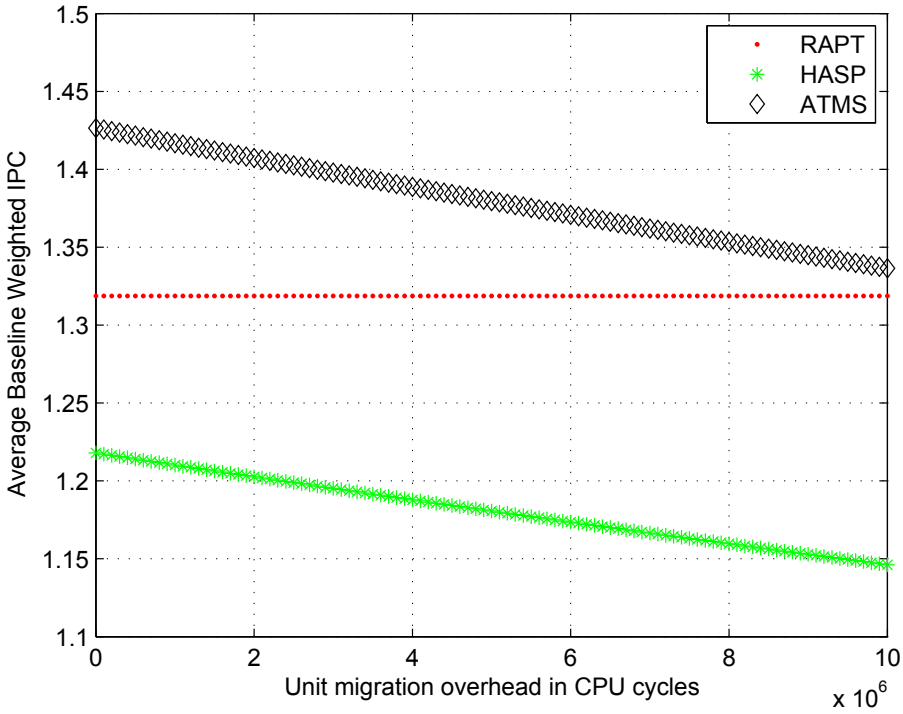
Furthermore, the improvement in the smaller system by the ATMS is more than that in the larger system. It is mostly due to two factors: first of all, the optimal mapping pattern is not guaranteed in the larger system, because the software scheduling policy searches for the sub-optimal one to increase scalability. In a system that redundant resources are available for pattern evaluating, the optimal solution is still available in observance to the objective of minimizing the $StDev$. Secondly, contention for the resources lower than the LLC is more severer in the larger system than in the smaller system. As in the configuration, the lower resources are not

augmented with increasing cores on the chip, and thus it means more threads compete for the similar amount of resources. As a result, the potential for performance improvement is limited by such resources.

Similarly to the previous chapters, the fairness of the ATMS is estimated by its outstanding $abwIPC$ results. Meanwhile, its average variance of IPCs is 0.036 in the two-core simulation and 0.039 in the four-core simulation. The average variance of IPCs for the sMIX is 0.062 and 0.035 in the two configurations, respectively.

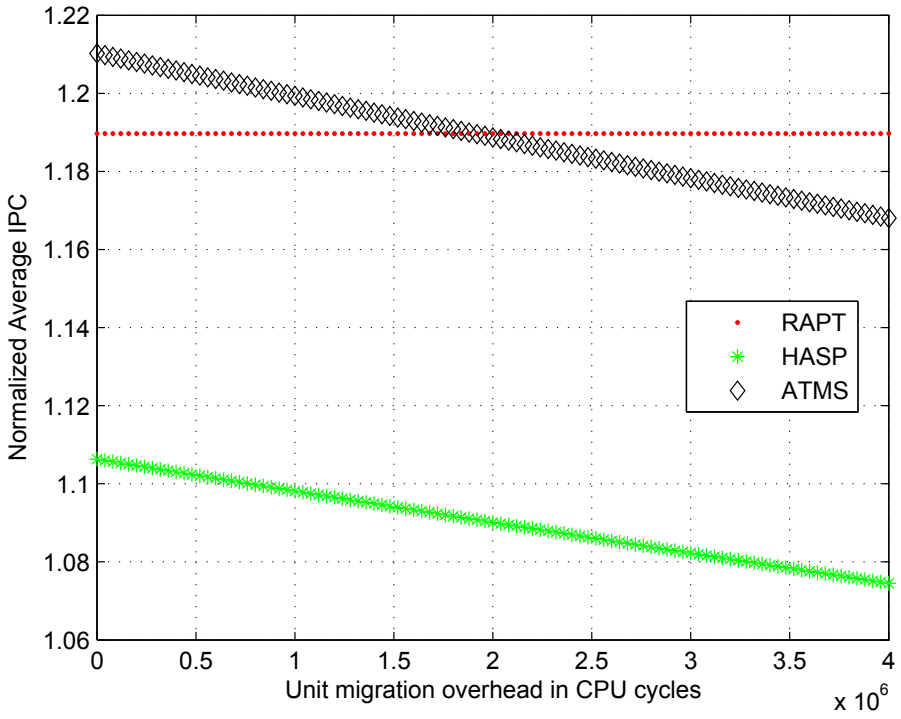


(a)

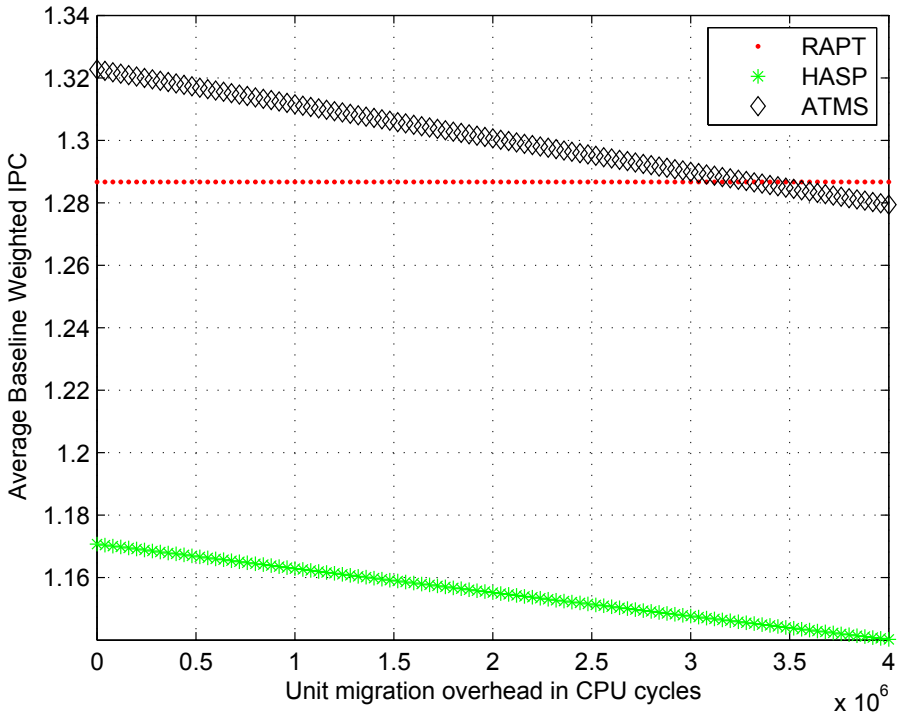


(b)

Figure 6.3: Performance of the proposed policies in a two-core four-threaded MMMP: (a) $avgIPC$, (b) $abwIPC$



(a)



(b)

Figure 6.4: Performance of the proposed policies in a four-core eight-threaded MMMP: (a) $avgIPC$, (b) $abwIPC$

CHAPTER 7

CONCLUSION AND FUTURE WORK

The complexity in resource allocation in the MMMP and the variation in workload behaviors are considered two major challenges in the architectural design. The resources are not only shared among domestic threads within a core, but also isolated by different cores. In order to manage the complicated resources in the MMMP, neither hardware nor software scheduling policies is able to take the full responsibility solely, but rather they need to collaborate to achieve the optimal results. Furthermore, the varying behaviors have great impacts on the inter-thread interference and thus the effectiveness of the scheduling policies. Hence the variation in workload behaviors further increases the difficulty in scheduling policies to respond effectively.

The Adaptive Thread Management Scheme in the MMMP was proposed in this study, which integrated both hardware and software approaches. In particular, the RAPT was the hardware scheduling policy that prioritized domestic threads in accordance with their predicted LLLs, which was provided by the OLM. Meanwhile, the HASP as the software scheduling policy evaluated the thread scheduling throughout multiple cores, and the objective was to minimize the standard deviation of LLLs among different LLCs. As an integrated design, the ATMS was a novel approach to manage threads in the MMMP.

7.1 The Problems and Solutions

Two distinguished approaches are adopted to maximize parallelism on a chip in the MMMP: the multicore architecture and the multithreaded architecture. Nevertheless, they together increase the complexity in resource distribution in the MMMP. Multiple cores are able to supply several threads, but cooperation is reduced through limiting threads' accesses to remote resources. Hence not all idle resources are helpful

to a thread, even if heavily demanded. On the other hand, the multithreaded architectures, especially the SMT, share execution resources among all domestic threads. It makes thread performance not sustainable anymore, because of the inter-thread interference, *i.e.*, threads may have impacts on each other via the shared resources. In summary, isolation and unification both exist in the MMMP.

Another motivation comes from execution phases in most workloads. It refers to virtually repeating behaviors at the granularity of millions of instructions to thousands of instructions. The phase behaviors, however, are not ideally periodic, but rather vary more or less in terms of memory accessing, phase duration and throughput. It greatly increases the difficulty in managing threads in the MMMP, since decisions ought to follow thread behaviors for better resource allocation. According to the experiments, a heuristic approach or fixed assumption could not adapt to the phase behaviors, and could not optimize the scheduling either.

As a result, complexity in resource distribution and variation in workload behaviors are spotted in the MMMP. In an effort to manage resources for optimal performance, the proposed design was equipped with integration and adaptability. The former characteristic was introduced by the collaboration between hardware and software scheduling policies, while the latter one relied on the model built by the OLS regression.

7.2 The Proposed Policies

We started this design from observing workload behaviors by the OLM, then utilizing the OLM in the hardware scheduling policy RAPT, and in the software scheduling policy HASP as well. Some modifications were implemented in an effort to embed the RAPT and the HASP in the MMMP with appropriate collaboration, which formed the main framework of the proposed design of the ATMS. Such revisions

mostly involved the synchronization of both policies, hardware assignment, and upgraded capability. Consequently, assembly of the ATMS had been accomplished, which constituted both hardware and software approaches. The ATMS ensured compatibility of the schedulers at different levels, such that system performance was optimized at different granularity.

It was examined separately that the overall system throughput and average thread improvement, *i.e.*, $avgIPC$ and $abwIPC$, respectively, while system overhead was also discussed comprehensively. In particular, the hardware scheduling policy RAPT was able to improve the system performance over the ICOUNT policy by 32%, while the software scheduling policy HASP achieved 22% more than the static scheduling policy sMIX. By implementing the ATMS in the system, it was better than the combination of the sMIX and ICOUNT by 43%, which resulted from the efforts of both the RAPT and the HASP. The fairness of the proposed scheme was justified by their outstanding $abwIPC$, as well as the average variances of thread IPCs. As a result, the proposed scheme was superior among its corresponding peers, and could be used to manage system resources very well.

7.2.1 Adaptability

As the execution phases were spotted in most workload behaviors, they provided great potential for the scheduling policies. The execution phases were of some periodic characteristics, but could not be observed easily. Some related studies might involve *a priori* knowledge to examine the execution phases, while in the proposed scheme, the phases are observed by the OLM adaptively via a machine-learning approach. The OLM was able to summarize the phase behaviors for the hardware and software scheduling policies in accordance with their granularity. Furthermore, the OLM was also capable of estimating future resource demands that are speci-

fied to every thread in terms of the cache misses in the LLC, so as to guide the scheduling policies in a proactive way. Furthermore, the proposed approach was fully supported by the optimized hardware components, and hence, its overheads were greatly minimized compared to some other designs.

7.2.2 Integration

Furthermore, components in the ATMS had a mutually beneficial relationship. It was never the case that several independent entities were forced together, but rather they required each other for a better performance.

The RAPT expected moderate competition and diverse demands within a core. The competition referred to the contention for resources lower than the LLC, which was proved to be the dominant factor affecting performance in the MMMP. Competition could be evened by the RAPT, but it was hardly able to greatly reduce it, since threads had their stable demands for off-chip resources that were inevitable in the long run. Given the demands by all threads, the best approach for high performance was to utilize everything that the MMMP had. Once such an approach was accomplished by the HASP, the RAPT was less likely to deal with severe competition. Furthermore, the demand diversity was more likely to exist by such an approach, such that the RAPT had more space to coordinate thread priorities. For example, threads waiting for the data from lower memory levels might issue less instructions to the pipeline due to pending dependency, while other threads enjoyed higher priority to utilize the pipeline resource since they were ready to proceed.

The ATMS minimized blind spots in management even if the system was large. The HASP replaced two pairs of threads from four extreme cores, while others were left for the RAPT for the sake of controllable overhead, when the system is large. Threads in those cores probably did not place extraordinary pressure on any shared

resources, so migration was not conducted on them. When the OS scheduler chose to ignore the problem, the ATMS was still able to employ the RAPT to manage the shared resources. Thread behaviors would give themselves different priorities at the instruction fetch stage, so the shared resources were still managed under the overall objective: efficient usage. Because of the integrated approach in the ATMS, scalability did not result in naive resource allocation in the MMMP.

7.2.3 Hardware Effectiveness

The ATMS improved hardware effectiveness in the sense of more achievement by the same hardware. It was common to see in architectural designs to increase hardware complexity, and even to involve OS computation [25, 26, 95], and OS scheduler designs were usually associated with few architectural efforts. What they employed was solely for either scheduling policy, while the ATMS used the OLM for both scheduling policies. Now that the ATMS made the engines eligible for both sides, it improved their efficiency significantly. Furthermore, the overhead for both policies were evaluated separately, *i.e.*, either enumerated clearly or included in performance results, which in conclusion was no more than the peer designs. Given power was analyzed at full utilization, the ATMS might be able to lead to a power-efficient system, *i.e.*, more throughput per unit power.

7.2.4 Coordinated Hardware and Software

We had reviewed a large quantity of related designs in the second chapter, which were limited in either part of the hardware of the software. Most OS schedulers were designed at the software level, while the available hardware performance counters were highly dependent on the target platform. Meanwhile, most hardware scheduling polices were weak at managing threads across cores. The demand for a

system-wide management scheme increases especially rapidly in the many-core era. The ATMS fulfilled the demand by overcoming the gap between hardware and software and between the policies at both levels. It validated a vertical communication and cooperation in the computer architecture, which would better validate some other management goals. For instance, it offered the feasibility to enforce Quality of Service (QoS). By some minor revisions, the ATMS would be able to grant a specific thread higher priority at both levels, such that it might move fast in the queue in the OS and its instructions would enter the pipeline more than others. Hence QoS of the thread could be strictly guaranteed. In summary, the ATMS provided a novel platform that coordinated both hardware and software for the thread management in the MMMP.

7.3 Future Work

There are three topics recommended for future work: other relationships in the workload model, more objectives in thread management, and hierarchical thread management in a very large-scale system.

First of all, linearity might not be the only relationship in workload behaviors, but rather some other models, *e.g.*, logarithmic or polynomial, are also interesting and promising. The employment of a new model will be motivated by research on representative workload suites, which may be typical in certain disciplines. The validation ought to be based on preliminary results, significance test and performance results. Moreover, models with no hypothesis for any specific relationship, but processing for the best available option by machine learning are very attractive too.

Secondly, about objectives in management, higher performance is not sustainable if the overhead, especially power consumption, is not considered. Therefore,

power should be taken into consideration in resource management schemes. It is not necessarily the case that power is always minimized with maximized performance, but a practical initiative is to maintain power under a threshold, while further reduce it when high performance is guaranteed. Other possible objectives may include hot spot avoidance and load balancing.

Thirdly, the management scheme can be implemented in multiple levels, such that it can be applied to a very large system, *e.g.*,cloud computing and super computers. Lower levels may be responsible for load balancing and thread paring, since the overhead is small associated with local manipulations. Higher levels will address some other topics, such as power and thermal. This would enable the scheme to manage the very large system with diverse objectives, such that it pushes the system toward an autonomic one.

BIBLIOGRAPHY

[1] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith, "Multicore resource management," *IEEE Micro*, vol. 28, no. 3, pp. 6–16, 1999.

[2] C. Liu and J.-L. Gaudiot, "The impact of resource sharing control on the design of multicore processors," in *Procs. of Algorithms and Architectures for Parallel Processing*, vol. 5574, Taipei, Taiwan, jun. 2009, pp. 315–326.

[3] D. Kang, C. Liu, and J.-L. Gaudiot, "The impact of speculative execution on SMT processors," *The International Journal of Parallel Programming*, vol. 36, pp. 361 – 385, 2008.

[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. San Francisco, CA: Morgan Kaufmann Publishers, 2011.

[5] D. H. Albonesi, R. Balasubramonian, S. G. Dropsbo, S. Dwarkadas, F. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster, "Dynamically tuning processor resources with adaptive processing," *Computer*, vol. 36, no. 12, pp. 49–58, dec. 2003.

[6] *The 3rd Generation Intel Core vPro Processor Family*, Intel, 2012. [Online]. Available: www.intel.com/content/dam/www/public/us/en/documents/white-papers/

[7] Intel, "The 2nd generation intel core processor family desktop," *Intel Datasheet*, vol. 1, p. 38, 2011.

[8] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," *IEEE Micro*, vol. 30, pp. 7 – 15, 2010.

[9] *Transforming Mission-Critical Computing*, Intel, 2011. [Online]. Available: www.intel.com/content/dam/doc/product-brief/

[10] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirosky, "Performance estimation of multistreamed, superscalar processors," in *Proceedings of the 27th Hawaii International Conference on System Sciences*, vol. 1, jan. 1994, pp. 195 – 204.

[11] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proc. of 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, jun. 1995, pp. 392–403.

[12] S. J. Eggers, J. S. Emer, H. M. Leby, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *Micro, IEEE*, vol. 17, no. 5, pp. 12 –19, sep/oct 1997.

[13] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on SMT processors," in *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, LA, sep. 2003, pp. 15–25.

[14] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *IEEE Transactions On Computer*, pp. 79 – 85, 1997.

[15] H. P. Hosftee, "Introduction to the cell broadband engine," *IBM White Paper*, 2005.

[16] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, and et al., "A 48-core IA-32 processor with on-die message-passing and DVFS in 45nm CMOS," in *2010 IEEE Asian Solid State Circuits Conference*, nov. 2010, pp. 1 –4.

[17] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. Stamm, "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proc. of 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, may 1996, pp. 191–202.

[18] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, 4th ed. San Francisco, CA: Morgan Kaufmann Publishers, 2007.

[19] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, Santa Clara, CA, 1991, pp. 176–188.

[20] D. Patterson, "Latency lags bandwidth," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, San Jose, CA, oct. 2005, pp. 71–75.

[21] D. Tullsen and J. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *Proc. of 34th ACM/IEEE International Symposium on Microarchitecture*, Austin, TX, dec. 2001, pp. 318–327.

[22] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernández, "Dynamicall controlled resource allocation in SMT processor," in *Proceedings of 37th International Symposium on Microarchitecture*, Portland, OR, dec. 2005, pp. 171–182.

[23] A. El-Moursy and D. H. Albonesi, "Front-end policies for improved issue efficiency in SMT processors," in *Proceedings of 9th International Symposium on High-Performance Computer Architecture*, Anaheim, CA, feb. 2003, pp. 31–40.

[24] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernádez, "Dcache warn: an i-fetch policy to increase SMT efficienty," in *Proceedings of 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, apr. 2004, pp. 74–83.

[25] H. Wang, I. Koren, and C. M. Krishna, "Utilization-based resource partitioning for power-performance efficiency in SMT processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 7, pp. 1150–1163, jul. 2011.

[26] S. Choi and D. Yeung, "Learning-based SMT processor resource distribution via hill-climbing," in *Proceedings of The 33rd Interntional Symposium on Computer Architecture*, Boston, MA, jul. 2006, pp. 239–251.

[27] H. Wang, R. Sangireddy, and S. Baldawa, "Optimizing instruction scheduling through combined in-order and o-o-o execution in smt processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 389–403, mar. 2009.

[28] S. C. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, jun. 1995, pp. 24 –36.

[29] S. P. E. Corporation. (2005) SPECjbb 2005 java server benchmark. [Online]. Available: http://www.spec.org/jbb2005/

[30] E. Sarhan, A. Ghalwash, and M. Khafagy, "Specification and implementation of dynamic web site benchmark in telecommunication area," in *Proceedings of*

the 12th WSEAS international conference on Computers, Heraklion, Greece, 2008, pp. 863–867.

[31] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong, "Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, dec. 2010, pp. 337 –348.

[32] H. Cheng, C. Lin, J. Li, and C. Yang, "Memory latency reduction via thread throttling," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, dec. 2010, pp. 53–64.

[33] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, june 2009, pp. 290–301.

[34] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread to strand binding of parallel network applications in massive multi-threaded systems," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, january 2010, pp. 191–202.

[35] R. Kokku, T. L. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. M. Vin, "A case for run-time adaptation in packet processing systems," *SIGCOMM Computer Communnication Review*, vol. 34, no. 1, pp. 107–112, January 2004.

[36] T. Wolf, N. Weng, and C.-H. Tai, "Design considerations for network processor operating systems," in *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, Princeton, NJ, october 2005, pp. 71–80.

[37] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Lisbon, Portugal, march 2007, pp. 47–58.

[38] R. McGregor, C. Antonopoulos, and D. Nikolopoulos, "Scheduling algorithms for effective thread pairing on hybrid multiprocessors," in *Proceedings of The 19th IEEE International Parallel and Distributed Processing Symposium*, april 2005, pp. 28 – 37.

[39] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proceedings of the 38th annual international symposium on Computer architecture*, San Jose, CA, 2011, pp. 283–294.

[40] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González, "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Ontario, Canada, october 2008, pp. 240–249.

[41] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, Porto Alegre, Brazil, december 2011, pp. 362–373.

[42] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," *ACM Sig-Metrics Performance Evaluation Review*, vol. 30, pp. 66 – 76, 2002.

[43] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using os observations to improve performance in multicore systems," *Micro, IEEE*, vol. 28, no. 3, pp. 54 –66, may-june 2008.

[44] A. Fedorova and M. Seltzer, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, september 2007, pp. 25 –38.

[45] C. Luque, M. Moreto, F. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "Itca: Inter-task conflict-aware cpu accounting for cmps," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, Raleigh, NC, september 2009, pp. 203 –213.

[46] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, Pittsburgh, PA, march 2010, pp. 129 –141.

[47] L. Weng and C. Liu, "On better performance from scheduling threads according to resource demands in MMMP," in *Proc. of 16th International Workshop*

on Scheduling and Resource Management for Parallel and Distributed Systems, in con. with ICPP'10, San Diego, CA, sep. 2010, pp. 339–345.

[48] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Optimal task assignment in multithreaded processors: a statistical approach," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, London, England, UK, march 2012, pp. 235–248.

[49] C. Acosta, F. Cazorla, A. Ramirez, and M. Valero, "Thread to core assignment in SMT on-chip multiprocessors," in *The 21st International Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brazil, oct. 2009, pp. 67 – 74.

[50] A. El-Moursy, R. Garg, D. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a CMP of multi-threaded processors," in *The 20th International Parallel and Distributed Processing Symposium*, Rhodes, Greece, apr. 2006, pp. 1–10.

[51] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring), Atlantic City, NJ, 1967, pp. 483–485.

[52] T. Morad, U. Weiser, A. Kolodnyt, M. Valero, and E. Ayguade, "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," *Computer Architecture Letters*, vol. 5, no. 1, pp. 14 –17, jan.-june 2006.

[53] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, Paris, France, april 2010, pp. 125–138.

[54] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, march 2009, pp. 253–264.

[55] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in smt processors," *ACM SIGPLAN Notice*, vol. 44, no. 3, pp. 133–144, march 2009.

[56] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *2012 IEEE International*

Symposium on Performance Analysis of Systems and Software, april 2012, pp. 145 –155.

[57] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for smt processor scheduling," *ACM Transactions on Architecture and Code Optimization*, vol. 38, no. 1, pp. 91–102, march 2010.

[58] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *The 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, december 2006, pp. 423 –432.

[59] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Ontario, Canada, october 2008, pp. 208–219.

[60] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, France, june 2010, pp. 60–71.

[61] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer, "CRUISE: cache replacement and utility-aware scheduling," in *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems*, London, England, UK, march 2012, pp. 249–260.

[62] S. Cho and L. Jin, "Managing distributed, shared l2 caches through OS-level page allocation," in *The 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, december 2006, pp. 455 – 468.

[63] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*, Nuremberg, Germany, 2009, pp. 89–102.

[64] J. Weinberg and A. Snavely, "Symbiotic space-sharing on SDSC's datastar system," in *The 12th Workshop on Job Scheduling Strategies for Parallel Processing*, 2006, pp. 192 – 209.

[65] A. C. Sodan and L. Lan, "LOMARC: Lookahead matchmaking for multiresource coscheduling on hyperthreaded CPUs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 11, pp. 1360 –1375, nov. 2006.

[66] E. Frachtenberg, G. Feitelson, F. Petrini, and J. Fernandez, "Adaptive parallel job scheduling with flexible coscheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 11, pp. 1066 – 1077, nov. 2005.

[67] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. San Francisco, CA: John Wiley & Sons, Inc., 2008.

[68] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, London, UK, 2012, pp. 261–272.

[69] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of mpi programs," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, San Diego, CA, jun 2003, pp. 84–94.

[70] F. Karablieh, R. Bazzi, and M. Hicks, "Compiler-assisted heterogeneous checkpointing," in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, 2001.

[71] D. G. von Bank, C. M. Shub, and R. W. Sebesta, "A unified model of pointwise equivalence of procedural computations," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1842–1874, nov 1994.

[72] Z. Zhu and Z. Zhang, "A performance comparison of dram memory system optimizations for smt processors," in *The 11th International Symposium on High-Performance Computer Architecture*, feb. 2005, pp. 213 – 224.

[73] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable performance in smt processors: synergy between the os and smts," *Computers, IEEE Transactions on*, vol. 55, no. 7, pp. 785 – 799, july 2006.

[74] S. Sair and M. Charney, "Memory behavior of the SPEC2000 benchmark suite," IBM, Research Report, oct. 1999.

[75] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, LA, september 2003, pp. 220–231.

[76] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *The IEEE 14th International Symposium on High Performance Computer Architecture*, New Orleans, LA, february 2008, pp. 367 –378.

[77] Y. Xie and G. H. Loh, "Dynamic classificaion of program memory behaviors in cmps," in *The 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Beijing, China, june 2008, pp. 340–351.

[78] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, feburary 2005, pp. 340–351.

[79] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th annual international symposium on Computer architecture*, San Diego, CA, june 2007, pp. 381–391.

[80] J. Chen, L. K. John, and D. Kaseridis, "Modeling program resource demand using inherent program characteristics," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 1–12, june 2011.

[81] C. Pereira, J. Lau, B. Calder, and R. Gupta, "Dynamic phase analysis for cycle-close trace generation," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Jersey City, NJ, September 2005, pp. 321–326.

[82] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, december 2006, pp. 359–370.

[83] J. Lau, S. Schoenmackers, and B. Calder, "Transition phase classification and prediction," in *The 11th International Symposium on High-Performance Computer Architecture*, San Antonio, TX, february 2005.

[84] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th annual international symposium on Computer architecture*, San Diego, CA, june 2003, pp. 336–349.

[85] J. K. Hollingsworth, "An online computation of critical path profiling," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, Philadelphia, PA, may 1996, pp. 11–20.

[86] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, OR, november 2009, pp. 1–12.

[87] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, London, UK, march 2012, pp. 223–234.

[88] B. Fields, R. Bodik, and M. Hill, "Slack: maximizing performance under technological constraints," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, may 2002, pp. 47 –58.

[89] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proceedings of 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 74 –85.

[90] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European conference on Computer systems*, Paris, France, april 2010, pp. 139–152.

[91] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: a scheduler for heterogeneous multicore systems," *SIGOPS Operating System Review*, vol. 43, no. 2, pp. 66–75, april 2009.

[92] T. T. Soong, *Fundamentals of probability and statistics for engineers.* Hoboken, NJ: John Wiley & Sons, Incorporated, 2004.

[93] R. D. Yates and D. J. Goodman, *Probalility and stochastic processes: a friendly introduction for electrical and computer engeineers*, 2nd ed. San Francisco, CA: Wiley, John & Sons, Incorporated, 2004.

[94] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[95] R. Bitirgen, E. Ipek, and J. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *The 41st IEEE/ACM International Symposium on Microarchitecture*, nov. 2008, pp. 318–329.

[96] B. Hickmann, A. Krioukov, M. Schulte, and M. Erle, "A parallel ieee p754 decimal floating-point multiplier," in *The 25th International Conference on Computer Design*, oct. 2007, pp. 296–303.

[97] B. Curran, B. McCredie, L. Sigal, E. Schwarz, B. Fleischer, Y.-H. Chan, D. Webber, M. Vaden, and A. Goyal, "4GHz+ low-latency fixed-point and binary floating-point execution units for the POWER6 processor," in *IEEE International Solid-State Circuits Conference*, feb. 2006, pp. 1728–1734.

[98] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005, http://sesc.sourceforge.net.

[99] P. M. Ortego and P. Sack. (2004) SESC: Super ESCalar simulator. [Online]. Available: http://sesc.sourceforge.net/sescdoc.pdf

[100] J. Veenstra and R. Fowler, "MINT: a front end for efficient simulation of shared-memory multiprocessors," in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Durham, NC, feb. 1994, pp. 201 –207.

[101] D. A. Patterson and C. H. Sequin, "RISC I: A reduced instruction set VLSI computer," in *Proceedings of the 8th annual symposium on Computer Architecture*, Minneapolis, MN, 1981, pp. 443–457.

[102] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *SIGARCH Comput. Archit. News*, vol. 8, no. 6, pp. 25–33, oct. 1980.

[103] J. Renau, B. Fraguela, and L. Wei. (2002) SESC: Super ESCalar simulator. [Online]. Available: http://sesc.sourceforge.net/slide1.pdf

[104] Y. Sazeides and T. Juan, "How to compare the performance of two SMT microarchitectures," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, aug. 2001, pp. 180–183.

[105] M. Gulati and N. Bagherzadeh, "Performance study of a multithreaded superscalar microprocessor," in *Proceedings of 2nd International Symposium on High-Performance Computer Architecture*, San Jose, CA, feb. 1996, pp. 291–301.

[106] S. Hily and A. Seznec, "Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading," IRISA, Tech. Rep., feb. 1997.

[107] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proc. of 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, LA, oct. 2003, pp. 244–255.

[108] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Journal of Instruction Level Parallelism*, no. 7, 2005, pp. 1 – 28.

[109] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Canada, october 2008, pp. 220–229.

[110] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi, "The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 7, pp. 1192 –1205, july 2011.

[111] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "AKULA: a toolset for experimenting and developing thread placement algorithms on multicore systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria, sep. 2010, pp. 249–260.

VITA

LICHEN WENG

February 13, 1986	Born, Nanjing, Jiangsu, China
2008	B.E., Electrical Engineering and Automation Nanjing Normal University Nanjing, Jiangsu, China
2008–2011	Doctoral Student Florida International University Miami, Florida
2008–2009	Teaching Assistant Florida International University Miami, Florida
2008–2012	Research Assistant Florida International University Miami, Florida
2011–2012	PhD Candidate Florida International University Miami, Florida
2012	M.S., Computer Engineering Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

D. Arteaga, M. Zhao, C. Liu, P. Thanarungroj, and L. Weng, "Cooperative Virtual Machine Scheduling on Multi-core Multi-threading Systems - A Feasibility Study". In *Workshop on Micro Architectural Support for Virtualization, Data Center Computing and Clouds, in conjunction with MICRO–43*, Atlanta, GA, December 5, 2010.

L. Weng, Quan, G. and C. Liu, "PCOUNT: A Power Aware Fetch Policy in Simultaneous Multithreading Processors". In *The 1st International IEEE Workshop on Thermal Modeling and Management: Chips to Data Centers, in conjunction with IGCC 2011*, Orlando, FL, July 25, 2011.

L. Weng, and C. Liu, "On Better Performance from Scheduling Threads according to Resource Demands in MMMP". In *The 6th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems, in conjunction*

with ICPP'10, San Diego, CA, September 13, 2010.

L. Weng, X. Niu and C. Liu, "Intelligent Controller with Cache for Critical Infrastructures". In *The IET International Conference on Frontier Computing -Theory, Technologies and Applications*, Taichung, Taiwan, August 4–6, 2010.

L. Weng, and C. Liu, "Fetching According to the Evaluated L2 Cache Misses By OLS Regression in SMT Architecture". In *Poster Session in the 16th Conference on Architecture Support for Programming Languages and Operating Systems*, Newport Beach, CA, March 6, 2011.

L. Weng, and C. Liu, "Hardware-aided Monitoring of L1 and L2 D-Cache Misses in SMT". In *Poster Session in the 3rd Workshop on Functionality of Hardware Performance Monitoring, in conjunction with MICRO-43*, Atlanta, GA, December 4, 2010.