ADAPTIVE THREAD AND MEMORY ACCESS SCHEDULING IN CHIP MULTIPROCESSORS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

İsmail Aktürk

July, 2013

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Özcan Öztürk(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Güdükbay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

ADAPTIVE THREAD AND MEMORY ACCESS SCHEDULING IN CHIP MULTIPROCESSORS

İsmail Aktürk
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Özcan Öztürk
July, 2013

The full potential of chip multiprocessors remains unexploited due to architecture oblivious thread schedulers used in operating systems, and thread-oblivious memory access schedulers used in off-chip main memory controllers. For the thread scheduling, we introduce an adaptive cache-hierarchy-aware scheduler that tries to schedule threads in a way that inter-thread contention is minimized. A novel multi-metric scoring scheme is used that specifies the L1 cache access characteristics of a thread. The scheduling decisions are made based on multi-metric scores of threads. For the memory access scheduling, we introduce an adaptive compute-phase prediction and thread prioritization scheme that efficiently categorize threads based on execution characteristics and provides fine-grained prioritization that allows to differentiate threads and prioritize their memory access requests accordingly.

Keywords: Adaptive scheduling, chip multiprocessors, inter-thread contention, thread phase prediction, multi-metric scoring.

ÖZET

ÇOK ÇEKİRDEKLİ İŞLEMCİLERDE UYARLAMALI İŞ PARÇACIĞI VE BELLEK ERİŞİMİ ÇİZELGELEME

İsmail Aktürk
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doç. Dr. Özcan Öztürk
Temmuz, 2013

İşletim sistemlerinde kullanılan mimari-kayıtsız iş parçacığı çizelgeleyicileri ve iş parçacığı-kayıtsız harici ana bellek erişim kontrol birimi çizelgeleyicileri nedeniyle çok çekirdekli işlemcilerin potansiyelleri tam olarak değerlendirilememektedir. Bu problemın çözümüne yönelik olarak iş parçacığı çizelgeleme için, iş parçacıkları arası çekişmeyi en aza indirmeyi amaçlayan ve bu doğrultuda iş parçacıkları çizelgeleyen uyarlamalı önbellek-sıradüzeni-farkında iş parçacığı çizelgeleyicisi sunulmuştur. Süreçlerin birinci seviye önbellek kullanım özelliklerini belirten yeni çok-metrikli puanlama tekniği geliştirilmiş ve kullanılmıştır. Sunulan çizelgeleyicide, çizelgeleme iş parçacıkların çok-metrikli puanlamaları esas alınarak gerçekleştirilir. Harici ana bellek erişim çizelgelemesi için ise, iş parçacıkları etkin olarak yürütme özelliklerine göre gruplandıran ve iş parçacıkları hassas olarak önceliklendirebilen uyarlamalı işlem fazı tahmini ve iş parçacığı önceliklendirme yöntemi sunulmuştur.

Anahtar sözcükler: Uyarlamalı çizelgeleme, çok çekirdekli işlemciler, iş parçacıkları-arası çekişme, iş parçacığı fazı öngörüsü, çok-metrikli puanlama.

All praise is due to God alone, the Sustainer of all the worlds,
the Most Gracious, the Dispenser of Grace

Acknowledgement

I would like to express my appreciation to my advisor Assoc. Prof. Dr. Özcan Öztürk for his support, invaluable advices, suggestions and steadfast guidance during my study at Bilkent University.

Also, I would like to thank Assoc. Prof. Dr. Uğur Güdükbay and Assoc. Prof. Dr. Süleyman Tosun for their time to be a part of my thesis committee.

Finally, I would like to express my heartfelt gratitude to my family for their patience and support, especially to Ayşegül.

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Document Organization 3

2 Related Work **5**

2.1 Thread Scheduling 5

2.1.1 Replacement and Partitioning 6

2.1.2 Cache-sharing-aware Scheduling 7

2.1.3 Phase Prediction and Thread Classification 8

2.1.4 Coscheduling 9

2.2 Memory Access Scheduling 11

3 Adaptive Cache-Hierarchy-Aware Thread Scheduling **14**

3.1 Introduction 14

3.2 Problem Statement 16

3.3 Motivation 18

3.4 Contributions 20
3.5 Chip Multiprocessors of Simultaneous Multithreading 21
3.6 Inter-thread Contention and Slowdown 22
3.7 Performance Counters and Monitoring 22
3.8 Phase Detection and Prediction 23
3.9 Multi-metric Scoring Scheme 24
3.9.1 Scalability of Mutli-metric Scoring Scheme 26
3.10 Adaptive Thread Scheduling 26

4 Adaptive Compute-phase Prediction and Thread Prioritization 33

4.1 Introduction 33
4.2 Problem Statement 34
4.3 Motivation 35
4.4 Contributions 35
4.5 Memory Model 36
4.6 Adaptive Compute-phase Prediction 38
4.7 Adaptive Thread Prioritization 44

5 Evaluations 46

5.1 Adaptive Cache-hierarchy-aware Thread Scheduling 46
5.1.1 Simulation Environment 46
5.1.2 The Effect of Scheduling on System Performance 49

5.1.3 Slowdown of Benchmarks 51

5.1.4 The Effect of Scheduling on Cache Performance 54

5.1.5 Sensitivity of Performance to the Thread Quantum 59

5.1.6 Sensitivity of Performance to the Weights of Thread Attributes 60

5.1.7 Sensitivity of Performance to the Resolution of Thread Attributes 61

5.1.8 Sensitivity of Performance to Scoring Thresholds 62

5.2 Adaptive Compute-phase Prediction and Thread Prioritization . . 63

5.2.1 Simulation Environment 63

5.2.2 Workloads 64

5.2.3 The Effect of Scheduling on Sum of Execution Times . . . 65

5.2.4 The Effect of Scheduling on Power Consumption 69

6 Conclusions and Future Work **72**

Bibliography **76**

Appendix **82**

A Extended Evaluations for Adaptive Cache-Hierarchy-Aware Thread Scheduling **82**

A.1 L1 Hits/Misses Variations 83

A.2 L1 Hit Ratio Variations 88

A.3 L2 Hits/Misses Variations 93

A.4 L2 Hit Ratio Variations 95

A.5 Performance Variations of Benchmarks 97

Vita **113**

List of Figures

3.1 The L2 miss variation of four threads running on two cores under different scheduling schemes. 16

3.2 The IPC variation of four threads running on two cores under different scheduling schemes. 17

3.3 Typical chip multiprocessor architecture and its memory subsystem. 21

3.4 Attribute vector expresses execution characteristics of a thread. . 24

3.5 Attribute vector that expresses execution characteristics of a thread in higher resolution. 26

3.6 The flow of adaptive cache-hierarchy-aware thread scheduling. . . 27

3.7 Coscheduling score calculation. 28

3.8 Cache-hierarchy-unaware scheduling. The number of L2 hits is four, while the number of L2 misses is 17. 31

3.9 Cache-hierarchy-aware scheduling. The number of L2 misses reduced to 14. 32

4.1 Typical memory bank architecture found in a DRAM rank. 37

4.2 Default compute-phase prediction. 41

4.3 Adaptive compute-phase prediction. 42

4.4 Updating a threshold in adaptive compute-phase prediction. . . . 43

5.1 Performance of benchmarks under different scheduling schemes. . 50

5.2 Comparison of overall system performance under different scheduling schemes. 51

5.3 Slowdowns of benchmarks under different scheduling schemes. . . 54

5.4 Miss per kilo instructions vs. L2 cache size for the benchmarks. . 55

5.5 L2 miss per kilo instruction of benchmarks under different scheduling schemes. 56

5.6 L2 miss ratio of benchmarks under different scheduling schemes. . 57

5.7 L1 miss per kilo instruction under different scheduling schemes. . 58

5.8 L1 miss ratio of benchmarks under different scheduling schemes. . 59

5.9 System performance for different thread quantum lengths. 60

5.10 System performance changes with respect to relative weights of thread attributes. 61

5.11 System performance with respect to the number of bits used to represent each attribute in an attribute vector. 62

5.12 Comparison of CP-WO and ACP schedulers for the given workloads. Sum of execution times is normalized with respect to FR-FCFS. 67

5.13 Comparison of CP-WO and ATP schedulers for the given workloads. Sum of execution times is normalized with respect to FR-FCFS. 67

5.14 Comparison of CP-WO and ACP-TP schedulers for the given workloads. Sum of execution times is normalized with respect to FR-FCFS. 67

5.15 Total sum of execution times for the given workloads. 68

5.16 Comparison of total memory system power of CP-WO and ACP-TP schedulers for the given workloads. The total memory system power is normalized with respect to FR-FCFS. 69

5.17 Comparison of total system power of CP-WO and ACP-TP schedulers for the given workloads. The total system power is normalized with respect to FR-FCFS. 70

5.18 Sum of total memory system power for the given workloads. . . . 70

5.19 Sum of total system power consumption for the given workloads. . 71

A.1 L1 hits and misses under static schedule 1. 83

A.2 L1 hits and misses under static schedule 2. 83

A.3 L1 hits and misses under static schedule 3. 84

A.4 L1 hits and misses under static schedule 4. 84

A.5 L1 hits and misses under static schedule 5. 85

A.6 L1 hits and misses under static schedule 6. 85

A.7 L1 hits and misses under static schedule 7. 86

A.8 L1 hits and misses under dynamic-offline schedule. 86

A.9 L1 hits and misses under adaptive cache-hierarchy-aware scheduling. 87

A.10 L1 hit ratio under static schedule 1. 88

A.11 L1 hit ratio under static schedule 2. 88

A.12 L1 hit ratio under static schedule 3. 89

A.13 L1 hit ratio under static schedule 4. 89

A.14 L1 hit ratio under static schedule 5. 90

A.15 L1 hit ratio under static schedule 6. 90

A.16 L1 hit ratio under static schedule 7. 91

A.17 L1 hit ratio under dynamic-offline schedule. 91

A.18 L1 hit ratio under adaptive cache-hierarchy-aware scheduling. . . 92

A.19 L2 hits and misses under static scheduling schemes. 93

A.20 L2 hits and misses under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 94

A.21 L2 hit ratio under static scheduling schemes. 95

A.22 L2 hit ratio under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 96

A.23 IPC of *blackscholes* under static scheduling schemes. 97

A.24 IPC of *blackscholes* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 98

A.25 IPC of *vips* under static scheduling schemes. 99

A.26 IPC of *vips* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 100

A.27 IPC of *canneal* under static scheduling schemes. 101

A.28 IPC of *canneal* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 102

A.29 IPC of *dedup* under static scheduling schemes. 103

A.30 IPC of *dedup* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 104

A.31 IPC of *facesim* under static scheduling schemes. 105

A.32 IPC of *facesim* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 106

A.33 IPC of *x264* under static scheduling schemes. 107

A.34 IPC of *x264* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 108

A.35 IPC of *fluidanimate* under static scheduling schemes. 109

A.36 IPC of *fluidanimate* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 110

A.37 IPC of *freqmine* under static scheduling schemes. 111

A.38 IPC of *freqmine* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes. 112

List of Tables

4.1 Prioritization policy for the read requests. 45

5.1 Chip multiprocessor and memory configuration for evaluations. . . 47

5.2 PARSEC benchmarks used in evaluations. 47

5.3 Static schedules used in evaluations. 48

5.4 Slowdown of a thread when scheduled with another thread on the same core. 52

5.5 Slowdown of a thread when scheduled with another thread under the static scheduling scheme. 52

5.6 Slowdown of a thread when scheduled with another thread under the dynamic-offline scheduling scheme. 53

5.7 Slowdown of a thread when scheduled with another thread under the cache-hierarchy-aware scheduling. 53

5.8 System performance with respect to the attribute thresholds for threads. 63

5.9 System configurations used in evaluations. 64

5.10 Workloads used in evaluations. 66

Chapter 1

Introduction

1.1 Overview

The number of transistors available on a die is no longer increasing according to Moore's Law [1] due to power constraints and diminishing returns. However, the demand for increased performance and higher throughput is still in place. To provide higher throughput and increased performance without bumping into physical limits of Moore's Law, novel multiprocessor architectures have emerged, including chip multiprocessors that contains multiple cores on a single chip [2]. Another way to provide higher throughput and increased performance is to run more than one thread on each core with multithreading, namely simultaneous multithreading [3]. The choice of threads to be scheduled on the same core has signicant impact on overall system performance. Inter-thread contention occurs since coscheduled threads are competing for shared resources. The primary shared resource that influence the performance is the cache. An efficient scheduling should minimize the contention for shared caches to maximize utilization and system performance. Since the execution characteristics of threads varies over time, the scheduling decision has to be remade based on provisioned behaviors of threads for the near future.

The frequency and power walls have forced chip manufacturers to change their

design philosophy from uniprocessors to chip multiprocessors. While multicore architectures provide higher aggregated throughput, the underlying memory subsystem remains a performance bottleneck. The memory subsystems operate in lower frequencies and they have to serve to multiple threads running on different cores simultaneously. This creates a contention on memory subsystem and has a significant impact on the overall system performance. Traditional memory access scheduling algorithms designed for uniprocessors are inadequate for chip multiprocessors. For this reason, an efficient memory access scheduler is required to exploit the performance promises of chip multiprocessors.

In response to increased pressure on memory subsystem due to the memory requests generated by multiple threads, an efficient memory access scheduler has to fulfill the following goals:

- serve memory requests in a way that cores are kept as busy as possible
- organize the requests in a way that the memory bus idle-time is reduced

The work done in the scope of this thesis is given in two parts. In the first part of the thesis, we focus on thread scheduling and introduce an adaptive cache-hierarchy-aware scheduling algorithm. The proposed algorithm uses hardware counters that provide statistics regarding cache access pattern of each thread and employs an intelligent scheduling decision mechanism that tries to schedule threads in a way that inter-thread contention is minimized. The originality of this work is the use of multi-metric scoring scheme that specifies the L1 cache access characteristics of a thread. The scheduling decisions are made based on these characteristics. While previous studies are focused on the performance of last-level cache (LLC) to optimize scheduling decisions, our evaluations indicate that the eventual performance of LLC is dependent on how the upper levels of cache hierarchy are used. Thus, adaptive cache-hierarchy-aware scheduling effectively utilizes upper levels of cache, and thereby improves the throughput and maximizes the system performance.

In the second part of the thesis, we focus on memory access scheduling and

introduce an adaptive compute-phase prediction and thread prioritization algorithm. The necessity of distinguishing threads based on their execution characteristics is addressed by recent studies. In such studies, threads are categorized into two groups, namely memory-non-intensive (i.e., threads in compute phase), and memory-intensive (i.e., threads in memory-phase). Saturation counters provide effective metrics to determine the execution phase of a thread. However, they become slower to react (i.e., classifying threads in a timely manner) when fix thresholds are used. Adaptive compute-phase prediction scheme determines the proper thresholds on the fly, leading to better classification of threads in a timely manner. Although, distinguishing threads of different groups and prioritizing one group over another improve the performance, the potential performance gain is missed due to the inability to differentiate threads in the same group. Adaptive thread prioritization scheme provides fine-grained prioritization that allows to differentiate the threads in the same group and prioritize them accordingly.

1.2 Document Organization

The organization of the thesis is as follows. The related work is given in Chapter 2. We discuss about related work on thread scheduling in Section 2.1, including cache replacement and partitioning algorithms, coscheduling methods. In Section 2.2, we discuss about related work on thread classification and prioritization, and memory access scheduling.

We introduce our proposed adaptive cache-hierarchy-aware scheduling algorithm and give the implementation details in Chapter 3. We discuss multi-metric scoring scheme and adaptive thread scheduling in Sections 3.9 and 3.10, respectively. Then, we introduce our proposed adaptive compute-phase prediction and thread prioritization algorithm in Chapter 4.

We provide extensive evaluations in Chapter 5 to present and analyze the effectiveness of proposed thread and memory access schedulers. In Section 5.1, we provide experimental results regarding adaptive cache-hierarchy-aware scheduler.

We analyze the effectiveness of adaptive compute-phase prediction and thread prioritization in Section 5.2.

We conclude the thesis and provide future work in Chapter 6. Appendix A provides extended evaluations for adaptive cache-hierarchy-aware thread scheduler.

Chapter 2

Related Work

2.1 Thread Scheduling

Tremendous research efforts have been made on scheduling over the last several decades. Although its long history, scheduling is still relevant and it is one of the most important aspects of computing. The shift from single chip processor to chip multiprocessor made scheduling problem even interesting and compelling. Jiang et al. [4] proved that scheduling in chip multiprocessors where the core number is grater than two is an NP-complete problem. For this reason, there are numerous heuristics developed for scheduling in chip multiprocessors.

There are three main concerns regarding scheduling. The first one is to improve the computing efficiency (e.g., [5], [6]). The second concern is fairness (e.g., [7]) and the last one is performance isolation (e.g., [8]). There are vast amount of studies targeted these concerns.

Deciding threads to be coscheduled is one part of the story. In addition to that, there is also need to decide the amount of resource to be allocated to each thread. To this end various replacement and cache partitioning strategies have been proposed. Notice that, scheduling algorithms are not alternatives to replacement and cache partitioning strategies; however, they all have impact on

each other and overall system performance.

2.1.1 Replacement and Partitioning

The threads scheduled on the same core compete for shared cache resources. A request from a thread can conflict with a request from another one. A thread may need to evict data that belongs to a different threads to bring its own data into shared cache without considering whether the evicted data will be used by other threads, or not. Likewise, the benefits obtained through cache usage may differ among threads. Thus, allowing a thread to use more cache resources although it does not obtain much benefit from it, may prohibit the possible benefit that could be obtained by other threads. Such interference and evictions reduce the performance of multiple threads. If they are not coordinated appropriately, such evictions can be destructive for the overall system performance. There are various eviction and replacement strategies such as Least-Recently-Used (LRU) [9], LRU-based replacement methods [10], [11], sampling-based adaptive replacement (SBAR) [12]. In addition to replacement policies, there are various partitioning strategies such as way-partitioning [13] and cache partitioning [7], [11], [12].

It is difficult for operating system scheduler to ensure a faster progress for a high-priority thread on a chip multiprocessor, because the performance of a thread could be arbitrarily decreased by a high-miss-rate thread that is running concurrently with high-priority thread. Fedorova et al. [8] proposed an operating system scheduler to ensure performance isolation. In their proposal, threads end up with equal cache allocations, if threads that are running concurrently have similar cache miss rates. The shared cache is allocated based on demand; so, if the threads have similar demands they will have similar cache allocations.

Despite the abundance of replacement and partitioning strategies, they all come with certain limitations. For example, LRU cannot differentiate the requests from different threads. This causes LRU to blindly and unfairly evict the cache blocks to be used soon by another thread. When this thread tries to access the

cache blocks needed, it will end up with a cache miss. On the other hand, way-partitioning schemes differentiate threads and their requests. For this reason, the eviction decisions can be made properly without penalizing other threads blindly. However, way-partitioning schemes have limited scalability. If there are more threads than the number of cache ways, then the scheme would not work as intended.

To improve the cache access efficiency and system performance both replacement and scheduling strategies should be in place. LRU or way-partitioning schemes are orthogonal to the proposed cache-hierarchy-aware scheduling. Any replacement policy can be used along with cache-hierarchy-aware scheduler. It is beyond the scope of this work to tune replacement policy that would work best with the proposed cache-hierarchy-aware scheduler. Rather, we focus on the cache access characteristics of threads and try to come up with the best scheduling in which scheduled threads have the least interference and the number of evictions is minimized.

2.1.2 Cache-sharing-aware Scheduling

Cache-sharing-aware scheduling in operating systems can mitigate the cache contention among scheduled threads by assigning threads that can benefit from running on the same core by sharing data. Such cache-sharing-aware scheduling schemes can improve cache usage efficiency and program performance considerably in an environment where data sharing among threads is considerable. However, Zhang et al. [14] claimed that cache sharing has insignicant impact on performance of modern applications. This is due to the fact that there is very limited sharing of the same cache block among different threads in such applications. These applications are highly parallelized, where each thread is working on independent cache different block that are independent from each other. For this reason, it is very unlikely that they will access the same data block, so cache-sharing-aware schedulers have limited applicability.

Tam et al. [15] proposed a scheduling scheme to schedule threads based on data

sharing patterns that are detected online through hardware performance counters. The proposed scheme detects data sharing patterns and clusters threads based on the data sharing patterns. Then, the scheduler tries to map threads that belong to the same cluster onto the same processor, or as close as possible to reduce the number of remote cache accesses for shared data.

Settle et al. [16] developed a memory monitoring framework providing statistics in simultaneous multithreaded processors. Statistics regarding memory accesses of threads gathered from the proposed framework can be used to build a scheduler that minimizes capacity and conflict misses. For each thread, L2 cache accesses are monitored on a set basis to generate per-thread cache activity vectors. These vectors indicate the sets that are accessed most of the time. The intersection of these vectors specifies the sets that are likely to be conflicting. This information is used in scheduling decision.

2.1.3 Phase Prediction and Thread Classification

Sherwood et al. [17] introduced phase prediction method based on basic block vectors. Basic block vector represents the code blocks executed during a given interval of execution.

Chandra et al. [18] focused on L2 cache contention on dual-core chip multiprocessors. They proposed analytical model to predict number of L2 cache misses due to contention of threads on L2 cache.

Cazorla et al. [19] introduced a dynamic resource control mechanism and allocation policy in simultaneous multithreaded processors. The policy monitors the usage of resources by each thread and tries to allocate a fair amount of resources to each thread to avoid monopolization. It classifies threads into the groups based on cache access patterns as fast and slow. Then, it allocates the resources to these groups accordingly. Threads with pending L1 data misses are classified as member of the slow group and the ones without any pending L1 data misses are classified as member of the fast group. Another classification is made

as active and inactive based on the usage of certain resources. This classification allows borrowing resources from an inactive thread for the sake of another one that is active and looking for resources. Although they also used pending L1 data misses as a classification method, our approach differs in variety of ways. First, we use multiple L1 access characteristics such as number of accesses, miss ratio and number of evictions that provides better representation of execution characteristics of threads. Second, they do not rely on L1 access statistics for scheduling, instead they use it for clustering threads. Third, the main aim of the paper is not to develop a scheduler, but it is to develop a dynamic allocation policy for shared resources.

El-Moursy et al. [5] introduced a scheduling algorithm in which threads are assigned to processors based on the number of ready and in-flight instructions. The number of ready and in-flight instructions are strong indicators of different execution phases. The algorithm tries to schedule threads that are in compatible phases. They also used hardware performance counters to gather information required to assess the compatibility of thread phases.

Kihm et al. [20] proposed a memory monitoring framework that makes use of activity vectors that allow scheduler to estimate and predict cache utilization and inter-thread contention dynamically. However, they do not propose any scheduling algorithm that actually employs activity vectors.

2.1.4 Coscheduling

Tian et al. [21] proposed an A*-search-based algorithm to accelerate the search for optimal schedules. They formulated optimal co-scheduling as a tree-search problem and developed A*-based algorithm to find optimal schedule. The authors reduced constraints on finding optimal scheduling such that they allowed threads of different lengths. Further, they developed and evaluated two approximation algorithms, namely A*-cluster and local matching. A*-cluster algorithm is a derivative of A*-search-based algorithm that employs online adaptive clustering. It trades accuracy for scalability. The local-matching algorithm applies

graph theory to find the best schedule at a given time without any provision for the upcoming schedules. Although optimal scheduling algorithms are costly and inefficient for practical purposes, they can provide insights to enhance the practical scheduling algorithms and associated complexities with them.

Jiang et al. [22] proposed a reuse-distance based [23] locality model that provides proactive prediction of the performance of scheduled processes. The prediction is used in run-time scheduling decisions. They employed the proposed locality model in designing cache-contention-aware proactive scheduling that assigns processes to the cores according to the predicted cache-contention sensitivities. However, predictive model has to be constructed for each application through an offline profiling and learning process.

Snavely et al. [6] introduced a symbiotic scheduler, called SOS (Sample, Optimize, Symbios) simultaneous multithreaded processors. It identifies the characteristics of threads that are scheduled through sampling. SOS runs in two distinct phases: sample phase and symbiosis phase. It gathers information about threads running together in different schedule permutations during the sample phase. After this sample phase, SOS picks the schedule that is predicted to be optimal and proceeds to run this schedule in the symbiosis phase. The performance metrics of a schedule is gathered through hardware counters. SOS employs many predictors to identify the best schedule. One interesting result provided by Snavely et al. is that IPC alone is not a good predictor. It may happen that threads with higher IPCs monopolize system resources and can be detrimental to threads with lower IPCs. The limitation of this work is that it tries many schedules during sample phase to predict the best schedule to be executed in symbiosis phase. For workloads that are composed of many threads that exceed the available hardware resources, the sample phase would be much longer. In such a case, threads can change their characteristics that would not be reflected during the symbiosis phase. It is very limiting that symbiosis phase would be inaccurate due to the change of execution characteristics of threads during sample phase. Limited number of samples can be used to avoid longer sample phase; however, the probability of missing better schedules is increased in this case.

Suh et al. [24] proposed online memory monitoring scheme that uses hardware counters as well. The use of hardware counters provides estimates for isolated cache hits/misses with respect to the cache size. The estimation does not require to change cache configuration. This is achieved by employing single pass simulation method introduced by Sugumar and Abraham [25]. The provided estimation is used in designing memory-aware scheduling that schedules processes based on the cache capacity requirements. The marginal gains in cache hits for different sizes of cache for each process are monitored. Then a process that has low cache capacity requirement is scheduled with a process that has high cache capacity requirement to minimize the overall miss ratio.

DeVuyst et al. [26] proposed a scheduling policy for chip multiprocessors that allows unbalanced schedules (i.e., uneven distribution of threads among the available cores) if they provide higher performance and energy efficiency. The main challenge of allowing unbalanced schedules is to have an increased search space with a great extent.

2.2 Memory Access Scheduling

Rixner et al. proposed a First-Ready First-Come First-Serve memory access scheduler (FR-FCFS) [27] that prioritizes the requests that will be row-buffer hit. If there is no request that will be row-buffer hit, then the scheduler issues the oldest request first.

Mutlu and Moscibroda introduced a stall-time fair memory scheduler [28] that aims to balance the slowdown experienced by each thread. To do that, the scheduler gives priority to the requests of threads that are slowed down the most. In a similar effort [29], authors introduced a parallelism-aware batch scheduler that batches the requests based on their arrival time and their owners. The batch having the oldest request is given higher priority. In addition, the requests of a certain thread is serviced in parallel in different banks of ranks.

Kim et al. proposed a thread cluster memory scheduling [30] that divides

threads into two separate clusters and employs different scheduling policies for each cluster. The threads are clustered based on their memory access patterns. The first cluster consists of memory-non-intensive threads and the second cluster consists of memory-intensive threads. The scheduler prioritizes the requests of memory-non-intensive threads to improve throughput. Periodically, it prioritizes memory-intensive threads to provide fair access to the underlying memory subsystem.

Ipek et al. introduced a self-optimizing memory controller [31] that employs reinforcement learning to optimize the scheduling policy on the fly. They observed that the fixed schedulers are designed for average cases. For this reason, they can not perform well with dynamic workloads with changing memory access patterns. They employed machine learning techniques to make the memory controller capable of adapting and optimizing the scheduling policy based on the change in the memory access pattern of the workload.

There are a few studies that target reducing energy consumption of the memory subsystem. Hur and Lin proposed a power-aware memory scheduler [32] that is based on adaptive history-based scheduler. It uses the history of recent memory commands to select the memory command that will be issued next. The scheduling goals are represented as states in a finite state machine. Power saving is one of the goals along with minimizing latency and finding a balance between read and write requests.

Mukundan and Martinez proposed a self-optimizing memory scheduler [33] that targets to achieve different goals including reducing energy consumption. Their scheduler is based on reinforcement learning technique introduced by Ipek et al. [31]. They employed genetic algorithm to select the appropriate objective function automatically based on the current state of the system.

Ishii et al. proposed a memory access scheduler that employs phase prediction and thread prioritization. It predicts the execution phase of threads and prioritizes the threads in compute phase [34]. They also proposed a writeback-refresh overlap that refreshes one rank at a time and issues pending write commands of the ranks that are not refreshing. Instead of refreshing all ranks at the same

time, refreshing one rank at a time and issuing pending write commands of other ranks reduces the idle time of the memory bus and enhance the performance of the memory subsystem. The memory controller can issue the refresh commands similar to the Elastic Refresh, presented by Stuecheli et al. [35], such that it issues the refresh commands when the refresh quantum is exceeded, or when the read queue is empty.

Chapter 3

Adaptive Cache-Hierarchy-Aware Thread Scheduling

3.1 Introduction

Typical workloads running on chip multiprocessors are composed of multiple threads. These threads may exhibit different execution characteristics. In other words, they may run in different phases (e.g., memory phase, compute phase). Besides different threads, even a particular thread's execution characteristics may change over its life time. When threads are scheduled together that are running in phases that exacerbates contention for shared resources, the system performance decreases and throughput reduces due to conflicts. On the other hand, when threads running in cooperative phases are scheduled together, the contention for shared resources is diminished that yields to better resource utilization and higher throughput and improved system performance.

The choice of threads to be scheduled on the same core has signicant impact on overall system performance. Inter-thread contention occurs since coscheduled threads are competing for shared resources. The primary shared resource that influence the performance is the cache. An efficient scheduling should minimize the contention for shared caches to maximize utilization and system performance.

Since the execution characteristics of threads varies over time, the scheduling decision has to be remade based on provisioned behaviors of threads for the near future.

Other shared resources include functional units, instruction queues, memory, interconnections between resources, the translation look-aside buffer (TLB), renaming registers, and branch prediction tables. While threads share these resources to improve utilization, they also compete for these resources that may reduce efficiency. The utilization is enhanced when a thread uses a resource that would otherwise have gone unused. On the other hand, the efficiency of a shared resource may be reduced due to conflicting behaviors of threads. Shared cache is such a resource that is sensitive to interactions among threads. Our focus in this part of the thesis will be on scheduling of threads based on interactions on shared caches.

The way of making good use of shared caches is to understand underlying chip multiprocessor's cache architecture and to schedule multithreaded applications accordingly. For this reason, we briefly discuss chip multiprocessor architecture and underlying cache hierarchy. From the operating point of view, scheduling decisions have to be made based on the measures that affect the performance the most. Thus, we make a detailed survey on possible measures and evaluate their effects on performance. We observe that, contrary to the common thought, L1 cache access pattern of threads has a great impact on performance. To elaborate, we focused on L1 cache access patterns of threads and formulate a score for each thread that reflects execution characteristics of threads. The score of a thread specifies the intensity to compete for shared resources, or namely the *friendliness* of the thread. A thread that uses decent shared cache tends to be friendly, namely it causes less degradation to its co-runners, and it suffers less from its co-runners. Although the notion of friendliness is widely used in recent studies; we observed that they consider just a particular metric to determine friendliness, such as IPC of each thread or miss ratio. Such metrics are well indicators for particular cases; however, they become insufficient for general cases where great diversity is expected. Due to lack of adequate measure of friendliness, we developed a multi-metric scoring scheme to specify the execution characteristics of threads

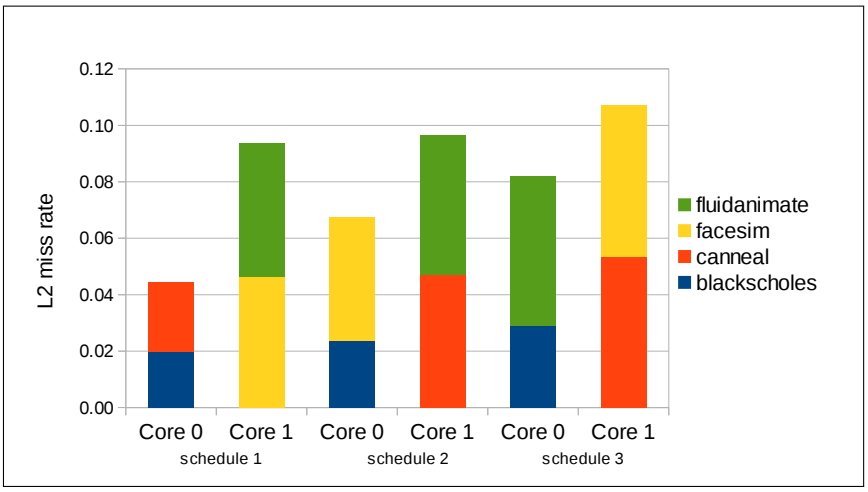

Figure 3.1: The L2 miss variation of four threads running on two cores under different scheduling schemes.

and make scheduling decisions on this multi-metric score.

3.2 Problem Statement

The conflicts among threads are difficult to predict due to their unrepeatable nature [36]. The behavior of a thread changes over time. For example, a thread may have high memory demands during the initialization and data loading, and following that it may have high CPU demand while processing loaded data. Loading and processing may occur several times that eventually changes behavior of a thread over time. Thus, static scheduling schemes are likely to fail on minimizing conflicts among threads.

An intuitive scheduling would be to group memory intensive threads with threads that are non-memory intensive. However, it is not always possible to find such pairs (e.g., all threads may be memory intensive in a particular time). Also, threads may be memory intensive; however, their memory access pattern may change drastically that affects the overall performance. For example, streaming threads may generate more memory requests; however, they do not get any benefit from cache hierarchy, since they have limited (or no) locality. Also, streaming

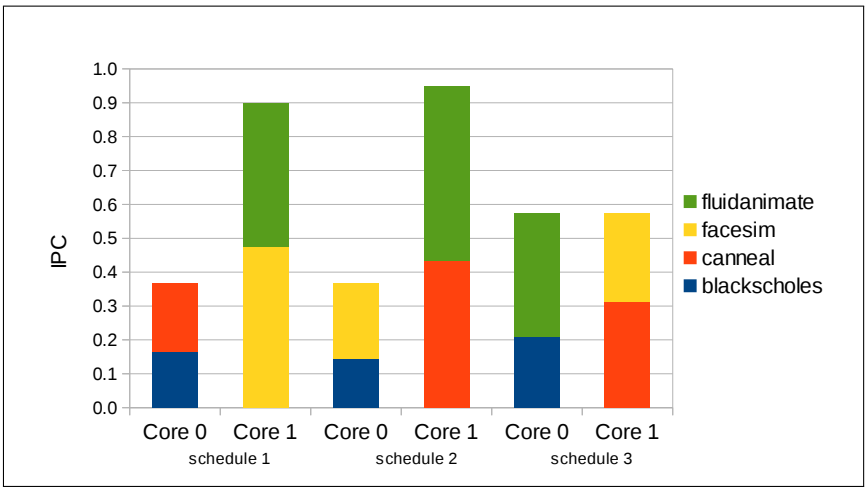

Figure 3.2: The IPC variation of four threads running on two cores under different scheduling schemes.

behavior of such threads are detrimental to other threads which are memory intensive. They evict the cache lines of other threads without gaining any benefit in return.

Figures 3.1 and 3.2 show the variances in L2 miss ratio and instruction per cycle (i.e., IPC) of threads under different scheduling, respectively. Four benchmarks are running on two cores where two threads share private L1 cache and all threads share unified L2 cache as LLC.

Existing schedulers used in operating systems are unaware of multi-level cache hierarchies and access/sharing pattern of threads running on chip multiprocessors. For this reason, traditional schedulers are oblivious to the access patterns of threads and they may schedule threads in a way that their memory accesses contradict with each other. This, in turn, hurts the cache performance leading to high miss ratios, high number of evictions and longer time to serve memory requests because data has to be brought from lower levels of memory hierarchy.

In addition, the use of profiling provided by compiler-directed approaches can not exploit the full potential of chip multiprocessors, since such profiling may not reflect the dynamically changing inputs and execution characteristics of threads.

Similarly, schedulers used in simultaneous multithreaded processors try to schedule threads based on each thread's expected resource utilization to maximize, but do not consider variances and changes in thread execution characteristics over time.

Scheduling of threads has to be made based on the measures that affect the performance the most. Thus, it is essential to figure out possible measures and evaluate their effects on performance. As a primary shared resource, caches and their performance measures are in focus. Most of the research is focused on the interaction between threads and their effects on the last-level cache (LLC). Currently, there is little understanding about their effects on higher levels of cache hierarchy that eventually affect the lower levels of cache hierarchy. For this reason, they pass over the primary source of demand for LLC accesses that are caused by cache misses on the upper levels of cache hierarchy. If the higher level cache accesses are scheduled wisely, the efficiency of shared resources can be improved and pressure on LLC can be reduced. The potential benefits of scheduling higher level cache accesses wisely are in two folds. First, higher level caches will be used efficiently and there will be less number of misses, thus latency will be reduced. Second, the probability of eviction of a block will be reduced due to the less number of misses in higher levels of the cache hierarchy. This reduces the penalty for lower level cache accesses.

Based on these observations, we conclude that, cache-hierarchy-aware scheduling for chip multiprocessors, which adopts dynamically changing execution characteristics of threads, is inevitable.

3.3 Motivation

Numerous research efforts have been made on minimizing cache conflicts and capacity misses of shared LLC (in most cases L2) for both multiprocessors and chip multiprocessors. Although such efforts are effective (i.e., minimizing cache conflicts and capacity misses of LLC), they ignore the effects of higher levels of

cache hierarchy on eventual LLC performance. Typically, each core has L1 that is shared by multiple threads in chip multiprocessors. Being oblivious to L1 cache conflicts and misses eventually creates more pressure on lower level caches (e.g., L2) and results in high latency lower level cache accesses.

The fundamental motivation behind focusing on LLC in previous research efforts is that a miss on LLC requires high latency main memory access. Although this is a valid argument, it does not justify to underestimate the effect of L1 (or any cache level above LLC) on memory access latency. Contrary, we claim that L1 cache access pattern (i.e., number of accesses, misses, evictions, etc.) has great impact on overall memory access performance. This is our main motivation to build cache-hierarchy-aware scheduler.

In recent study, Zhang et al. [14] pointed out that there is very limited sharing of the same cache block among different threads. Modern applications are highly parallelized, where each thread is working on independent cache block. For this reason, it is very unlikely that they will access the same data block, so there is limited or no data sharing. For example, threads of data-parallel programs may process different sections of data. Similarly, threads of pipeline programs may execute different tasks that may not use the same data set. In both cases, there is no concern of shared data among multiple threads; however, the way the threads use shared cache has an influence on performance. This observation is important, since programs show different characteristics in different phases of the execution so that no particular mapping work well for all the phases. With this motivation, we propose *adaptive* cache-hierarchy-aware scheduler. More specifically, adaptive cache-aware-hierarchy scheduling aims to adopt changing execution characteristics of threads and tries to find best scheduling that improves the performance by reducing the cache contention and conflicts among coscheduled threads.

3.4 Contributions

In this part of the thesis, we present a detailed study to show the importance of cache-hierarchy-aware scheduling for applications running on chip multiprocessors. We investigate the impact of scheduling threads with different execution characteristics and observe that the best scheduling for a given thread varies depending on other threads that are scheduled along with it.

We introduce a fine-grained, multi-metric scoring scheme to classify threads with respect to their execution characteristics. We use this fine-grained, multi-metric scoring scheme to predict threads that get along with each other and schedule them on the same core. The metrics used in scoring scheme are gathered from L1 cache, as opposed to LLC as in the most of the previous works.

We propose a novel cache-hierarchy-aware scheduler that schedules threads in a way that it minimizes the number of accesses to the lower level of cache/memory hierarchy and reduces the number of evictions required on shared caches that eventually limits the interference. Such a strategy leads to higher system throughput and improved performance.

The proposed cache-hierarchy-aware scheduler is adaptive, such that it takes dynamically changing execution characteristics of threads into account. We observe that by employing our adaptive cache-hierarchy-aware scheduling, the performance (i.e., instruction per cycle) of the benchmarks used in this work are improved by up to 12.6% and an average of 7.3% over the static schedules. The improvements are due to reduced interference among coscheduled threads, leading to reduced number of evictions/misses and balanced number of accesses that minimizes capacity conflicts.

3.5 Chip Multiprocessors of Simultaneous Multithreading

Chip multiprocessors [2] and simultaneous multithreading [3, 37, 38] are two approaches that have been proposed to increase processor efficiency. Chip multiprocessors have multiple cores that share a number of caches and buses. Figure 3.3 show typical chip multiprocessor architecture and its memory subsystem. It has four cores and each core can host two threads concurrently.

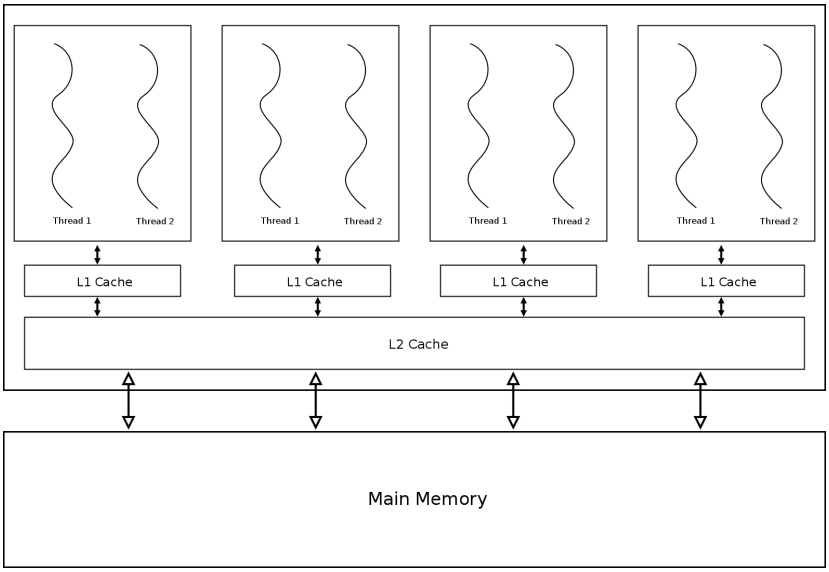

Figure 3.3: Typical chip multiprocessor architecture and its memory subsystem.

While each core has private L1 cache, all cores share a common on-chip L2 cache. Typically, L1 caches have a latency of 1 to 4 cycles, while L2 caches have a latency of 10 to 20 cycles. On the other hand, off-chip main memories have a latency of 100 to 200 cycles. Since off-chip memory access is extremely expensive in terms of cycles and power, it is essential to utilize provided on-chip cache hierarchy and minimize the number of off-chip memory accesses.

3.6 Inter-thread Contention and Slowdown

When there are multiple threads running on chip multiprocessors concurrently, there will be interference among all the threads. Threads running on the same core compete for L1 cache, while they compete with all other threads running on chip multiprocessor for shared L2 cache. To assess the interference among threads and make a good scheduling evaluation, it is necessary to formulate the slowdown of a thread when running along with other threads.

To avoid the distractions from other complexities, such as difference between program execution times and context switches in operating system, we consider the following simplified scenario. There are N threads of the same number of instructions to be executed. The average slowdown of all threads can be calculated as geometric mean of slowdowns of threads. The scheduler that minimizes the average slowdown as given in Expression 3.1 is more desirable.

$$min \sqrt[N]{\prod_i^N \frac{IPC(i)_{stand\_alone}}{IPC(i)_{coscheduled}}} \tag{3.1}$$

There is a trade-off between minimizing the average slowdown and maximizing the overall system performance (i.e., IPC). It is possible to have schedules that have lower average slowdown, but they also result in lower performance. On the other hand, it is possible to have schedules that provide higher performance, but they also have higher average slowdown. Therefore, a good scheduler should find a balance between slowdown and performance.

3.7 Performance Counters and Monitoring

The chip multiprocessors have performance monitoring units (PMUs) with integrated hardware performance counters. The statistics that are needed to operate proposed scheduling algorithm can be collected through PMUs. PMUs

can provide fine-grained statistics with relatively low overhead [15]. Parekh et al. [39] used hardware performance counters that provide cache miss and related information to schedule threads wisely in simultaneous multithreaded processors. Similarly, Bulpin and Pratt [40] used performance counters to develop symbiotic coscheduling approach on simultaneous multithreaded processors.

For our proposed cache-hierarchy-aware scheduler, we focus on L1 cache access pattern of threads and classify them based on their propensity to compete for L1 cache and their relative effectiveness of L1 cache usage. Although classification of threads is widely adopted in scheduling research, we observed that they consider a particular metric to classify threads, such as IPC of each thread and miss rate. Such metrics are well indicators for certain cases; however, they do not work well for other cases. Thus, there is no silver-bullet metric that provides the best for all cases. With this in mind, we developed a multi-metric scoring scheme to specify execution characteristics of threads. Then, the score obtained through multi-metric scoring scheme is used to make scheduling decisions.

3.8 Phase Detection and Prediction

The prediction of thread's cache access behavior for the next interval is essential to obtain desired performance. Simply, predicting thread's cache access behavior for the next interval will be the same as the previous interval provides reasonable accuracy (e.g., between 84% and 95% [16]). Although the accuracy of the prediction can be increased by using more complex prediction methods, we believe that using last interval behavior as a prediction model for the next interval is sufficient for our purpose. It is a fair trade-off to have decent prediction accuracy with less complexity, compared to marginal gain in accuracy with high complexity.

3.9 Multi-metric Scoring Scheme

The behavior of a thread can be generalized by expressing three attributes for a given interval. These attributes are aggressiveness, density and inefficacy. These attributes are represented in a binary vector, called *attribute vector*. The illustration of attribute vector is given in Figure 3.4.

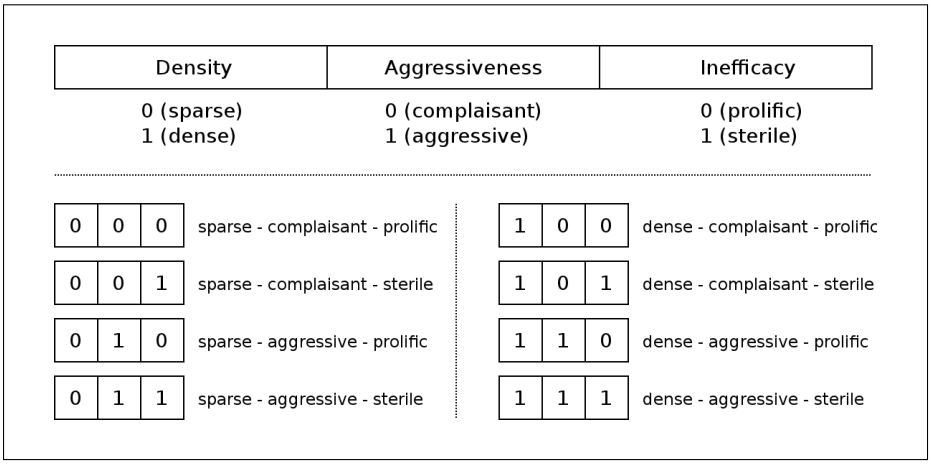

Figure 3.4: Attribute vector expresses execution characteristics of a thread.

Each attribute corresponds to different characteristics of a thread. These characteristics have impact on the overall performance, eventually. The description of attributes are as follows.

Aggressiveness determines the degree of acrimony of a thread, specifying how much a thread interfere with other threads running concurrently on the same core. Aggressiveness of a thread is related to its propensity of evicting cache blocks of other threads. A thread that has higher eviction rate is considered as aggressive, while the one with lower eviction rate is considered as complaisant.

Density determines the relative intensity of cache accesses of a thread with respect to the sum of cache accesses of all threads. If a thread has higher number of cache accesses, then it is considered as dense. On the other hand, it is considered as sparse if a thread has lower number of cache accesses relative to the number of overall cache accesses made during the given interval.

Inefficacy determines the degree of efforts of a thread that goes unrewarded.

If a thread has high cache miss ratio, then it is considered as sterile. On the other hand, it is considered as prolific if a thread has high cache hit ratio.

Although the attributes are related, they are considered as orthogonal to each other. Note that, a thread may be sterile, but not aggressive if its misses do not cause evictions.

These attributes are represented as bits in the attribute vector. The following formulas are used to determine whether a thread has certain attribute or not.

$$Aggressiveness(T_i) = \begin{cases} 1 & if \quad \frac{number\ of\ L1\ evictions(T_i)}{number\ of\ L1\ accesses(T_i)} \geq \tau_a, \\ 0 & else \end{cases}$$

$$Density(T_i) = \begin{cases} 1 & if \quad \frac{number\ of\ L1\ accesses(T_i)}{\sum_{j=0}^{N} L1\ accesses(T_j)} \geq \tau_d, \\ 0 & else \end{cases}$$

$$Inefficacy(T_i) = \begin{cases} 1 & if \quad \frac{number\ of\ L1\ misses(T_i)}{number\ of\ L1\ accesses(T_i)} \geq \tau_i, \\ 0 & else \end{cases}$$

where τ_a, τ_d and τ_i are thresholds for aggressiveness, density and inefficacy, respectively. They are determined empirically. N is the number of threads running on the chip multiprocessor.

Each attribute vector corresponds to a decimal value that specifies a multi-metric score of a thread. This value is calculated as:

$$Score = \sum_{i=0}^{2} 2^i \times AV_i$$

where AV_i represents i^{th} bit of *attribute vector* of a thread (AV_i represents the least significant bit when i = 0, and AV_i represents the most significant bit when i = 2).

3.9.1 Scalability of Mutli-metric Scoring Scheme

In case of having large number of threads running on chip multiprocessors with extensive number of cores, the 3-bit attribute vector and scoring scheme may not differentiate execution characteristics of threads in a desired resolution. This may yield to have coarse-grained schedules. To have higher resolution of execution characteristics of threads with fine-grained schedules, it is better to expand the attribute vector. For each attribute, more bits can be used to specify the attribute in higher resolution. An example of attribute vector with higher resolution is shown in Figure 3.5.

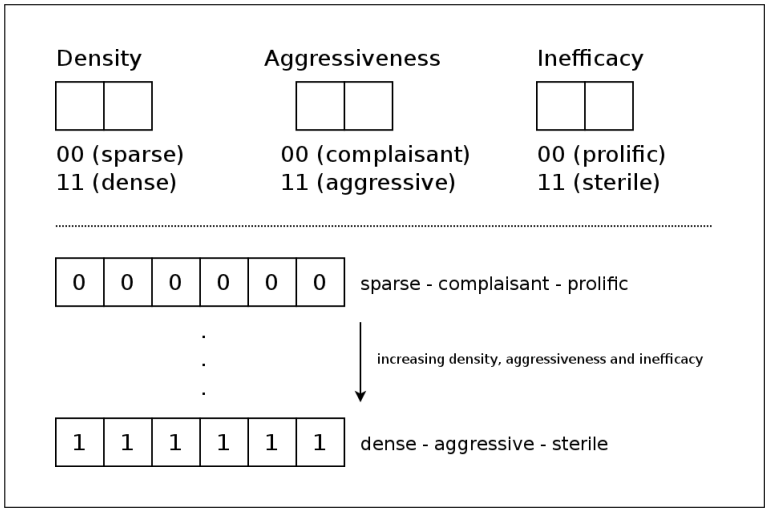

Figure 3.5: Attribute vector that expresses execution characteristics of a thread in higher resolution.

3.10 Adaptive Thread Scheduling

Figure 3.6 illustrates the flow of adaptive cache-hierarchy-aware thread scheduling in chip multiprocessors.

After collecting information regarding L1 cache performance and updating attribute vectors of threads, the scheduling decision can be made. The scheduling decision is made based on the multi-metric scores of threads.

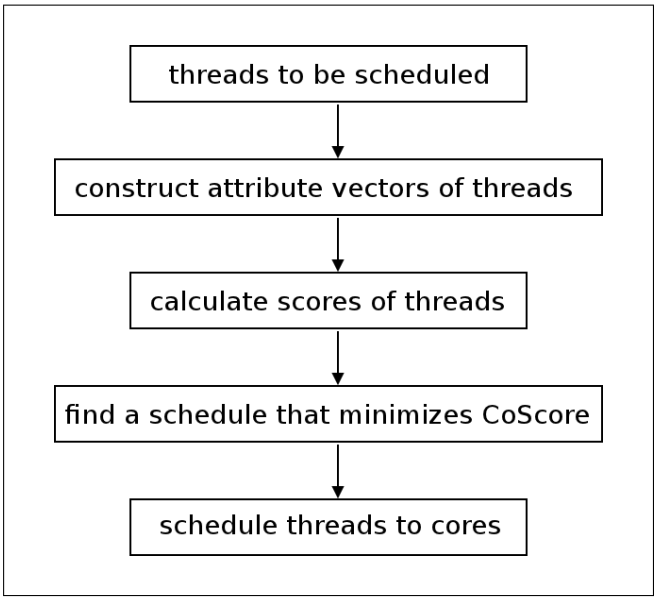

Figure 3.6: The flow of adaptive cache-hierarchy-aware thread scheduling.

Each candidate schedule has a score expressed as coscheduling score, in short *CoScore*. The aim of scheduling is to find a schedule that minimizes CoScore calculated as:

$$CoScore(T_i, T_j) = AV(T_i) \quad \& \quad AV(T_j) \tag{3.2}$$

where T_i and T_j are candidate threads to be coscheduled (i.e., $T_i \neq T_j$); and $AV(T_i)$ and $AV(T_j)$ are attribute vectors of T_i and T_j, respectively. Note that T_i and T_j are candidate threads, so they are not scheduled, yet.

The CoScore is simply a logical bitwise AND operation between multi-metric scores of candidate threads. It is simple, yet an effective way to find desired schedules that improve the performance. This, simple AND operation favors scheduling threads can get along with each other. Figure 3.7 shows the illustration of calculating CoScore and finding the schedule that minimizes CoScore.

CoScore's most significant bits are dominant in selecting the schedule. Our approach tends to prefer a CoScore 011 over 100. Hence, metrics can be prioritized according to their positions in the CoScore.

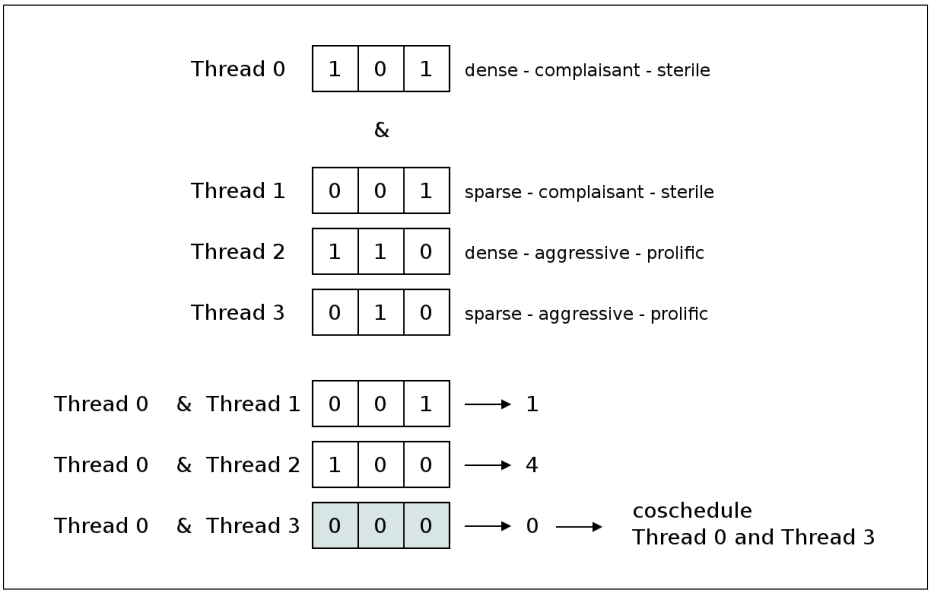

Figure 3.7: Coscheduling score calculation.

A schedule that has the lowest CoScore is selected as candidate. If there are more candidates, then the one that preserves locality is selected (i.e., no thread migration is required).

The calculation of CoScore starts with a thread that has the lowest multi-metric score. Then, a thread that will minimize the scheduling score when scheduled with the current thread is found. If there are multiple threads with lowest multi-metric score, a preference is given to the thread that has the higher IPC. If still there are multiple threads that have the same multi-metric score and the same IPC, then a thread is chosen randomly.

When all threads are scheduled to appropriate cores, the performance counters are reset. With the new scheduling period, attribute vectors of threads are reconstructed, multi-metric scores of threads are reevaluated and scheduling is re-executed as discussed. The details of this adaptive cache-hierarchy-aware thread scheduling algorithm is given in Algorithm 1.

When a thread is scheduled to execute on a different core than it was running on before, the cache blocks required by this thread have to be reloaded from L2 or lower level of cache hierarchy. While this comes with an overhead, we determined

Algorithm 1: Cache-hierarchy-aware thread scheduling algorithm.

```
UnSched_T → unscheduled threads;
UnMapped_T → matched but not mapped threads;
S, C → set of threads to be scheduled;
T_s, T_c → threads to be scheduled;
P_s, P_c → cores on which T_s and T_c run during the last interval, respectively;

while UnSched_T ≠ empty do
    T_s ← a thread that has the lowest score;
    if S has multiple threads then          /* with the lowest score */
        T_s ← select thread ∈ S that has the highest IPC;
        if S has multiple threads then      /* with the highest IPC */
            T_s ← select a thread ∈ S randomly;
        end
    end

    C ← a thread ∈ UnSched_T that minimizes CoScore;
    if C has multiple threads then      /* with the lowest CoScore */
        T_c ← select thread ∈ C that run on P_s recently;
    else
        T_c ← select a thread ∈ C with the highest IPC;
        if C has multiple threads then      /* with the highest IPC */
            T_c ← select a thread ∈ C randomly;
        end
    end

    if P_s is available then
        map T_s and T_c to P_s;
    end
    else if P_c is available then
        map T_s and T_c to P_c;
    end
    else
        UnMapped_T ← T_s and T_c
    end
end

while UnMapped_T ≠ empty do
    T_s and T_c ← select matched threads from UnMapped_T;
    map T_s and T_c to the available(free) cores;
end
```

that it is amortized over long execution intervals. This is due to the fact that the number of cache misses will be reduced as a result of reduced interventions in the scheduled thread.

Figure 3.8 illustrates that how cache-hierarchy-unaware scheduling can penalize the threads that could perform much better. Notice the lower L1 hit ratio and higher L2 miss ratio. The number of L2 hits is four, while the number of L2 misses is 17.

Figure 3.9 illustrates that how cache-hierarchy-aware scheduling actually reduces the number of L2 misses and increases the L1 hit ratio. The same example is used with Figure 3.8. The pressure due to the L1 misses is reduced that results in less number of L2 accesses and L2 misses. While the number of L1 hits is increased from 0 to 4, the number of L2 misses reduced from 17 to 14. Since this is just an illustration, we do not consider the effects of L1 hits on core 0. In reality, core 0 is most likely generate more memory requests compared to core 1, since core 0 can continue issuing instructions in a higher rate due to higher L1 hit ratio.

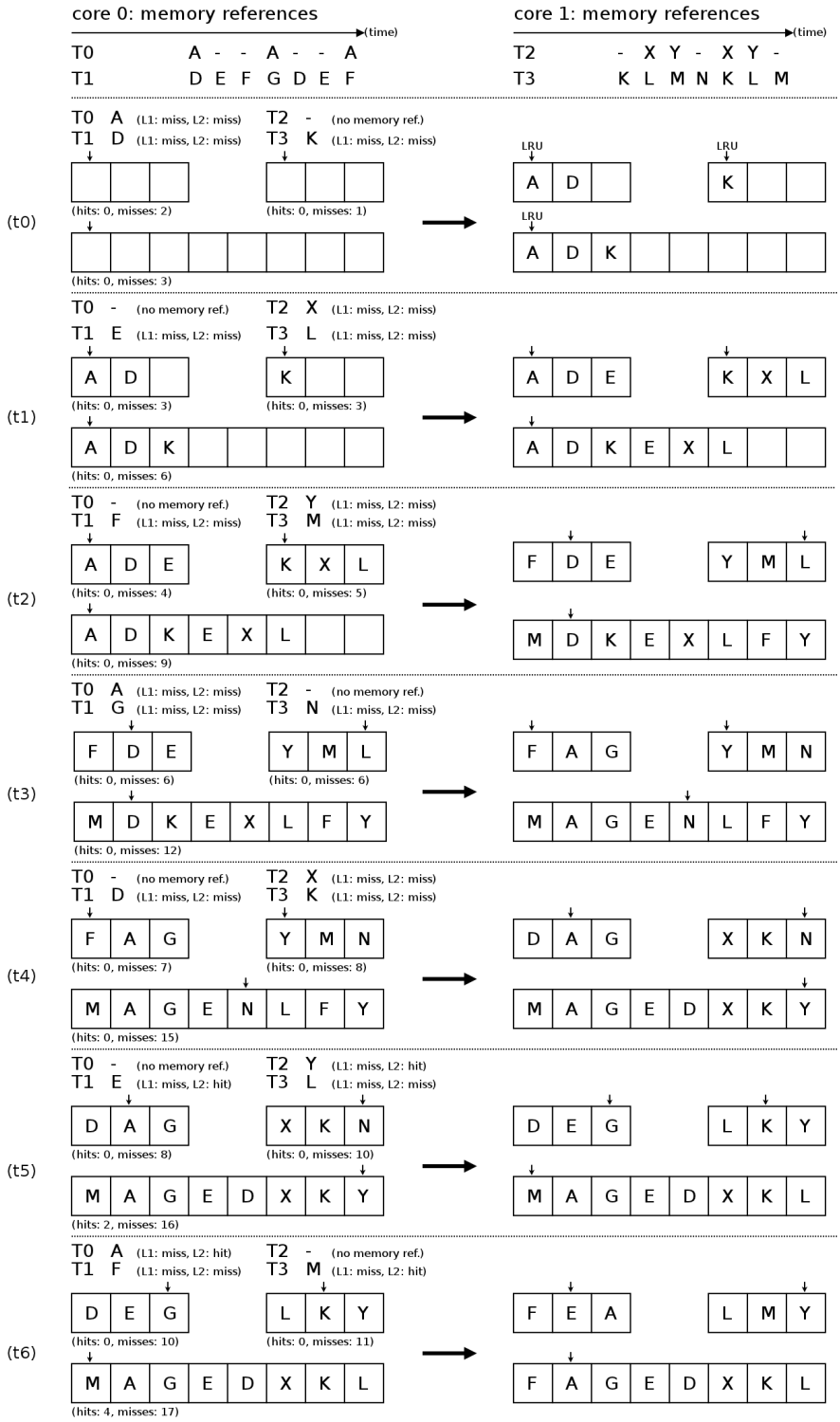

Figure 3.8: Cache-hierarchy-unaware scheduling. The number of L2 hits is four, while the number of L2 misses is 17.

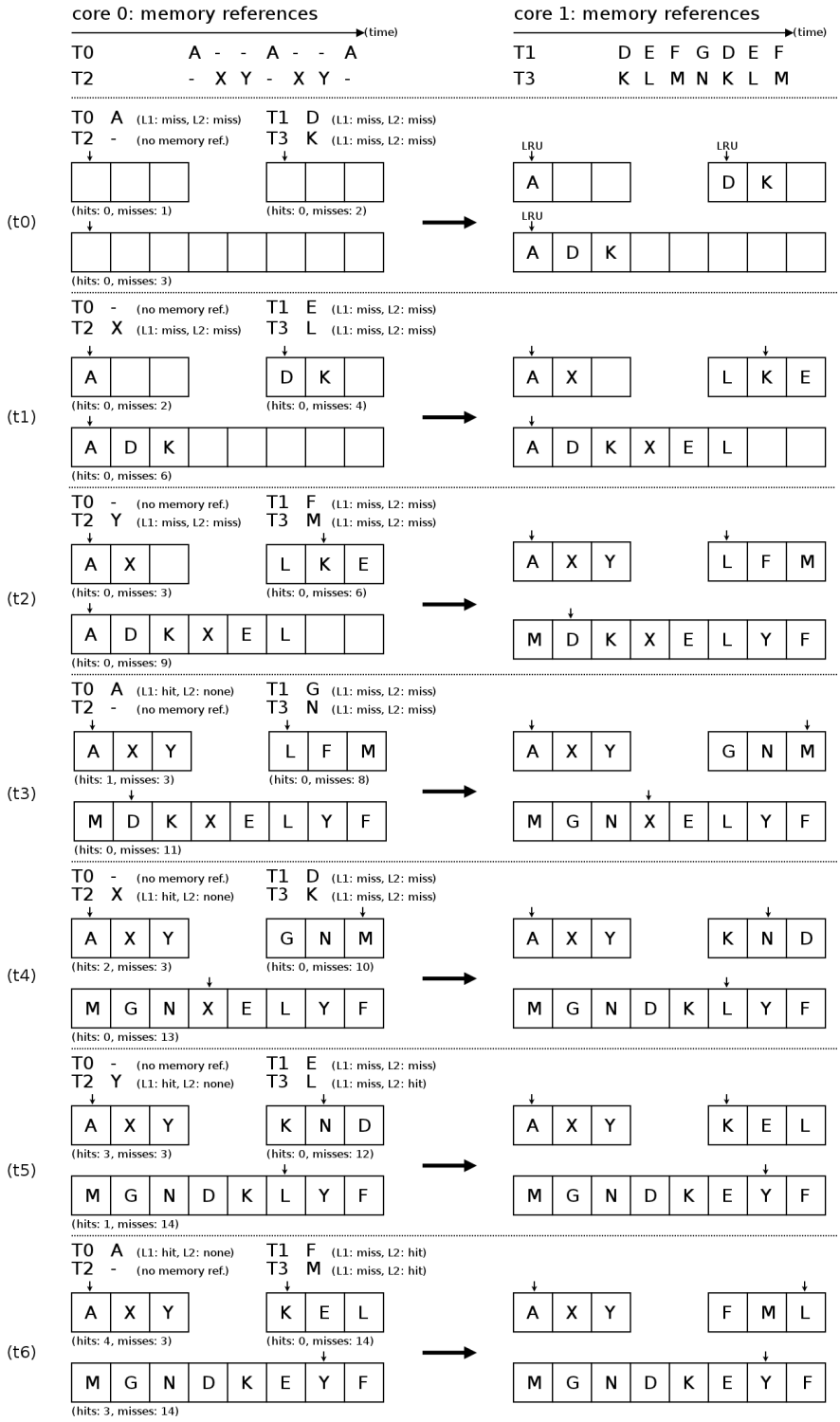

Figure 3.9: Cache-hierarchy-aware scheduling. The number of L2 misses reduced to 14.

Chapter 4

Adaptive Compute-phase Prediction and Thread Prioritization

4.1 Introduction

In response to increased pressure on memory subsystem due to the memory requests generated by multiple threads, an efficient memory access scheduler has to fulfill the following goals:

- serve memory requests in a way that cores are kept as busy as possible
- organize the requests in a way that the memory bus idle-time is reduced

One approach to keep cores as busy as possible is to categorize and prioritize threads based on their memory requirements. Threads can be categorized into two groups: memory-non-intensive (i.e., threads in compute phase), and memory-intensive (i.e., threads in memory phase). Kim et al. [30] proposed a memory access scheduler that gives higher priority to memory-non-intensive threads, and gives lower priority to memory-intensive threads. The reason behind such prioritization is that memory-non-intensive threads (i.e., threads in compute phase)

can make fast progress in their executions, so the cores can be kept busy. On the other hand, memory-intensive threads (i.e., threads in memory phase) have more memory operations and they do not use computing resources as often.

Ishii et al. followed the same idea of prioritizing the threads based on their memory access requirements. They enhanced prioritization mechanism with a fine-grained priority prediction method. This fine-grained priority prediction method is based on saturation counters [34]. They do not rely on time quantum (typically some millions of cycles) to categorize threads as memory-non-intensive and memory-intensive, instead they employ saturation counters to categorize threads on the fly. In addition, they proposed writeback-refresh overlap that reduces memory bus idle-time. Writeback-refresh issues pending write commands of the ranks that are not refreshing and refreshes a given rank concurrently. This means that the issuing write commands (of rank that is not refreshing) and refreshing a rank are overlapped. This reduces the idle time of the memory bus and enhances the performance of the memory subsystem.

4.2 Problem Statement

The necessity of distinguishing threads based on their memory access requirements is well understood and many research efforts have exploited this fact. Kim et al. [30] and Ishii et al. [34] provided examples of thread classification and prioritization mechanisms. They categorize threads into two groups, namely memory-non-intensive (i.e., threads in compute phase), and memory-intensive (i.e., threads in memory phase). Although they distinguish threads into different groups, they do not differentiate the threads in the same group. We believe that fine-grained prioritization is required even for the threads in the same group (i.e., memory-non-intensive or memory-intensive) to maximize the overall system performance and utilize the memory subsystem at the highest degree. For this reason, we introduce a fine-grained thread prioritization scheme that can be employed by existing state-of-the-art memory access schedulers.

In addition to that, the thread classification scheme presented in the work of Ishii et al. [34] is based on saturation counters. Saturation counters provide effective metrics to understand threads to be in compute phase or in memory phase. In determination of this, interval and distance thresholds are used. These thresholds are predefined and determined empirically. Although they are effective, they are vulnerable to short distortions and bursts that may result in wrong classification of threads. We believe that these thresholds have to be updated appropriately depending on the execution characteristics of the threads to classify them with higher accuracy. For this reason, we enhanced phase prediction scheme of Ishii et al. and make it adaptive.

4.3 Motivation

The classification of threads running on chip multiprocessors is essential to improve memory subsystem performance. Since the execution characteristics of the threads may change during their lifetime, such a classification has to be updated accordingly. Mainly, a thread can be either in compute phase, or memory phase for a given time of its execution. For this reason, the detection of a phase that a thread is currently in and prediction of the phase that a thread is going to be in have significant importance in scheduling memory accesses. There has to be a memory access scheduler that can predict the execution phases of threads efficiently, and prioritize them to access memory. The prioritization has to be fine-grained and the phase detection has to be accurate, thereby motivating us to implement fine-grained prioritization and adaptive phase prediction in memory access scheduler.

4.4 Contributions

In this part of the thesis, we introduce a memory access scheduling algorithm that is an enhancement to the state-of-the-art memory access scheduler presented by

Ishii et al. [34]. We propose a fine-grained thread prioritization scheme that is performed in two steps. In the first step, threads are categorized as memory-non-intensive and memory-intensive. Threads that are memory-non-intensive are given higher priorities than the threads that are memory-intensive. In the second step, threads that are in the same group are prioritized among themselves based on how much progress they can make in their executions. We call this approach as *adaptive thread prioritization*.

In addition to that, we enhanced compute-phase prediction scheme presented by Ishii et al. in a way that it detects phase changes in a timely manner. Ishii et al. used predefined thresholds for saturation counters to predict execution phases; however, predefined thresholds are inadequate to detect phase changes in a timely manner. Inadequately defined thresholds may result in certain threads to be prioritized unfairly longer while preventing others to be prioritized when they actually should be prioritized. Thus, the efficiency of phase prediction mechanism is correlated to the accuracy of thresholds used for saturation counters. We introduced a mechanism that determines the thresholds for each thread on the fly, considering the recent memory access pattern of a thread. Since the thresholds are determined at run-time, we call it *adaptive compute-phase prediction*.

Compared to the prior schedulers First-Ready First-Come First-Serve (FR-FCFS) and Compute-phase Prediction with Writeback-Refresh Overlap (CP-WO), our algorithm reduces the execution time of the generated workloads up to 23.6% and 12.9%, respectively.

4.5 Memory Model

The memory model used in our study is based on the architecture specified in USIMM simulation framework [41]. In the USIMM simulation framework, DRAM is separated into channels and each channel consists of ranks. Each rank has multiple banks. The write and read requests are queued in separate queues, namely write queue and read queue, respectively. The size of write queue is

64 for one-channel configuration, and 96 for four-channel configuration. On the other hand the size of read queue is considered to be infinite.

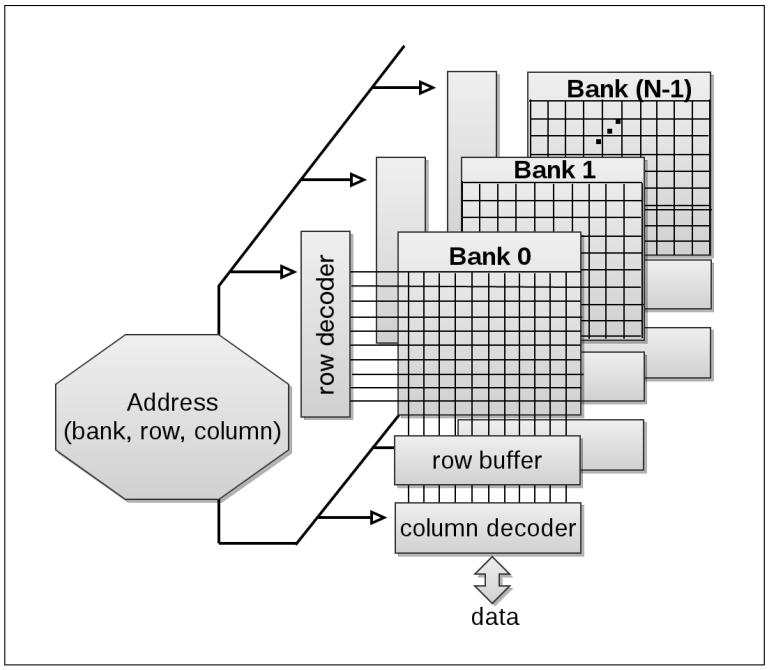

Figure 4.1: Typical memory bank architecture found in a DRAM rank.

Each rank of a DRAM has multiple banks. Typical memory bank architecture found in a DRAM rank is shown in Figure 4.1. A bank is a two-dimensional structure composed of rows and columns. Each bank operates in lockstep fashion. A row of a bank is accessed as a whole at a given time. When a row is accessed through *activate row* command, the entire row is brought into the row-buffer of that bank. The row-buffer allows to reduce the number of cycles needed to serve the subsequent requests that access to the same row. When a row is present in the row-buffer, a set of read/write requests to this row can be performed by executing *column read/write* command only. This reduces the total number of cycles needed to complete the requests made for that bank. When the *column read/write* commands are finished, the row-buffer is precharged that restores the content of the row-buffer into the corresponding row of a bank.

4.6 Adaptive Compute-phase Prediction

Ishii et al. [34] proposed a memory access scheduling algorithm that we call Compute Phase Prediction with Writeback-Refresh Overlap (CP-WO). Compute Phase Prediction with Writeback-Refresh Overlap scheduler can predict the execution phase of a thread on the fly. Typically there are two phases a thread may be in. A thread may be either in compute phase, or in memory phase. The threads in compute phase are memory-non-intensive. On the other hand, the threads in memory phase are memory-intensive. The threads in compute phase are given higher priorities for memory accesses. The reason behind this is that the threads in compute phase can make fast progresses and keep cores busy, thereby improving the performance and enhancing the utilization. On the other hand, the threads in memory phase spend more time on memory and have less computation, thus leave cores idle. For this reason, the threads in compute phase are prioritized.

The idea of prioritizing the threads in compute phase (i.e., memory-non-intensive) is also used by Kim et al. [30] in their thread cluster memory (TCM) scheduler. The thread cluster memory scheduler classifies threads into two groups, namely, memory-non-intensive threads and memory-intensive threads. It prioritizes the memory access requests of memory-non-intensive threads over memory-intensive threads. The prioritized memory-non-intensive threads will spend less amount of time on memory operations and return back to the execution much earlier. This way, cores in a chip multiprocessor can be kept as busy as possible, increasing the throughput and improving the performance.

The thread cluster memory scheduler classifies threads whenever the time quantum exceeds a certain threshold, typically in the range of million cycles. Threads are classified at the beginning of each quantum based on memory access patterns. Due to the dynamic behavior of threads, their execution characteristics (i.e., memory access pattern) may change before the time quantum expires. If this is the case, threads have to be re-clustered to properly prioritize them. However, the thread cluster memory scheduler does not have capability of re-clustering threads before the time quantum expires. Threads are treated as they

started, although they may change their execution phase, which in turn, requires adjustments in priorities of threads. For this reason, the thread cluster memory scheduler can not respond to the changes in memory access patterns of threads in a timely manner. Due to this limitation, it blindly misses possible improvements on performance and fairness.

To overcome the barrier in thread cluster memory scheduler, Ishii et al. employed saturation counters to classify threads. Saturation counters help to respond to changes in memory access pattern of a thread in a faster manner compared to time quantum approach of thread cluster memory scheduler. Another difference between thread cluster memory scheduler and the scheduler of Ishii et al. is that the former uses memory traffic generated by L2 cache miss to cluster threads, while the latter uses the committed number of instructions to cluster threads. We believe that the former provides better indication of threads being in compute phase, or in memory phase.

Ishii et al. used saturation counters to determine if a thread is in compute phase or in memory phase. These saturation counters are *interval counter* and *distance counter*. The interval counter specifies the number of committed instructions between the last two cache misses for a thread. If an interval counter (i.e., δ_i) of a thread exceeds the interval threshold (i.e., τ_i), then thread is predicted to be in compute phase and the distance counter (i.e., δ_d) is set to zero. On the other hand, the distance counter is incremented if the interval counter stays below the interval threshold. If there are consecutive accesses whose interval counter stays below the interval threshold that leads distance counter to exceed the distance threshold (i.e., τ_d), then a thread is considered to be in memory phase. The distance threshold determines how long a thread is going to be treated as it is in compute phase; although, it does not satisfy the interval counter constraint. This allows tolerating short distortions and small bursts that may be seen in compute phase and thereby, not treating a thread to be in memory phase, immediately. However, it is important to decide how long to tolerate a thread that does not satisfy the interval counter constraint before considering it to be in memory phase and vice versa.

The distance threshold (τ_d) and the interval threshold (τ_i) are predefined in the original work. The higher distance threshold becomes inappropriate for most of the cases since it keeps a thread in compute phase longer; although the thread actually is in memory phase. On the other hand, smaller distance threshold makes a thread vulnerable to short distortions and bursts, so a thread is treated as it is in memory phase; although, it is in compute phase. To deal with such anomalies, we introduced an *adaptive compute-phase prediction* scheme. Adaptive compute-phase prediction allows us to determine the distance threshold on the fly by monitoring memory access characteristics of a thread. The distance threshold determined adaptively tolerates short distortions and bursts that can be seen in compute phase, as it is in the original work. More importantly, adaptively determined distance threshold helps to predict the execution phase changes earlier compared to predefined distance threshold. The illustration of original compute-phase prediction is given in Figure 4.2. Similarly, the illustration of adaptive compute-phase prediction is given in Figure 4.3.

In Figure 4.2, the predefined distance threshold (τ_d) is set to five. Light boxes indicate that interval counter constraint is satisfied (i.e δ_i exceeds τ_i). Dark boxes indicate that interval counter constraint is not satisfied (i.e δ_i stays below τ_i). The distance counter (δ_d) is incremented if interval counter constraint is not satisfied and reset otherwise. When the distance counter (δ_d) exceeds the distance threshold (τ_d is five in this illustration), the thread is considered to be in memory phase. The thread is considered to be in compute phase when it satisfies interval counter constraint again.

On the other hand, in Figure 4.3, our adaptive compute phase prediction scheme observes that the interval counter constraint is satisfied, except consecutive two cache misses. By using this observation, our adaptive compute phase prediction determines that there is no need to consider a thread in compute phase if the distance counter (δ_d) exceeds two. Whenever the third consecutive access that does not satisfy interval counter constraint is occurred, adaptive compute-phase prediction concludes that a thread exits compute phase and goes into memory phase. Note that, adaptive compute-phase prediction can detect

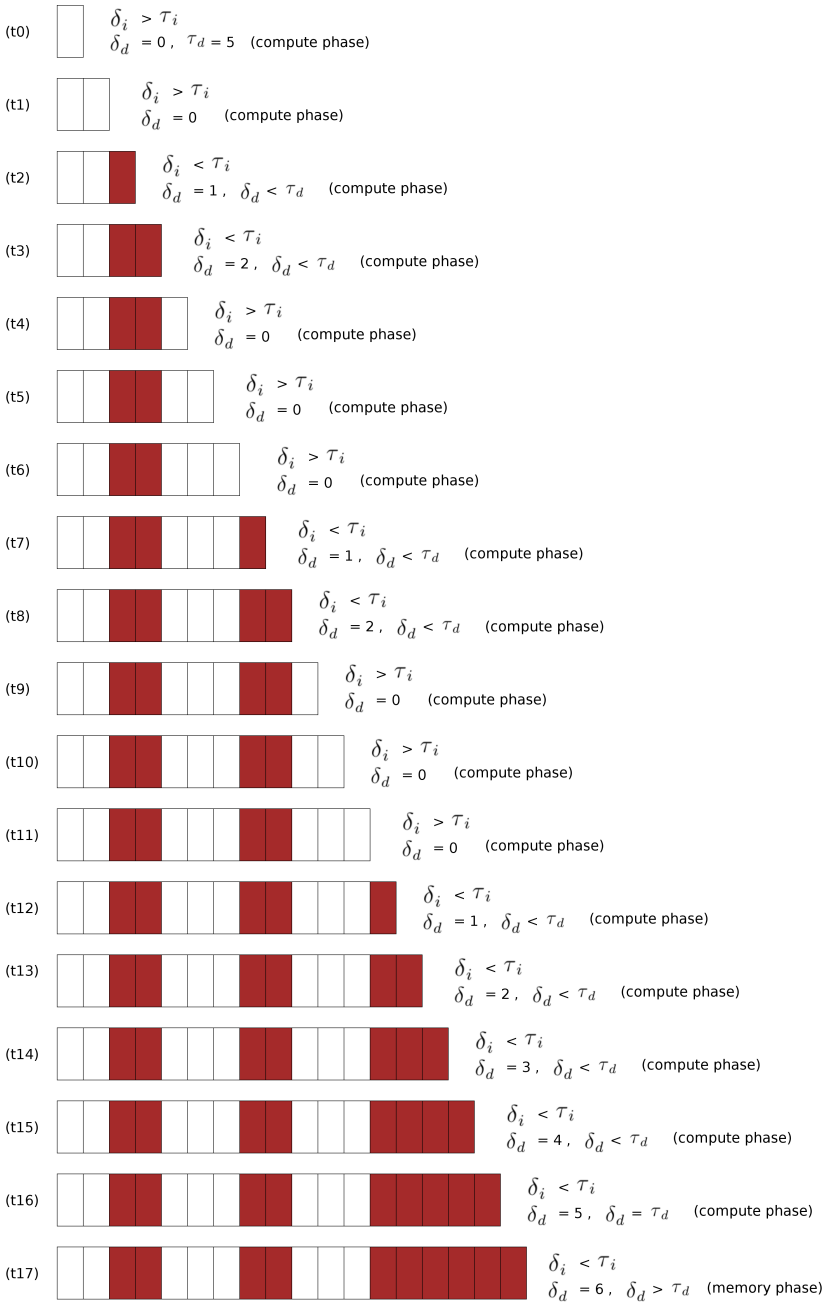

Figure 4.2: Default compute-phase prediction.

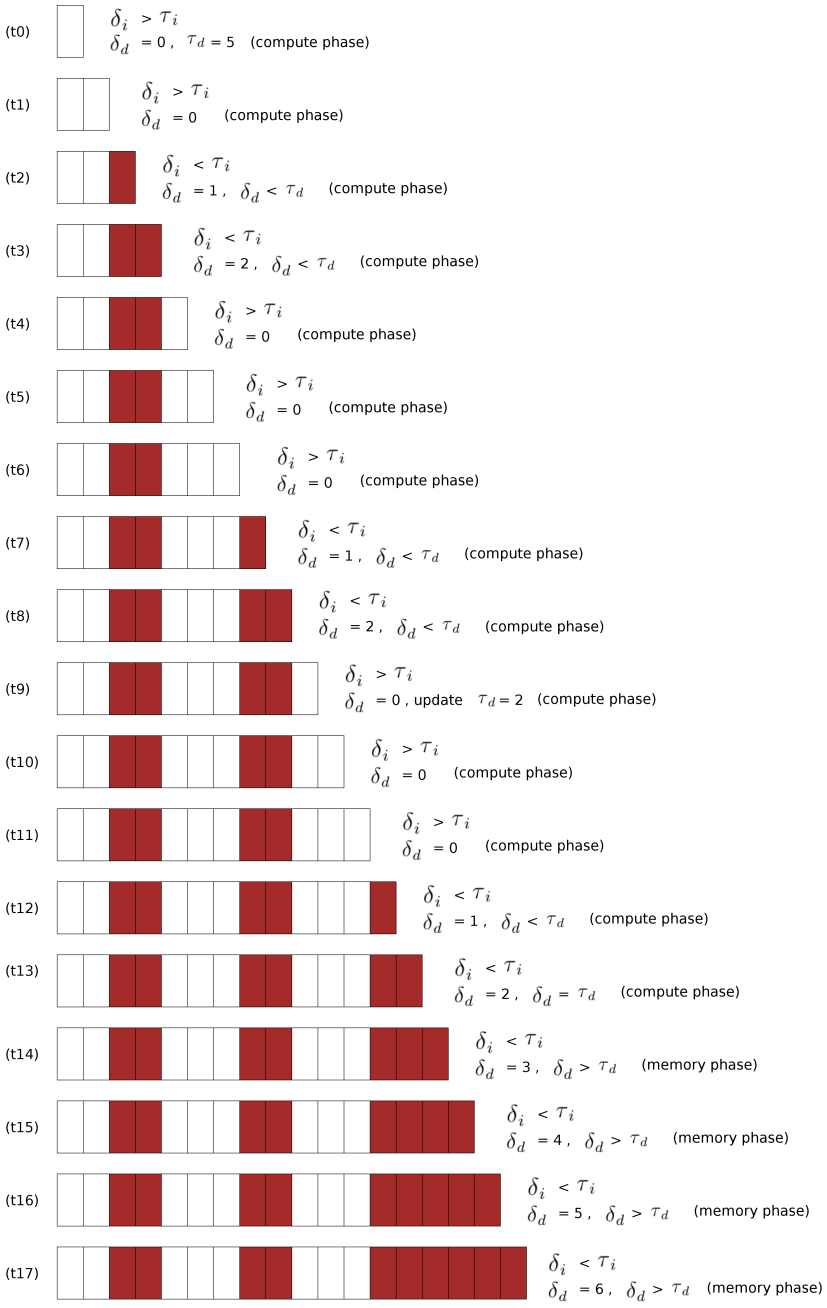

Figure 4.3: Adaptive compute-phase prediction.

the change in execution phase much earlier. Thus, adaptive compute-phase prediction increases the accuracy of prediction and reduces the time required which leads to improved overall performance and fairness.

Likewise, if the prediction seems to be wrong (i.e., after observing the third consecutive access that does not satisfy the interval constraint, the thread is considered to be in memory phase; although it satisfies the interval constraint on the fourth access), then the distance threshold (τ_d) is updated accordingly. An illustration of misprediction and updating distance threshold (τ_d) is given in Figure 4.4.

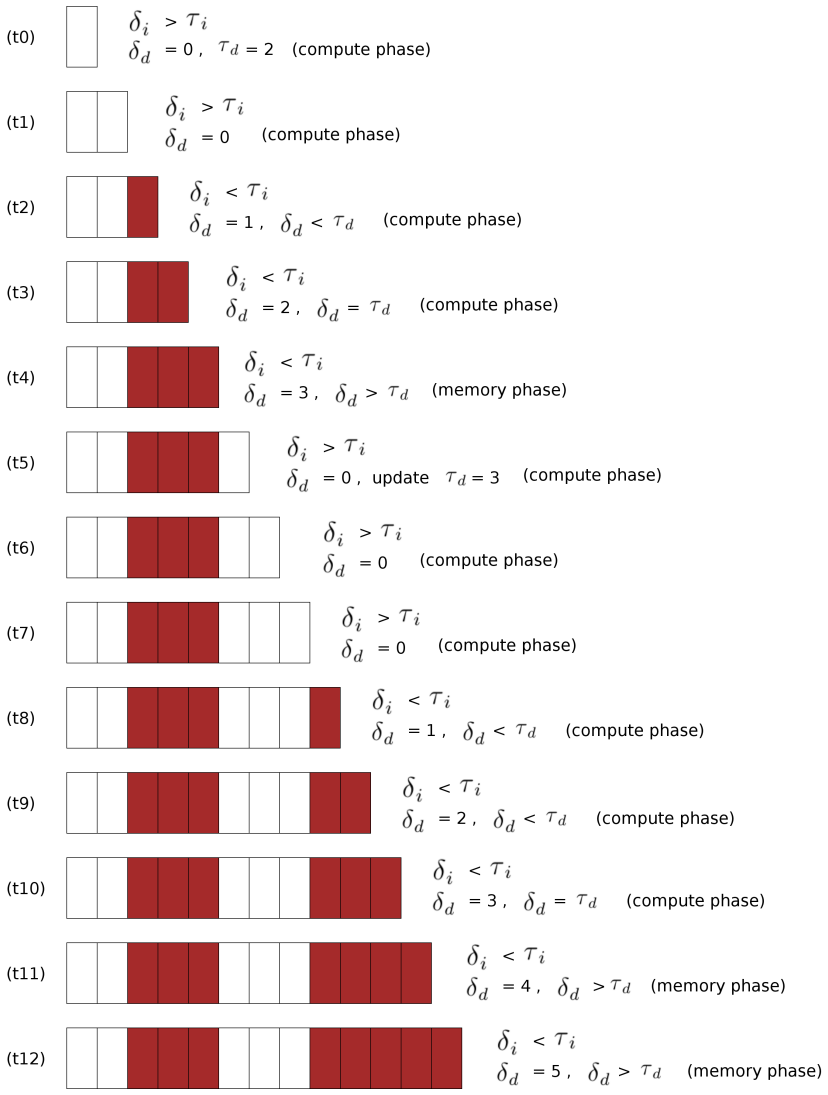

Figure 4.4: Updating a threshold in adaptive compute-phase prediction.

Instead of using predefined distance threshold to predict the execution phase of a thread, adaptive compute-phase prediction uses the upper limit to adaptively determine distance threshold (τ_d). Adaptively determined distance threshold (τ_d) can take a value between 1 and predefined distance threshold.

4.7 Adaptive Thread Prioritization

As described earlier, threads are classified into two groups. Threads in compute phase are prioritized over threads in memory phase. However, it is possible to have multiple threads in compute phase. In the scheduler of Ishii et al., the memory requests of threads in compute phase are serviced in the order they have received. Although it allows threads to make progress and keep cores busy, it misses possible performance benefits that could be obtained through fine-grained prioritization among threads of the same group.

We observed that prioritizing threads based on their potentials of making more progress on their execution increases the system performance even further. For this reason, we enhanced prioritization scheme of Ishii et al. in a way that threads in the same group are prioritized based on their potentials of making progress in their execution. when their memory requests serviced by the memory controller. We call this fine-grained prioritization scheme as *adaptive thread prioritization* since the priorities of threads are determined on the fly.

The usage of adaptive thread prioritization differs for threads in different groups (i.e., memory-non-intensive and memory-intensive). Adaptive thread prioritization works for threads in compute phase as follows. Among the threads in compute phase, the one that has the highest potential to make more progress is prioritized. On the other hand, threads in memory phase are prioritized based on whether they exhibit page hit and rank/bank locality. The adaptive thread prioritization is used as a tie breaker for threads in memory phase when there are multiple threads that exhibit page hit or rank/bank locality with recent memory accesses.

Priority	Reason for Prioritization	Command	Phase
1	timeout	no matter	no matter
2	low-MLP	no matter	no matter
3	adaptive prioritization	column access	compute
4	adaptive prioritization	activate	compute
5	adaptive prioritization	precharge	compute
6	page hit/adaptive prior.	column access	memory
7	rank/bank locality/adaptive prior.	activate	memory
8	rank/bank locality/adaptive prior.	precharge	memory

Table 4.1: Prioritization policy for the read requests.

The reason behind prioritizing threads in compute phase based on the progress they can make (i.e., employing adaptive thread prioritization) is to keep cores busy as much as possible. While cores are kept busy to execute threads in compute phase, the memory controller can service to memory requests of other threads. Thus, a thread that has more potential to keep a core busy for a longer period of time is prioritized over others.

On the other hand, if there is no thread in compute phase, then the main goal becomes to maximize memory throughput and reduce latency. For this reason, threads that exhibit row-buffer hit or rank/bank locality are given higher priorities. If there is no row-buffer hit, then the threads accessing the same bank/rank that was accessed recently are prioritized. When there are multiple threads that exhibit row-buffer hit, or bank/rank locality with recent memory access, then the adaptive thread prioritization is used to decide which thread is going to be prioritized.

We also employ aging in order to provide fair access to the memory. After a certain period of time, regardless of whether a thread is in compute phase or not, it is given the highest priority to avoid starvation. Threads that have low memory-level parallelism are also prioritized over other threads to let them finish their memory operations and continue on their execution as soon as possible. Table 4.1 summarizes the priority policy for the read requests. The smallest number in the table indicates the highest priority while the biggest number indicates the lowest priority.

Chapter 5

Evaluations

We provide extensive evaluations to present and analyze the effectiveness of proposed adaptive cache-hierarchy-aware thread scheduling in Section 5.1. Following that, we evaluate the adaptive compute-phase prediction and thread prioritization approach in Section 5.2.

5.1 Adaptive Cache-hierarchy-aware Thread Scheduling

5.1.1 Simulation Environment

We performed our experiments on multi2sim simulation framework that is developed to evaluate multicore-multithreaded processors [42]. Otherwise specified, we used the configuration given in Table 5.1 for chip multiprocessor and main memory.

We used PARSEC benchmarks to evaluate our proposed adaptive cache-hierarchy-aware thread scheduling algorithm. PARSEC is a set of benchmarks consists of multithreaded programs. It focuses on emerging workloads and was

4 cores
2 threads per core
Private L1 cache (combined), sets = 16, assoc. = 2
Policy = LRU, block size = 64, latency = 1
Shared L2 cache, sets = 64, assoc. = 4
Policy = LRU, block size = 64, latency = 10
Main memory, sets = 128, assoc. = 8, policy = LRU
Block size = 64, latency = 100
L1-L2 bus in/out bandwidth = 72
L2-Main memory bus in/out bandwidth = 264

Table 5.1: Chip multiprocessor and memory configuration for evaluations.

designed to be a representative set of next-generation shared-memory programs for chip multiprocessors [43]. In our experiments, we used eight benchmarks from PARSEC suite. Details of the benchmarks are given in Table 5.2.

blackscholes	Option pricing with Black-Scholes Partial Differential Equation (PDE)
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design
dedup	Next-generation compression with data deduplication
facesim	Simulates the motions of a human face
fluidanimate	Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
freqmine	Frequent itemset mining
vips	Image processing
x264	H.264 video encoding

Table 5.2: PARSEC benchmarks used in evaluations.

dedup uses the pipeline parallelization model with a dedicated pool of threads for each pipeline stage. *facesim* and *fluidanimate* are streaming programs. *blackscholes*, *canneal*, *freqmine*, *vips* and *x264* are data-level parallel programs with different amount and patterns of synchronizations and inter-thread communications.

At the very beginning of evaluations, we collected profiling regarding all the benchmarks. We run each benchmark along with other benchmarks one by one

on the same core and observed their respective performances. Then, we select the best schedules that maximize the performance (i.e., IPC) by using this profiling. At each interval, we scheduled threads in a way that the overall performance of the IPC is maximized. We referred these schedules as *dynamic-offline* and we used them as a baseline to compare against the proposed adaptive cache-hierarchy-aware thread scheduler. We also compared our adaptive cache-hierarchy-aware thread scheduler with possible static schedules.

Although IPC of threads obtained during offline profiling do not match the one obtained on the fly due to interactions of other scheduled threads, it provides a very good estimate of the highest IPC that can be achieved. Throughout the experiments, we observed that adaptive cache-hierarchy-aware thread scheduling outperforms static schedules and it is very close to the IPC achieved by dynamic-offline schedule.

Sch.	**Core 0**	**Core 1**	**Core 2**	**Core 3**
S1	blackscholes-vips	canneal-dedup	facesim-x264	fluidanimate-freqmine
S2	blackscholes-canneal	vips-dedup	facesim-fluidanimate	x264-freqmine
S3	blackscholes-dedup	vips-canneal	facesim-freqmine	x264-fluidanimate
S4	blackscholes-facesim	vips-fluidanimate	canneal-x264	dedup-freqmine
S5	blackscholes-x264	vips-freqmine	canneal-facesim	dedup-fluidanimate
S6	blackscholes-fluidanimate	vips-x264	canneal-freqmine	dedup-facesim
S7	blackscholes-freqmine	vips-facesim	canneal-fluidanimate	dedup-x264

Table 5.3: Static schedules used in evaluations.

We have generated seven different static schedules. Since there are eight benchmarks, we allowed a benchmark to run with different one in each schedule. By doing so, we aimed to cover all possible schedules for eight benchmarks (running on 4-core chip multiprocessor). We permuted the scheduled threads and generate distinct thread combinations. Since there are eight benchmarks, each benchmark can be scheduled with the remaining seven benchmarks at most. Note that, it does not matter on which core the two threads are scheduled; however, it matters which threads are scheduled together. The static schedules generated and corresponding threads running on cores are given in Table 5.3.

5.1.2 The Effect of Scheduling on System Performance

In this set of evaluations, we compared the effect of different scheduling schemes on performance for each benchmark (i.e., IPC). Figure 5.1 shows the IPC of each benchmark under different schedules.

As it can be seen from Figure 5.1, different schedules increase the performance for different benchmarks. There is no single schedule that outperforms the others for all benchmarks. This is also true for our proposed adaptive cache-hierarchy-aware scheduler. An important observation from this figure is that it is necessary to understand the main dynamics of the overall performance. Instead of increasing the performance of a particular benchmark, it is more desirable to find a balance among the performance of all threads. Our adaptive cache-hierarchy-aware scheduler works towards this goal. It tries to maximize the performance of overall system, not the performance of a particular thread. So, adaptive cache-hierarchy-aware scheduler does not favor (unfairly) a particular thread that may contribute to the overall performance the most (i.e., a thread that has highest potential to increase IPC in case of more resources are given to it). Instead, it tries to find a balance among threads where they contribute to the overall system performance.

In this set of evaluations, we compared the overall system performance provided by different scheduling schemes. The results are given in Figure 5.2. As it can be seen, our adaptive cache-hierarchy-aware scheduler outperforms all the static schedules, and barely left behind the dynamic-offline scheduling. Note that dynamic-offline provides the highest IPC that can be achieved; however, it requires profiling in advance. For this reason, dynamic-offline is not a practical scheduler, but it helps us to evaluate our approach against. Figure 5.2 also shows the effectiveness of adaptive cache-hierarchy-aware scheduler on maximizing overall system performance without unfairly favoring certain benchmarks.

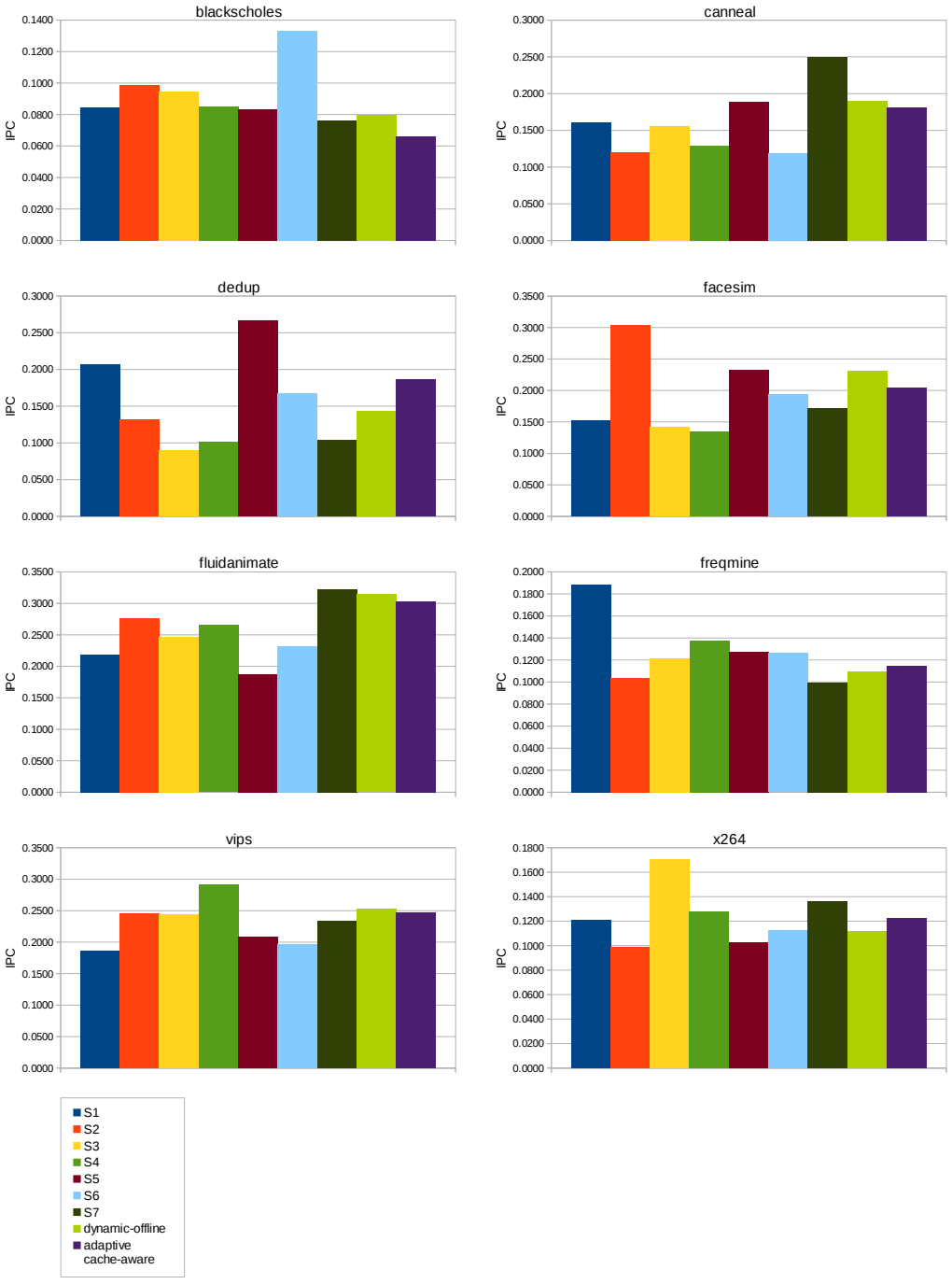

Figure 5.1: Performance of benchmarks under different scheduling schemes.

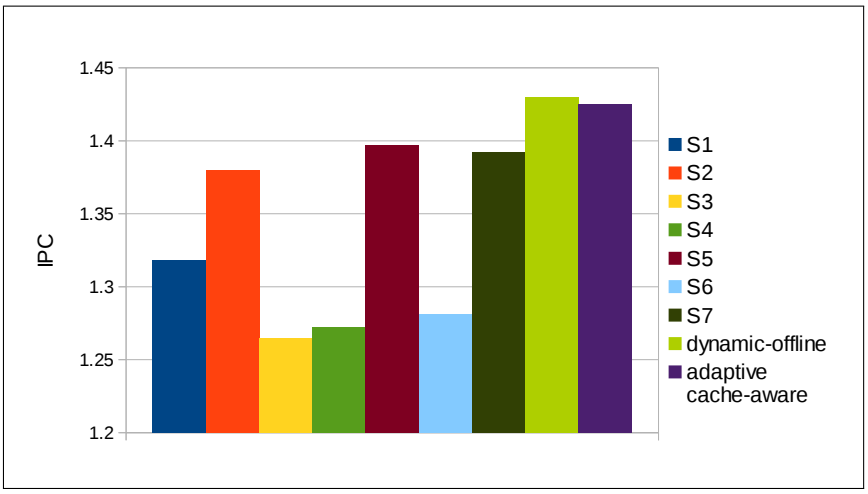

Figure 5.2: Comparison of overall system performance under different scheduling schemes.

5.1.3 Slowdown of Benchmarks

It may be misleading to focus solely on IPC when evaluating scheduling schemes. As we discussed in Section 5.1.2, the overall system performance can be maximized by unfairly favoring threads that have higher potential to contribute to the overall system IPC. However, such an approach is not desirable in most cases. Instead, the system performance has to be maximized such that each thread contributes to the overall system performance as much as possible while interference with other threads is minimized. In other words, fairness should not be traded for performance. The schedulers proposed for chip multiprocessors should also take slowdown of threads into account while trying to maximize system performance.

In this set of evaluations, we analyzed the behavior of a particular benchmark when it is scheduled with another one. Table 5.4 shows the slowdown of benchmarks when they run concurrently with another benchmark. Slowdown specifies the degree of vulnerability of a benchmark to the interference of the other thread.

The slowdowns given in Table 5.4 are observed on a single core that can run two threads concurrently. There is only one schedule possible, since two threads exist in this set of evaluations.

As noted earlier, a desired scheduler should maximize the system performance

	running with							
	blackscholes	canneal	dedup	facesim	fluidanimate	freqmine	vips	x264
blackscholes		2.1	3.4	2.5	1.7	2.6	2.9	3.5
canneal	2.8		4.3	2.5	1.5	2.8	2.7	3.0
dedup	2.1	1.6		1.7	1.3	1.9	1.9	2.1
facesim	3.2	2.6	4.4		1.7	2.8	3.2	3.3
fluidanimate	2.9	2.3	6.2	2.4		2.7	2.6	2.9
freqmine	3.3	2.8	4.3	2.7	1.9		3.3	3.4
vips	1.8	1.4	2.2	1.5	1.2	1.6		1.9
x264	3.0	2.0	2.9	2.1	1.5	2.2	2.5	

Table 5.4: Slowdown of a thread when scheduled with another thread on the same core.

while preventing unfairness. Since slowdowns of benchmarks provide a notion of fairness, we can evaluate the effectiveness of schedulers on maximizing system performance fairly. Table 5.5 shows the slowdown of benchmarks when scheduled with another thread on the same core, while the rest of the benchmarks are running on other cores under the static scheduling. The slowdowns presented in the rest of the section are observed on a quad-core chip multiprocessor (i.e., the configuration given in Table 5.1).

	running with							
	blackscholes	canneal	dedup	facesim	fluidanimate	freqmine	vips	x264
blackscholes		4.8	5.0	5.6	3.6	6.3	5.7	5.7
canneal	6.1		4.6	3.9	2.9	6.2	4.7	5.7
dedup	5.5	2.4		3.0	1.9	4.9	3.8	4.8
facesim	6.7	3.9	4.6		3.0	6.3	5.2	5.9
fluidanimate	5.6	4.0	6.9	4.6		5.9	4.9	5.2
freqmine	8.0	6.3	5.8	6.6	3.6		6.2	7.7
vips	3.0	2.3	2.3	2.4	1.9	2.7		2.9
x264	4.7	3.8	3.6	4.0	2.8	4.9	4.3	

Table 5.5: Slowdown of a thread when scheduled with another thread under the static scheduling scheme.

Compared to static schedules, threads can be scheduled with different threads throughout the execution in dynamic-offline scheduling. For this reason, we represented overall slowdown for each benchmark and difference between the minimum, maximum and average slowdown observed under the static schedules. Table 5.6 shows the slowdown of benchmarks under dynamic-offline scheduling. The first column of the table specifies the overall slowdown of benchmarks. The second, third and fourth columns of the table specify how much overall slowdown of a thread deviates from minimum, maximum and average slowdown observed under the static schedules, respectively. In other words, the second column of the table

is calculated as the subtraction of the minimum slowdown for a thread under the static scheduling from the slowdown of a thread under the dynamic-offline scheduling. The third and fourth columns are calculated in the same manner.

		δ from		
		min. (+/-)	max. (+/-)	avg. (+/-)
blackscholes	6.0	2.4	-0.3	0.7
canneal	3.9	0.9	-2.3	-1.0
dedup	3.5	1.6	-2.0	-0.3
facesim	3.9	0.9	-2.8	-1.2
fluidanimate	4.1	0.1	-2.8	-1.2
freqmine	7.3	3.6	-0.8	0.9
vips	2.2	0.3	-0.8	-0.3
x264	4.3	1.5	-0.5	0.3

Table 5.6: Slowdown of a thread when scheduled with another thread under the dynamic-offline scheduling scheme.

Similar to the dynamic-offline scheduling, adaptive cache-hierarchy-aware scheduling allows threads to be scheduled with different threads throughout the execution. Table 5.7 shows the slowdown of benchmarks under the adaptive cache-hierarchy-aware scheduling. Likewise, the first column of the table specifies the overall slowdown of benchmarks. The second, third and fourth columns of the table specify how much overall slowdown of a thread deviates from minimum, maximum and average slowdown observed under the static schedules, respectively.

		δ from		
		min. (+/-)	max. (+/-)	avg. (+/-)
blackscholes	7.3	3.7	1.0	2.0
canneal	4.1	1.1	-2.1	-0.8
dedup	2.7	0.8	-2.8	-1.1
facesim	4.4	1.4	-2.3	-0.7
fluidanimate	4.2	0.2	-2.6	-1.1
freqmine	6.9	3.3	-1.1	0.6
vips	2.3	0.3	-0.7	-0.2
x264	4.0	1.1	-0.9	-0.1

Table 5.7: Slowdown of a thread when scheduled with another thread under the cache-hierarchy-aware scheduling.

As indicated earlier, a desired scheduler should also try to minimize average slowdown while trying to increase system performance. To this end, our proposed adaptive cache-hierarchy-aware scheduler obtains decent slowdown and provides higher system performance. Figure 5.3 shows the comparison of slowdown of all the benchmarks running under different scheduling schemes.

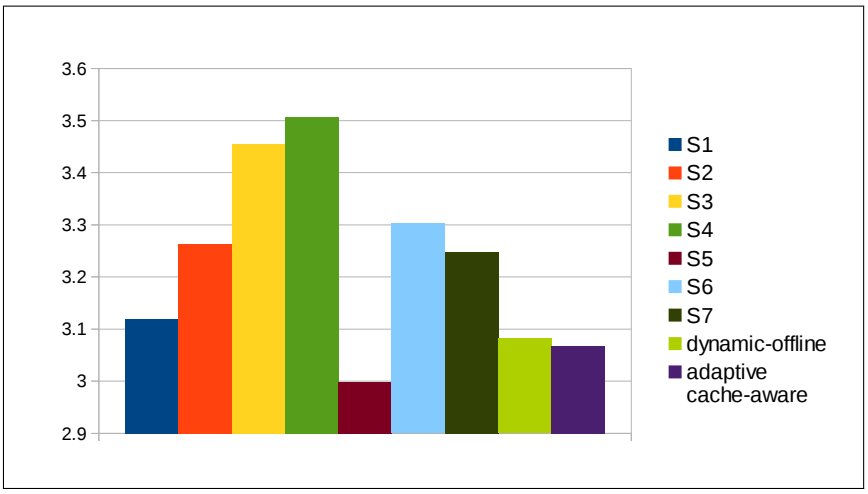

Figure 5.3: Slowdowns of benchmarks under different scheduling schemes.

Proposed adaptive cache-hierarchy-aware scheduler outperforms all other scheduling schemes, including dynamic-offline, except the fifth static schedule (i.e., S5). However, slowdown observed in S5 and adaptive cache-hierarchy-aware scheduler are very close. In addition, although S5 has lower slowdown, it does not improve the system performance as much as adaptive cache-hierarchy-aware scheduler. If we consider both slowdown and system performance, we can conclude that the adaptive cache-hierarchy-aware achieves better results compared to S5.

5.1.4 The Effect of Scheduling on Cache Performance

In this set of evaluations, we analyzed the effect of cache sizes on system performance. The reason behind this analysis is to determine the sensitivity of benchmarks to the allocated cache resources. Figure 5.4 shows the miss per kilo instruction with respect to L2 cache size for the benchmarks.

Although, all benchmarks benefit from increased L2 cache sizes, dedup and x264 benefit more compared to others. However, almost all benchmarks saturate at 128 KB. Such observations are important for designing cache architecture for chip multiprocessors.

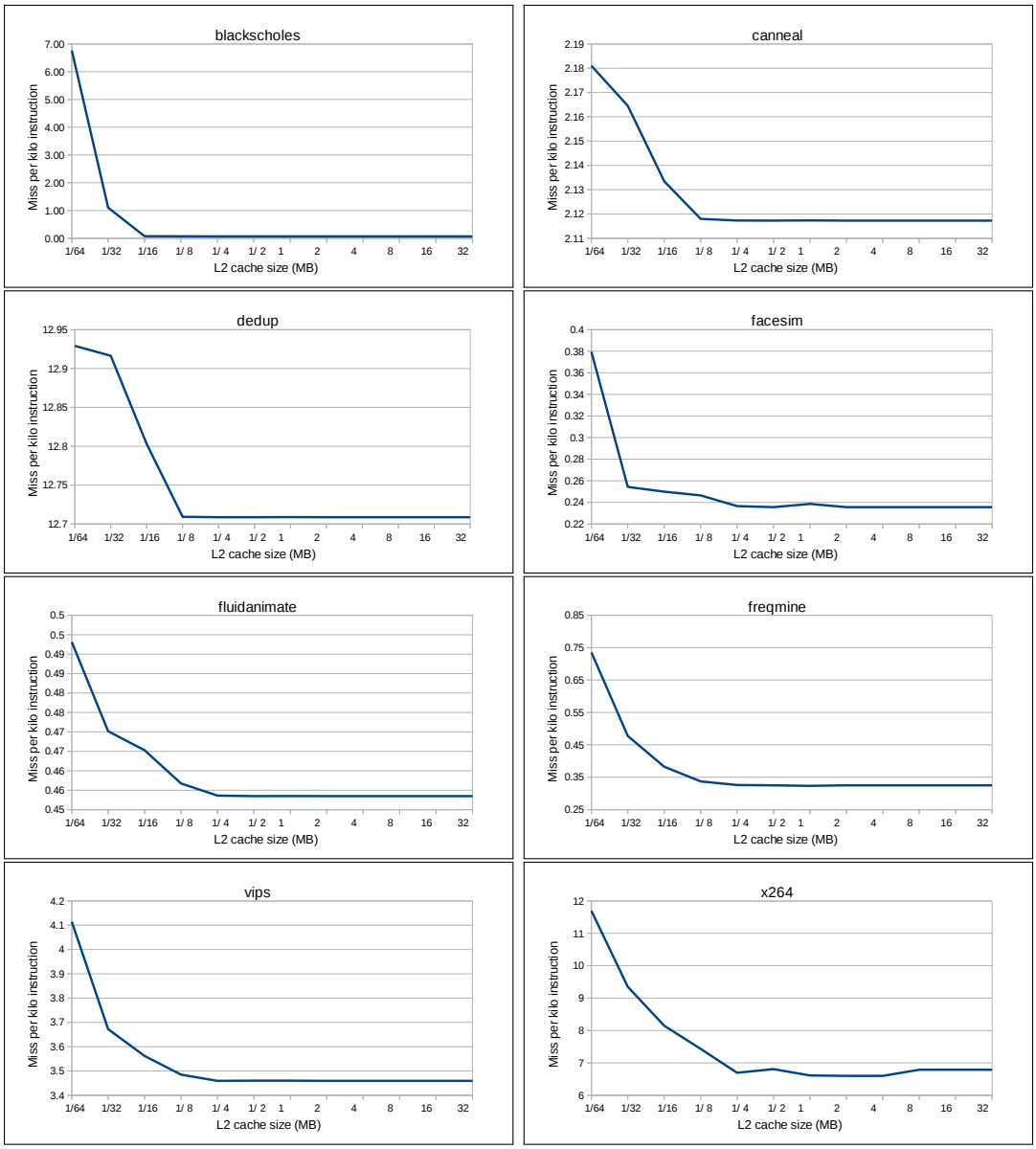

Figure 5.4: Miss per kilo instructions vs. L2 cache size for the benchmarks.

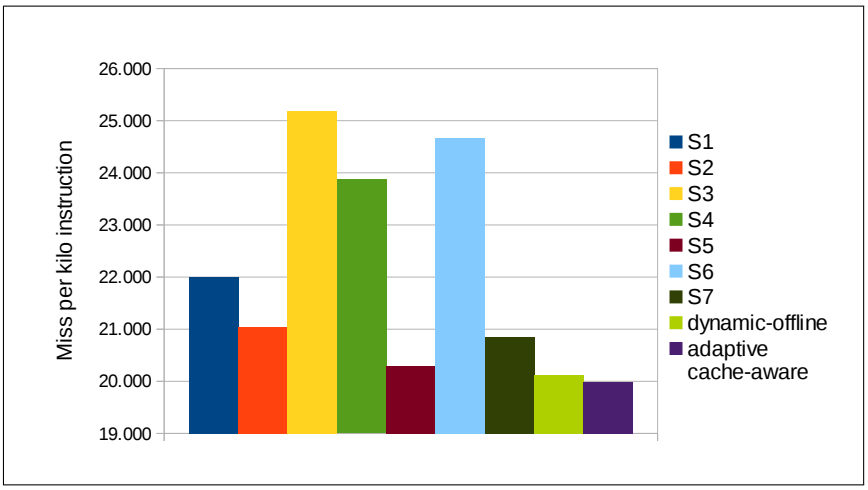

Figure 5.5: L2 miss per kilo instruction of benchmarks under different scheduling schemes.

An important metric to evaluate the effectiveness of a scheduler is LLC (i.e., L2 in our case). In this section, we justify why considering higher level of cache hierarchy in scheduling decisions will eventually affect the performance of LLC.

We used L2 miss per kilo instruction as a metric to evaluate the effectiveness of the scheduling schemes. The scheduling scheme that minimizes the L2 miss per kilo instruction is more desirable than the others. Figure 5.5 shows the number of L2 misses per kilo instruction for different scheduling schemes. As it can be seen, our proposed adaptive cache-hierarchy-aware scheduler has the lowest L2 miss per kilo instruction. Therefore, justifying our claim on the importance of higher level caches on LLC performance.

Figure 5.6 shows the L2 miss ratios of the benchmarks for different scheduling schemes. Although adaptive cache-hierarchy-aware scheduler does not have the minimum L2 miss ratio, we believe that it is reasonable. The results of L2 miss ratio might be misleading if it is considered without taking corresponding system performance (i.e., IPC) into account. There might be cases where the threads make slow progress due to the contention on shared resources, thus generating less number of cache accesses. These cache accesses might have higher hit ratio. On the other hand, there might be cases where threads make faster progress, thanks to wise scheduler that reduces the contention on shared resources, thus generate

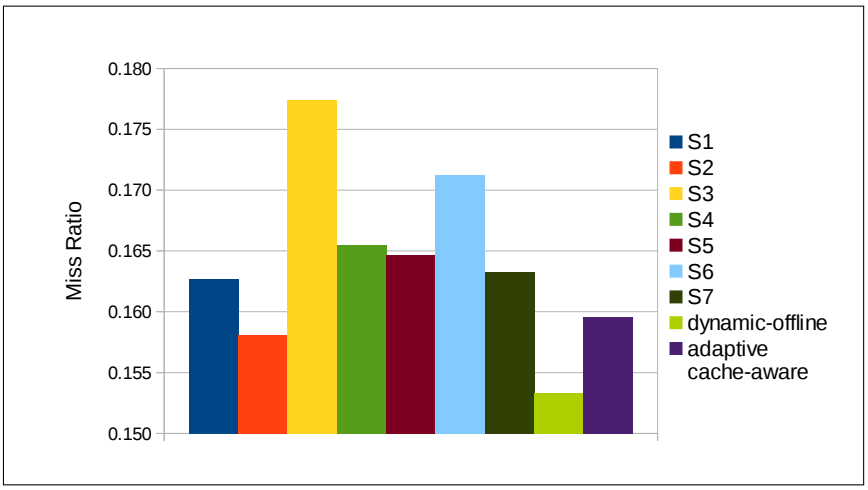

Figure 5.6: L2 miss ratio of benchmarks under different scheduling schemes.

more cache accesses. These accesses might have lower hit ratio compared to the first case. However, we can not conclude that the scheduling in the first case is better than the second one, solely it has the lower miss ratio. In fact, we need to look at what would be the miss ratio when the threads in the first case would also make the same progress as the threads in the second case. For this reason, we used L2 miss per kilo instruction as a metric for performance of cache, instead of L2 miss ratio.

As it can be seen from Figure 5.6, adaptive cache-hierarchy-aware scheduling has higher L2 miss ratio compared to the second static schedule (i.e. S2). However, the overall system performance of adaptive cache-hierarchy-aware scheduling is comparably higher than the performance of S2.

Likewise, the adaptive cache-hierarchy-aware scheduling utilizes the L1 cache much better compared to other scheduling schemes. Figure 5.7 shows the number of L1 misses per kilo instruction for different schedules. Adaptive cache-hierarchy-aware scheduler outperforms other scheduling schemes except the fifth static scheduling (i.e. S5). Although S5 has lower misses per kilo instruction, its overall system performance is lower than the adaptive cache-hierarchy-aware scheduler. Despite it seems awkward, there is a logical reason behind it. The accesses to L1 that are misses go to L2 cache. Some of these misses are also

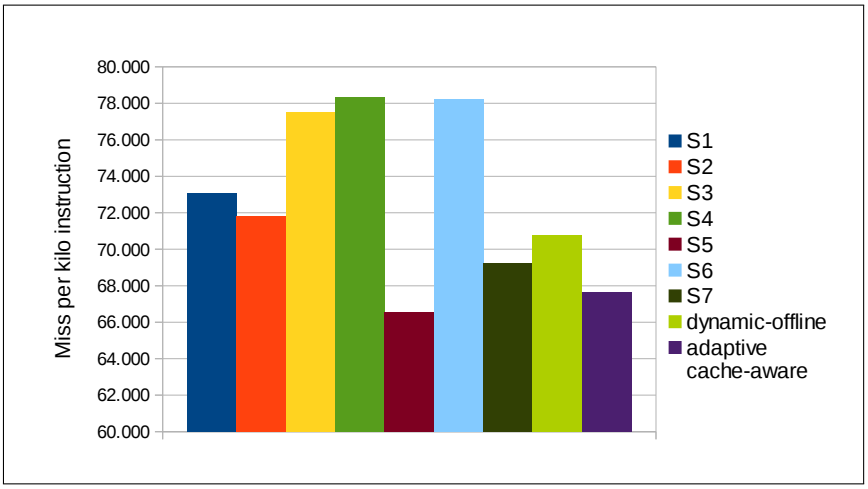

Figure 5.7: L1 miss per kilo instruction under different scheduling schemes.

misses in L2 cache. Thus, these misses require high latency main memory accesses. Compared to adaptive cache-hierarchy-aware scheduler, the L1 misses of S5 are not found in L2, so they have to be fetched from main memory. That is why S5 has lower IPC than adaptive cache-hierarchy-aware scheduler, although it has lower L1 miss per kilo instruction.

Figure 5.8 shows the L1 miss ratio of benchmarks under different scheduling schemes. Similar to the L2 miss ratio, adaptive cache-hierarchy-aware scheduler does not have the minimum L1 miss ratio. The same argument is also valid in this case. The results of L1 miss ratio might be misleading if it is considered without taking corresponding system performance (i.e., IPC) and L2 miss per kilo instruction into account. Note that, misses on L1 might be misses on L2, as well. In such cases, high latency memory access reduces the system performance. This is why the seventh static scheduling (i.e., S7) has lower IPC compared to adaptive cache-hierarchy-aware scheduler, although it has lower L1 miss ratio as shown in Figure 5.8. The same observation is valid for dynamic-offline. Although adaptive cache-hierarchy-aware scheduler has lower L1 miss ratio, dynamic-offline has higher IPC compared to adaptive cache-hierarchy-aware scheduler.

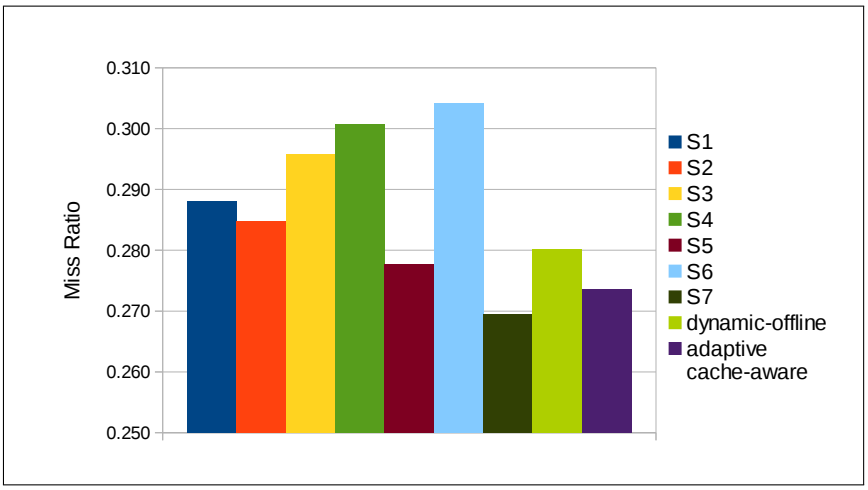

Figure 5.8: L1 miss ratio of benchmarks under different scheduling schemes.

5.1.5 Sensitivity of Performance to the Thread Quantum

Threads have a time quantum that is specied as the number of cycles to be executed. When threads exceed this quantum, the adaptive cache-hierarchy-aware scheduler updates the scheduling decisions as explained in Section 3.10. After this update, the quanta of threads are reset and they run on specified cores until the time quantum is exceeded again.

The number of cycles specified for quanta of threads has an influence on the performance. When the length of quantum is short (i.e., small number of cycles), the scheduling decision has to be made more often. The drawback of short quantum is that the decision of scheduling becomes vulnerable to short bursts and fluctuations on thread behaviors. In addition, the length of quantum may not be sufficient to compensate the overhead due to thread migration (in case a thread is scheduled on a different core).

On the other hand, if the length of quantum is too long, then the scheduling decision has to be made less often. The drawback of long quantum is that the execution characteristics of threads may change which may result in with inappropriate scheduling. For this reason, the length of quantum has an impact on the overall system performance.

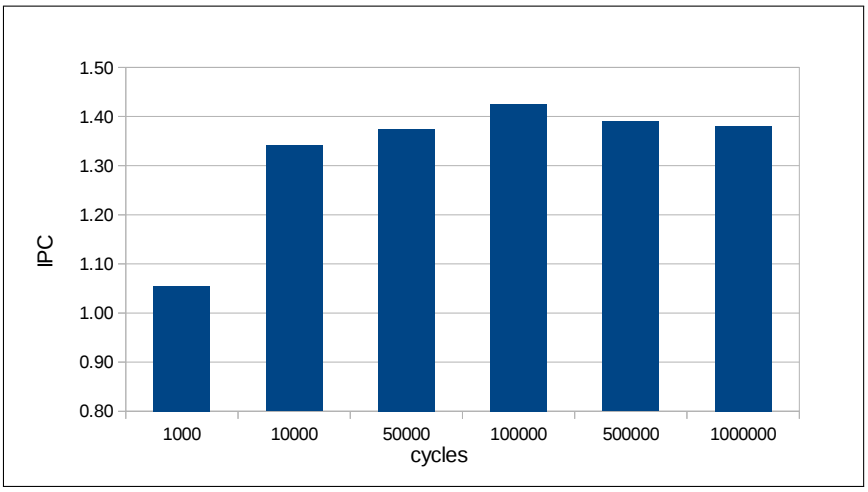

Figure 5.9: System performance for different thread quantum lengths.

Figure 5.9 shows the effect of quantum length on overall system performance. As it can be seen, the quantum of 100,000 cycles maximizes the system performance. The results reported in this chapter are gathered by using a quantum with 100,000 cycles.

5.1.6 Sensitivity of Performance to the Weights of Thread Attributes

The position of a bit (that represents a particular attribute of a thread) in an attribute vector determines the relative weight of the attribute of interest. The bit for an attribute with the most significant bit in the attribute vector naturally obtains the highest weight, while the bit for an attribute with the least significant bit in the attribute vector obtains the lowest weight. The weights of attributes (i.e., position of corresponding bits in attribute vector) have an impact on the overall system performance.

Figure 5.10 shows the effect of changing the position of bits for attributes in attribute vector. Each column represents different ordering of bits for attributes in an attribute vector. For example, ADI means that the bit for aggressiveness is the most significant bit in attribute vector, thus it has the highest weight. On the other hand, the bit for inefficacy is the least significant bit in the attribute

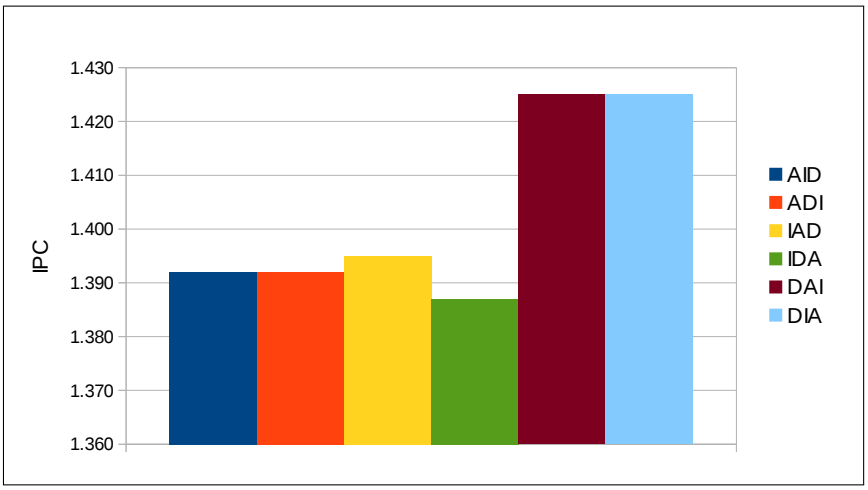

Figure 5.10: System performance changes with respect to relative weights of thread attributes.

vector, thus it has the lowest weight.

We used DAI (i.e., density being the most important attribute and inefficacy being the least important attribute) for the results reported in this chapter. The relative importance of density is comprehensible, since L1 cache is limited in size and the contention for cache blocks is severe. Thus, giving more weight to density allows adaptive cache-hierarchy-aware scheduler to have tendency to schedule threads that reduces such contention.

5.1.7 Sensitivity of Performance to the Resolution of Thread Attributes

Figure 5.11 shows the effect of the number of bits used for each attribute of a thread in an attribute vector. Fine-grained scores are possible when higher number of bits is used. Although one bit for each attribute is fairly enough in case of a small number of threads, it becomes harder to differentiate candidate schedules (i.e., deciding which one is better) when the number of threads increases. This is the case, because there is less number of distinct CoScores possible with less number of bits. For this reason, increasing the number of bits per attribute in

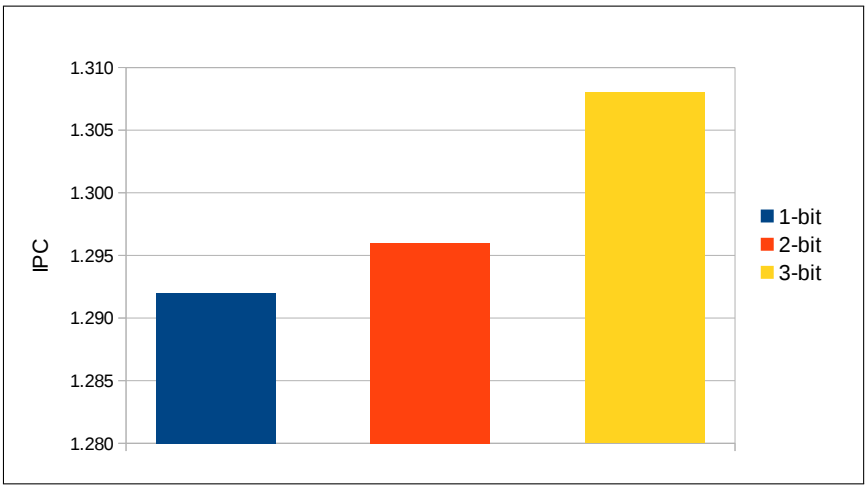

Figure 5.11: System performance with respect to the number of bits used to represent each attribute in an attribute vector.

attribute vector enables fine-grained scores. Figure 5.11 shows the overall system performance for 16 threads running on 8 cores. Notice that, increasing the number of bits results with a higher system performance.

It provides marginal benefit to use more bits for each attribute for our base simulation environment (i.e., 8 threads running on 4 cores). For this reason, we use a single bit per attribute for the results reported.

5.1.8 Sensitivity of Performance to Scoring Thresholds

Each thread has a multi-metric score based on its attributes: aggressiveness, density and inefficacy. The decision of a thread being aggressive/complaisant, dense/sparse and sterile/prolific is given through respective thresholds. Each attribute has its own threshold, where these thresholds are determined empirically. Table 5.8 shows the overall system performance with respect to different thresholds for the attributes.

The aggressiveness, density and inefficacy of a thread are determined as specified in Section 3.9. The overall system performance is maximized when thresholds are $\tau_a = 1.6$, $\tau_d = 0.3$ and $\tau_i = 0.4$ where τ_a, τ_d and τ_i are thresholds for

		A			
D	I	0.3	0.4	0.5	0.6
1.4	0.2	1.394	1.402	1.392	1.4
	0.3	1.374	1.402	1.394	1.381
	0.4	1.388	1.386	1.381	1.394
1.5	0.2	1.409	1.412	1.389	1.389
	0.3	1.393	1.393	1.398	1.405
	0.4	1.416	1.393	1.391	1.391
1.6	0.2	1.422	1.402	1.411	1.416
	0.3	1.407	**1.425**	1.409	1.409
	0.4	1.414	1.395	1.409	1.416
1.7	0.2	1.403	1.378	1.382	1.419
	0.3	1.405	1.341	1.393	1.403
	0.4	1.404	1.39	1.393	1.393

Table 5.8: System performance with respect to the attribute thresholds for threads.

aggressiveness, density and inefficacy, respectively. The maximum performance obtained is specified as bold in the table. We used $\tau_a = 1.6$, $\tau_d = 0.3$ and $\tau_i = 0.4$ for the results reported in this chapter.

5.2 Adaptive Compute-phase Prediction and Thread Prioritization

5.2.1 Simulation Environment

We used the USIMM simulation framework [41] for the evaluations reported in this part of the thesis. The framework comes with a default memory subsystem configuration. During the evaluations, we used two different configurations for the memory subsystem. The first configuration has 1 channel with 2 ranks, where each rank has 8 banks, so the total number of banks is 16. The second configuration has 4 channels with 2 ranks in each channel. Every rank has 8 banks, so the total number of banks is 64. The overview of the system configurations used in this part of the thesis is given in Table 5.9 where N is the number of

Parameter	1-channel configuration	4-channel configuration
core clock speed	3.2 GHz	3.2 GHz
ROB Size	128	160
retire width	2	4
fetch width	4	4
memory bus speed	800 MHz (DDR)	800 MHz (DDR)
ranks per channel	2	2
banks per rank	8	8
rows per bank	$32768 \times N$	$32768 \times N$
columns per row	128	128
cache line size	64 B	64 B
write queue size	64	96
read queue size	∞	∞

Table 5.9: System configurations used in evaluations.

processing cores.

The number of cores used in simulations is a function of the number of benchmarks in the workload. Each benchmark is considered to be executed on a different core. Therefore, if there are N benchmarks in the workload, it is assumed that there are N cores in the system.

5.2.2 Workloads

In the USIMM simulation framework, 24 different benchmarks are provided. The composition of these benchmarks are as follows. Out of 24 benchmarks, 14 of them are from PARSEC, 2 of them are from biobench, 2 of them are from Spec CPU-2006, 5 of them are from commercial transaction processing benchmarks and 1 of them is from STREAM benchmarks.

To make a fair comparison of the schedulers and to prevent bias, we performed our evaluations on the same workload set given in USIMM for Memory Scheduling Championship (MSC) held in 3rd JILP Workshop on Computer Architecture Competitions [44]. Notice that Ishii et al. presented their Compute-phase Prediction with Writeback-Refresh Overlap scheduler (CP-WO) in this workshop.

From the given 24 benchmarks, 32 different workloads have been generated. The workloads generated and corresponding shorthand notations used in evaluations are given in Table 5.10.

Initially, we evaluate the effectiveness of adaptive compute-phase prediction (ACP) and adaptive thread prioritization (ATP), separately. Then, we employed both adaptive compute-phase prediction and adaptive thread prioritization in Writeback-Refresh Overlap scheduler. The proposed scheduler is a combination of ACP and ATP, namely adaptive compute-phase prediction and thread prioritization (ACP-TP). The evaluations in this chapter is divided into two sections. In the following section, we evaluate the impact of the proposed methods on the sum of execution times. In the other section, we evaluate the impact of the proposed methods on power consumption. In our study, the main performance metric of the schedulers is the sum of execution times.

5.2.3 The Effect of Scheduling on Sum of Execution Times

Figure 5.12 shows the comparison of Compute-phase Prediction with Writeback-Refresh Overlap (CP-WO), and adaptive compute-phase prediction (ACP). The sum of execution times are normalized with respect to First-Ready First-Come First-Serve (FR-FCFS) scheduler. Similarly, Figure 5.13 shows the comparison of CP-WO and ATP. As it can be seen from these figures, different workloads benefit from different adaptation. Some of the workloads benefit more from adaptive compute-phase prediction, such as w23; while others benefit more from adaptive thread prioritization, such as w31.

Figure 5.14 shows the comparison of CP-WO, and ACP-TP. As it can be seen from the Figure 5.14, ACP-TP outperforms the FR-FCFS for all workloads, except the workload 25. For workload 25, FR-FCFS is slightly better than ACP-TP. In overall, ACP-TP reduces the total sum of execution time up to 23.6% (on average 10.9%) and 12.9% (on average 1.2%) compared to FR-FCFS and CP-WO, respectively.

name	# of ch.	benchmarks
w1	1	MT0-canneal, MT1-canneal, MT2-canneal, MT3-canneal
w2	4	MT0-canneal, MT1-canneal, MT2-canneal, MT3-canneal
w3	4	fluidanimate, fluidanimate, swaptions, swaptions, commercial2, commercial2, ferret, ferret
w4	4	fluidanimate, fluidanimate, swaptions, swaptions, commercial2, commercial2, ferret, ferret, blackscholes, blackscholes, freqmine, freqmine, commercial1, commercial1, stream2, stream2
w5	4	commercial3, commercial3, commercial3, commercial3, commercial3, commercial3, commercial3, commercial3
w6	4	libquantum, libquantum, libquantum, mummer, mummer, mummer, tigr, tigr
w7	1	blackscholes, blackscholes, freqmine, freqmine
w8	4	blackscholes, blackscholes, freqmine, freqmine
w9	1	commercial2
w10	1	commercial1, commercial1
w11	1	commercial1, commercial1, commercial2, commercial2
w12	1	fluidanimate, swaptions, commercial2, commercial2
w13	1	facesim, facesim, ferret, ferret
w14	1	stream2, stream2, stream2, stream2
w15	4	commercial2
w16	4	commercial1, commercial1
w17	4	commercial1, commercial1, commercial2, commercial2
w18	4	fluidanimate, swaptions, commercial2, commercial2
w19	4	facesim, facesim, ferret, ferret
w20	4	stream2, stream2, stream2, stream2
w21	1	tigr, tigr
w22	1	libquantum, libquantum
w23	1	libquantum, libquantum, mummer, mummer
w24	1	leslie3d, leslie3d, leslie3d, leslie3d
w25	1	MT0-fluidanimate, MT1-fluidanimate, MT2-fluidanimate, MT3-fluidanimate
w26	1	commercial4, commercial4, commercial5, commercial5
w27	4	tigr, tigr
w28	4	libquantum, libquantum
w29	4	libquantum, libquantum, mummer, mummer
w30	4	leslie3d, leslie3d, leslie3d, leslie3d
w31	4	MT0-fluidanimate, MT1-fluidanimate, MT2-fluidanimate, MT3-fluidanimate
w32	4	commercial4, commercial4, commercial5, commercial5

Table 5.10: Workloads used in evaluations.

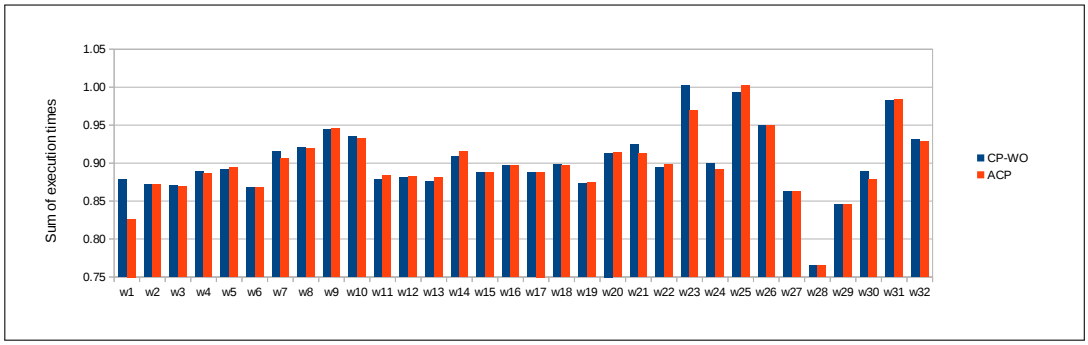

Figure 5.12: Comparison of CP-WO and ACP schedulers for the given workloads. Sum of execution times is normalized with respect to FR-FCFS.

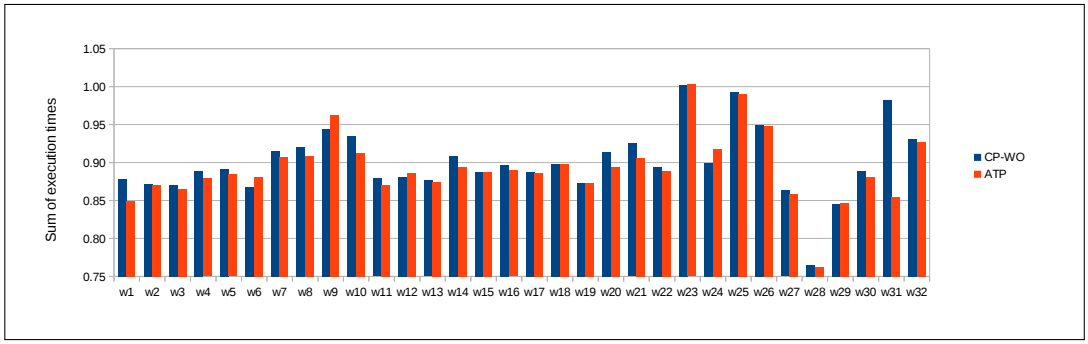

Figure 5.13: Comparison of CP-WO and ATP schedulers for the given workloads. Sum of execution times is normalized with respect to FR-FCFS.

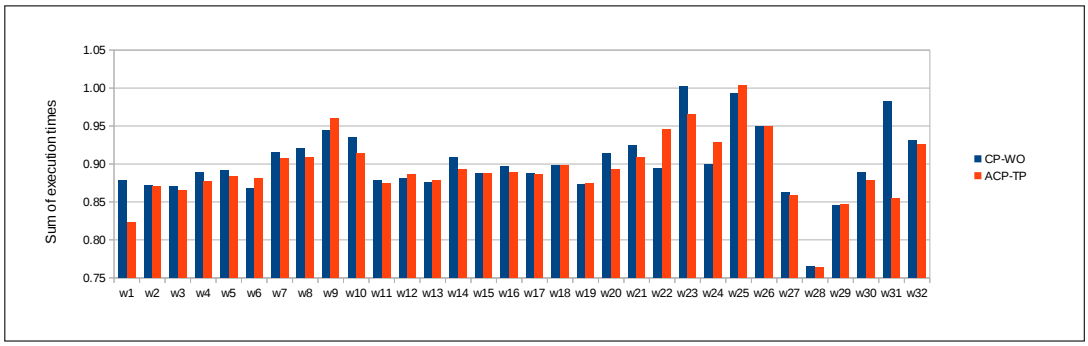

Figure 5.14: Comparison of CP-WO and ACP-TP schedulers for the given workloads. Sum of execution times is normalized with respect to FR-FCFS.

For the 22 out of 32 workloads ACP-TP outperforms CP-WO. For 10 of the workloads, CP-WO is better than ACP-TP. Out of these 10 workloads, 5 of them is less than 0.5% better compared to ACP-TP that can be omitted. When we look at the workloads in which ACP-TP left behind CP-WO, we can see that these are the workloads of a single benchmark (e.g. w9), or the same benchmark having multiple instances running on different cores (e.g. w22, w24, w25). In cases where there is no diversity in the memory access requests, similar to these workloads, the ACP-TP tries to prioritize the memory requests of threads; however, it does not provide any benefit since all the threads are of the same type (or there is only one thread). For this reason, the arrangements made on memory access requests for these workloads may show no benefit, even worse, they may degrade the performance. Since, we expect to have diversity and abundance in the tasks of real workloads, we consider such a flaw in ACP-TP as benign.

Figure 5.15 shows the total sum of execution times for the given workloads. It also shows the impact of adaptive compute phase prediction (i.e., ACP) and adaptive thread prioritization (i.e., ATP), separately. Notice that, the combination of these two (i.e., ACP-TP) has better performance compared to all others.

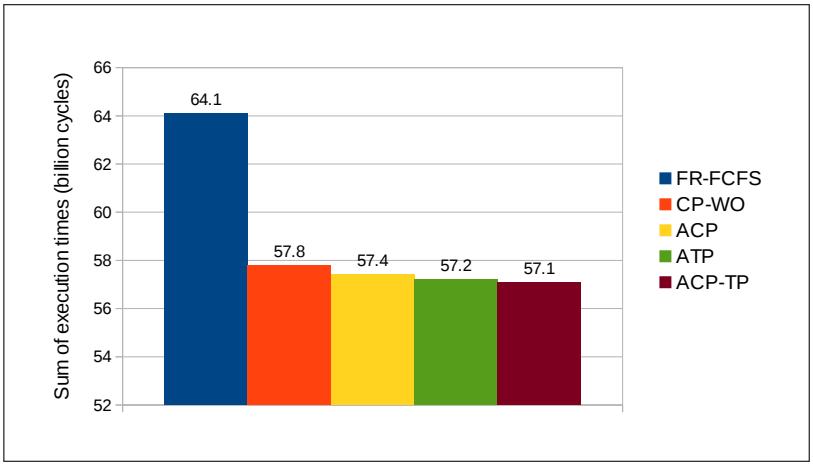

Figure 5.15: Total sum of execution times for the given workloads.

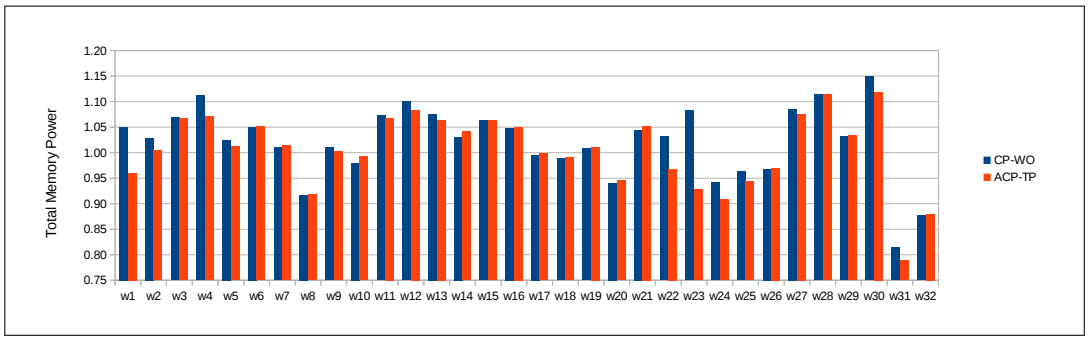

Figure 5.16: Comparison of total memory system power of CP-WO and ACP-TP schedulers for the given workloads. The total memory system power is normalized with respect to FR-FCFS.

5.2.4 The Effect of Scheduling on Power Consumption

Another concern in memory access scheduling is to minimize the power consumption of the memory subsystem. Since the performance and power consumption are conflicting goals most of the time, it is challenging to optimize both of them simultaneously. Although it is challenging, our adaptive compute-phase prediction and thread prioritization algorithm provides reasonable results. Figure 5.16 shows the comparison of total memory system power when the CP-WO and ACP-TP schedulers are used. The total memory power is normalized to power consumption of FR-FCFS. As it can be seen from the Figure 5.16, ACP-TP has lower memory power compared to CP-WO. On the other hand, FR-FCFS has the lowest memory power overall.

Similarly, Figure 5.17 shows the comparison of total system power when CP-WO and ACP-TP schedulers are used. The total system power is normalized to FR-FCFS. As it can be seen from the Figure 5.17, ACP-TP has the lowest total system power. This indicates that ACP-TP allows processing cores to run respective workloads faster which reduces the power consumed by processing cores (i.e., reduces the time of being idle, so the power consumed is reduced when processing core is idle). Although ACP-TP has higher power consumption on memory subsystem compared to FR-FCFS, the reduction of power consumption on processing cores provided by ACP-TP over weighs. Thus, ACP-TP has the lowest total system power consumption.

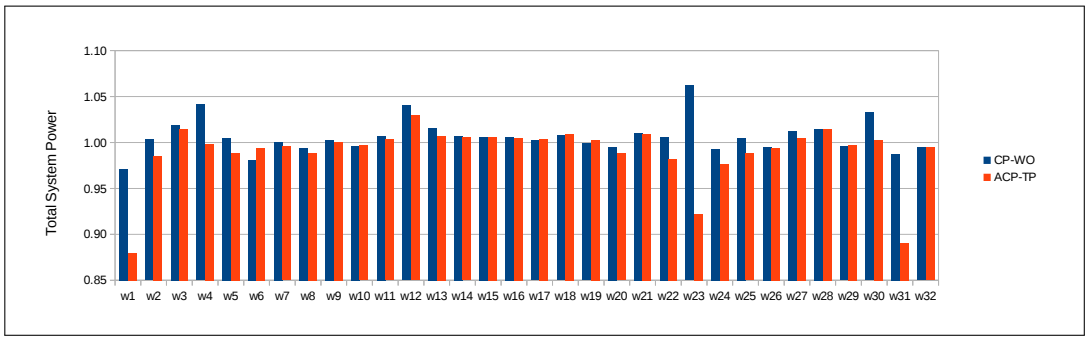

Figure 5.17: Comparison of total system power of CP-WO and ACP-TP schedulers for the given workloads. The total system power is normalized with respect to FR-FCFS.

Figure 5.18 shows the sum of total memory system power for the given workloads. It also shows the impact of adaptive compute-phase prediction (i.e., ACP) and adaptive thread prioritization (i.e., ATP) on power consumption, separately. Notice that, the combination of these two (i.e., ACP-TP) has lower memory power consumption.

Figure 5.19 shows the sum of total system power for the given workloads. It also shows the impact of adaptive compute-phase prediction (i.e., ACP) and adaptive thread prioritization (i.e., ATP), separately. Note that, the combination of these two (i.e., ACP-TP) has the lowest sum of total system power, as well.

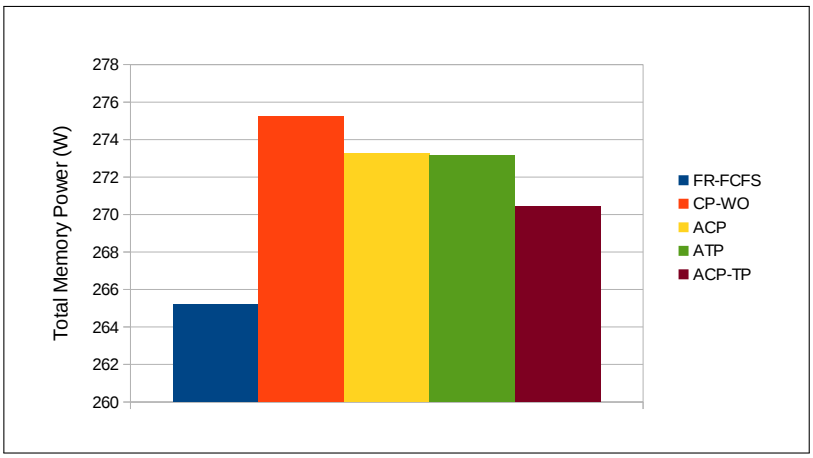

Figure 5.18: Sum of total memory system power for the given workloads.

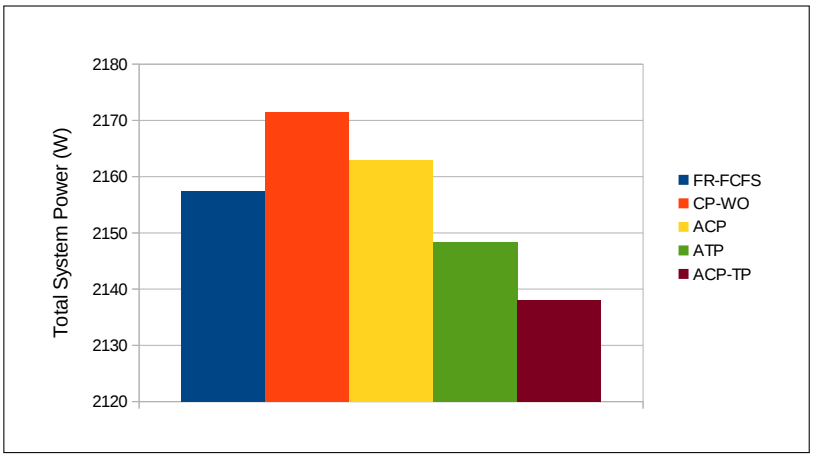

Figure 5.19: Sum of total system power consumption for the given workloads.

Chapter 6

Conclusions and Future Work

To provide higher throughput and increased performance without bumping into physical limits of Moore's Law, novel multiprocessor architectures have emerged, including chip multiprocessors that contains multiple cores on a single chip [2]. Another way to provide higher throughput and increased performance is to run more than one thread on each core with multithreading, namely simultaneous multithreading [3]. The choice of threads to be scheduled on the same core has signicant impact on overall system performance. Inter-thread contention occurs since coscheduled threads are competing for shared resources. The primary shared resource that influence the performance is the cache. An efficient scheduling should minimize the contention for shared caches to maximize utilization and system performance. Since the execution characteristics of threads varies over time, the scheduling decision has to be remade based on provisioned behaviors of threads for the near future.

The frequency and power walls have forced chip manufacturers to change their design philosophy from uniprocessors to chip multiprocessors. While multicore architectures provide higher aggregated throughput, the underlying memory subsystem remains a performance bottleneck. The memory subsystems operate in lower frequencies and they have to serve to multiple threads running on different cores simultaneously. This creates a contention on memory subsystem and has a significant impact on the overall system performance. Traditional memory access

scheduling algorithms designed for uniprocessors are inadequate for chip multiprocessors. For this reason, an efficient memory access scheduler is required to exploit the performance promises of chip multiprocessors.

To address these challenges, first, we propose a novel adaptive cache-hierarchy-aware thread scheduling algorithm that minimizes the number of accesses to the lower levels of cache/memory hierarchy and reduces the number of evictions due to contention. The adaptive cache-hierarchy-aware scheduling provides higher system throughput and improved performance.

We introduce and use a fine-grained, multi-metric scoring scheme to classify threads with respect to their execution characteristics in the proposed scheduling algorithm. The metrics used in scoring are obtained from L1 cache, as opposed to LLC as has been done in most of the previous studies.

We observe that our adaptive cache-hierarchy-aware scheduler improves the performance (i.e., instruction per cycle) of the benchmarks used in this work by up to 12.6% and an average of 7.3% over the static schedules.

The cache partitioning techniques and replacement policies to improve LLC performance are orthogonal to our approach, so they can be used along with our adaptive cache-hierarchy-aware scheduling scheme. We believe that integration of partitioning techniques with our adaptive cache-hierarchy-aware scheduler will provide even higher performance. Similarly, employing efficient replacement policies will result in with reduced number of evictions and misses, thus will improve the performance even further. As a future work, we will integrate cache partitioning and replacement policies with our adaptive cache-hierarchy-aware scheduler and evaluate the impact on system performance.

In addition to the multi-metric scoring scheme, the ability to predict/detect the regions of the cache that are used by threads can be helpful to minimize inter-thread conflicts. Such an ability will improve the performance even further. We left these enhancements as future work.

Second, we introduce a memory access scheduling algorithm that is based on

the state-of-the-art Compute-phase Prediction with Writeback-Refresh Overlap (CP-WO) scheduler proposed by Ishii et al. [34]. We improved thread prioritization scheme of Compute-phase Prediction with Writeback-Refresh Overlap scheduler. Instead of prioritizing threads based solely on their execution phases, our prioritization scheme allows to obtain fine-grained prioritization that is based on their potential to make progress in their execution. Since the properties and priorities of threads change over time, we call this approach as *adaptive thread prioritization*.

In addition to that, we enhanced compute-phase prediction scheme of Compute-phase Prediction with Writeback-Refresh Overlap scheduler in a way that it tolerates short distortions and bursts to prevent inaccurate predictions. The predefined thresholds for saturation counters to predict execution phases may prevent the detection of execution phase changes in a timely manner. Inadequately defined thresholds may result certain threads to be prioritized unfairly longer while preventing others to be prioritized when they actually should be prioritized. Thus, the efficiency of phase prediction mechanism is correlated to the accuracy of thresholds used for saturation counters. We introduced a mechanism that determines the thresholds for each thread on the fly, considering the recent memory access pattern of a thread. Since the thresholds are determined on the fly, we call it *adaptive compute-phase prediction*.

Compared to the prior schedulers First-Ready First-Come First-Serve (FR-FCFS) and Compute-phase Prediction with Writeback-Refresh Overlap (CP-WO), our algorithm reduces the execution time of the generated workloads up to 23.6% and 12.9%, respectively. Our adaptive compute-phase prediction and thread prioritization algorithm also provides reasonable power consumption results compared to CP-WO and FR-FCFS.

The proposed adaptive compute-phase prediction and thread prioritization algorithm has a flaw in workloads that consists of multiple instances of a particular task, or there is a single task in the workload. For such workloads, adaptive compute-phase prediction and thread prioritization algorithm performs poorly due to unnecessary effort to prioritize memory requests of thread(s) that are the

instances of the same task. Although we believe that such cases are rare and we expect to have diversity and abundance in the real workload tasks, we plan to enhance the presented algorithm to deal with such cases. This enhancement is also left as a future work.

Bibliography

[1] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[2] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, NY, USA), pp. 2–11, ACM, 1996.

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (New York, NY, USA), pp. 392–403, ACM, 1995.

[4] 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*, (New York, NY, USA), pp. 220–229, ACM, 2008.

[5] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a CMP of multi-threaded processors," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, (Washington, DC, USA), pp. 141–141, IEEE Computer Society, 2006.

[6] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, NY, USA), pp. 234–244, ACM, 2000.

[7] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2004.

[8] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, (Washington, DC, USA), pp. 25–38, IEEE Computer Society, 2007.

[9] P. J. Denning, "The working set model for program behavior," *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[10] W. Wong and J.-L. Baer, "Modified LRU policies for improving second-level cache behavior," in *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pp. 49–60, 2000.

[11] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Transactions on Computers*, vol. 41, no. 9, pp. 1054–1068, 1992.

[12] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 167–178, IEEE Computer Society, 2006.

[13] D. Chiou, S. Devadas, L. Rudolph, B. S. Ang, D. Chiouy, D. Chiouy, L. Rudolphy, L. Rudolphy, S. Devadasy, S. Devadasy, B. S. Angz, and B. S. Angz, "Dynamic cache partitioning via columnization," in *In Proceedings of Design Automation Conference*, 2000.

[14] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs?," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (New York, NY, USA), pp. 203–212, ACM, 2010.

[15] 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*, (New York, NY, USA), pp. 47–58, ACM, 2007.

[16] A. Settle, J. Kihm, A. Janiszewski, and D. Connors, "Architectural support for enhanced smt job scheduling," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 63–73, IEEE Computer Society, 2004.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, NY, USA), pp. 45–57, ACM, 2002.

[18] 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*, (Washington, DC, USA), pp. 340–351, IEEE Computer Society, 2005.

[19] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically controlled resource allocation in smt processors," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 171–182, IEEE Computer Society, 2004.

[20] J. L. Kihm, A. W. Janiszewski, and D. A. Connors, "Predictable fine-grained cache behavior for enhanced simultaneous multithreading (smt) scheduling," in *Proceedings of International Conference on Computing, Communications and Control Technologies*, 2004.

[21] K. Tian, Y. Jiang, and X. Shen, "A study on optimally co-scheduling jobs of different lengths on chip multiprocessors," in *Proceedings of the 6th ACM Conference on Computing Frontiers*, (New York, NY, USA), pp. 41–50, ACM, 2009.

[22] Y. Jiang, K. Tian, and X. Shen, "Combining locality analysis with online proactive job co-scheduling in chip multiprocessors," in *Proceedings of the*

5th International Conference on High Performance Embedded Architectures and Compilers, (Berlin, Heidelberg), pp. 201–215, Springer-Verlag, 2010.

[23] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (New York, NY, USA), pp. 245–257, ACM, 2003.

[24] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), pp. 117–128, IEEE Computer Society, 2002.

[25] R. A. Sugumar and S. G. Abraham, "Set-associative cache simulation using generalized binomial trees," *ACM Transactions on Computer Systems*, vol. 13, no. 1, pp. 32–56, 1995.

[26] M. DeVuyst, R. Kumar, and D. M. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, (Washington, DC, USA), pp. 140–149, IEEE Computer Society, 2006.

[27] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (New York, NY, USA), pp. 128–138, 2000.

[28] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 146–160, IEEE Computer Society, 2007.

[29] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2008.

[30] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 65–76, IEEE Computer Society, 2010.

[31] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 39–50, IEEE Computer Society, 2008.

[32] I. Hur and C. Lin, "A comprehensive approach to dram power management," in *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture*, pp. 305–316, Feb. 2008.

[33] J. Mukundan and J. Martínez, "Morse: Multi-objective reconfigurable self-optimizing memory scheduler," in *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture*, pp. 1–12, Feb. 2012.

[34] Y. Ishii, K. Hosokawa, M. Inaba, and K. Hiraki, "High performance memory access scheduling using compute-phase prediction and writeback-refresh overlap," in *Proceedings of 3rd JILP Workshop on Computer Architecture Competitions*, (Portland, OR, USA), 2012.

[35] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 375–384, IEEE Computer Society, 2010.

[36] R. Kumar and D. M. Tullsen, "Compiling for instruction cache performance on a multithreaded architecture," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, (Los Alamitos, CA, USA), pp. 419–429, IEEE Computer Society Press, 2002.

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

[38] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 866–880, 1999.

[39] S. S. Parekh, S. J. Eggers, and H. M. Levy, "Thread-sensitive scheduling for smt processors," tech. rep., University of Washington, 2001.

[40] J. R. Bulpin and I. A. Pratt, "Hyper-threading aware process scheduling heuristics," in *Proceedings of USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2005.

[41] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, H., N. Udipi, Aniruddha, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the utah simulated memory module," tech. rep., University of Utah, UUCS-12-002, 2012.

[42] R. Ubal, J. Sahuquillo, S. Petit, and P. López, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, Oct. 2007.

[43] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Jan. 2011.

[44] 3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC). http://www.cs.utah.edu/ rajeev/jwac12/, Jun. 2012.

Appendix A

Extended Evaluations for Adaptive Cache-Hierarchy-Aware Thread Scheduling

In this appendix, we provide extented evaluations for adaptive cache-hierarchy-aware thread scheduling. The evaluations include L1 and L2 hits/misses, L1 and L2 hit ratio of different schedules and performance of benchmarks under different schedules. These evaluations give more insight regarding the execution characteristics of threads, and their friendliness for a given thread.

A.1 L1 Hits/Misses Variations

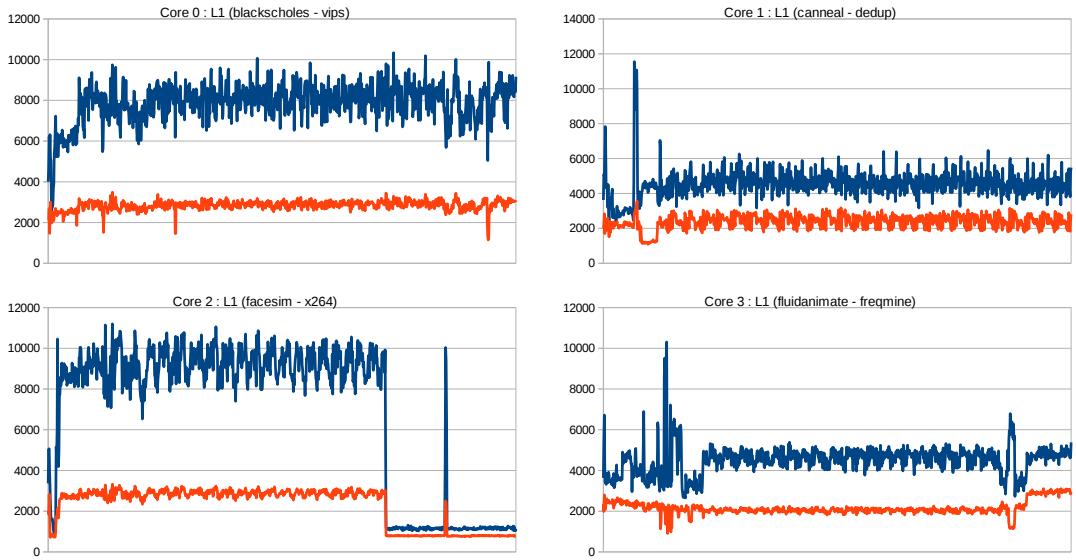

Figure A.1: L1 hits and misses under static schedule 1.

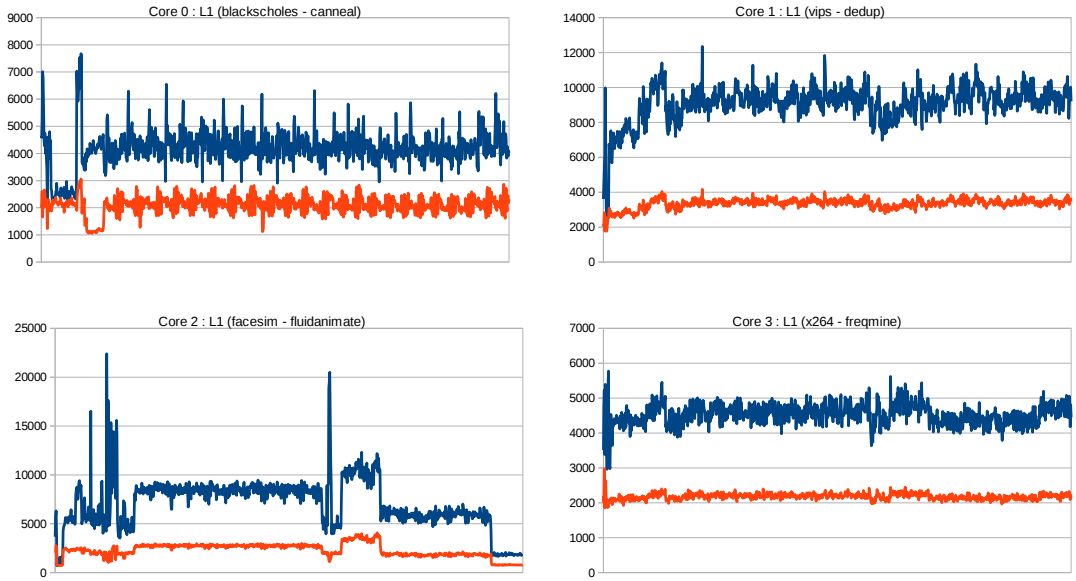

Figure A.2: L1 hits and misses under static schedule 2.

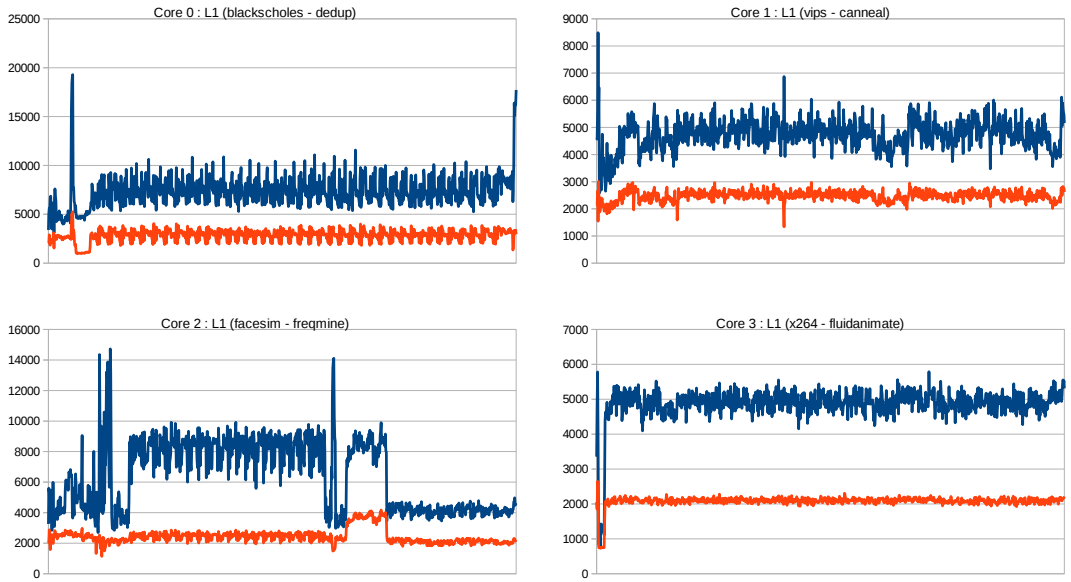

Figure A.3: L1 hits and misses under static schedule 3.

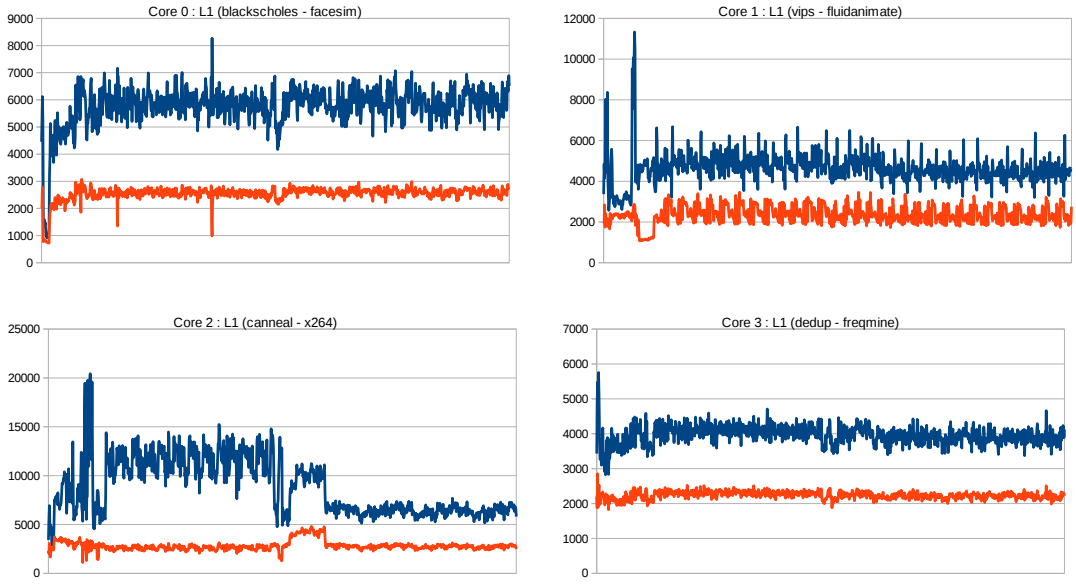

Figure A.4: L1 hits and misses under static schedule 4.

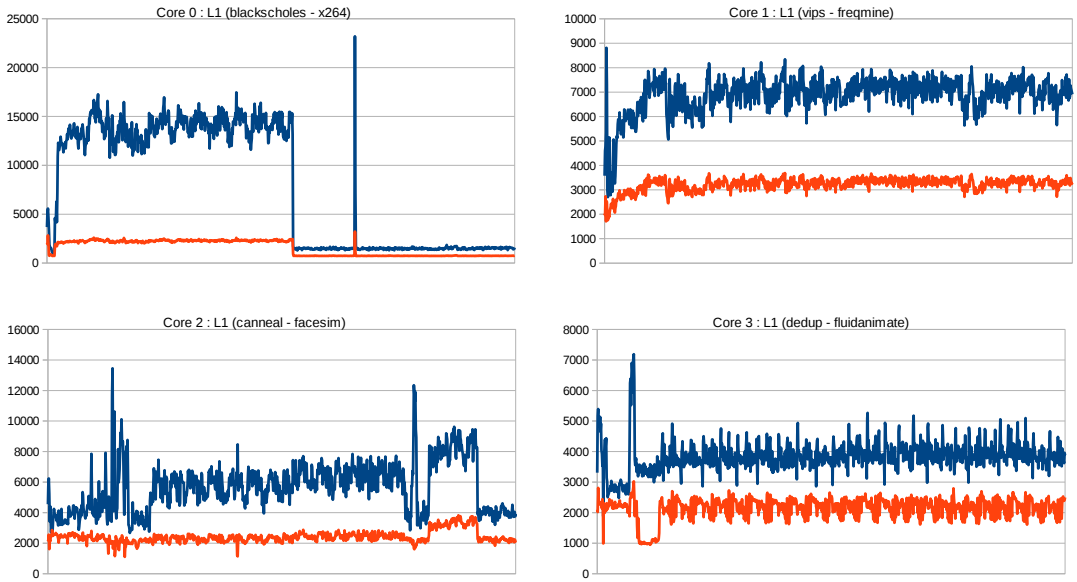

Figure A.5: L1 hits and misses under static schedule 5.

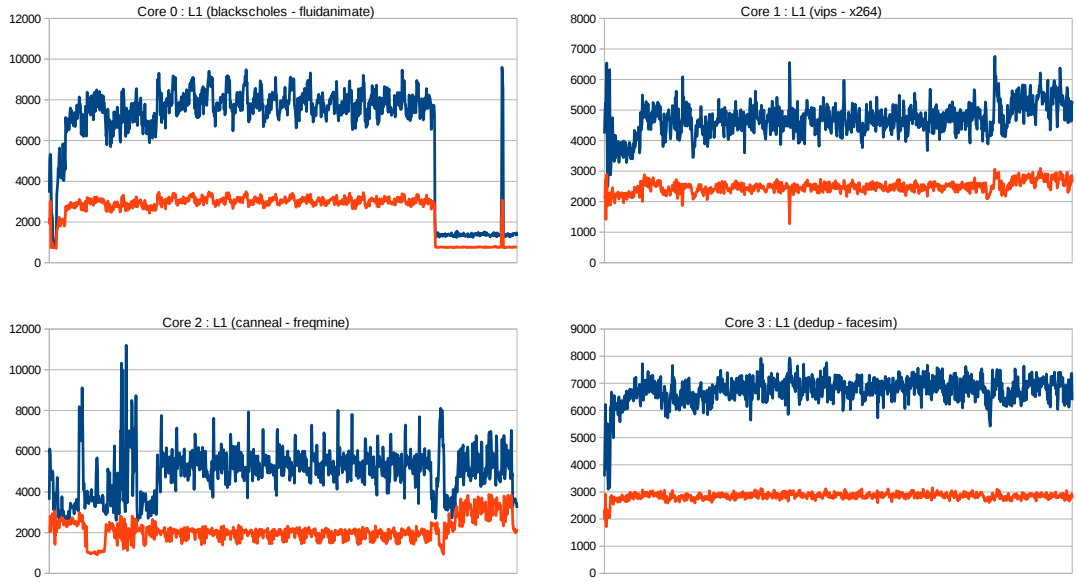

Figure A.6: L1 hits and misses under static schedule 6.

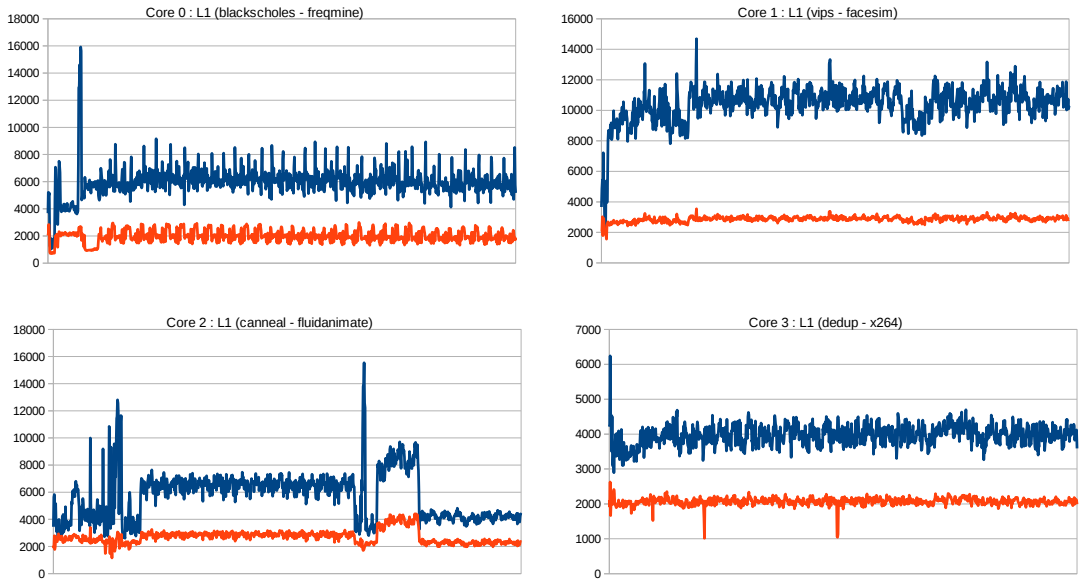

Figure A.7: L1 hits and misses under static schedule 7.

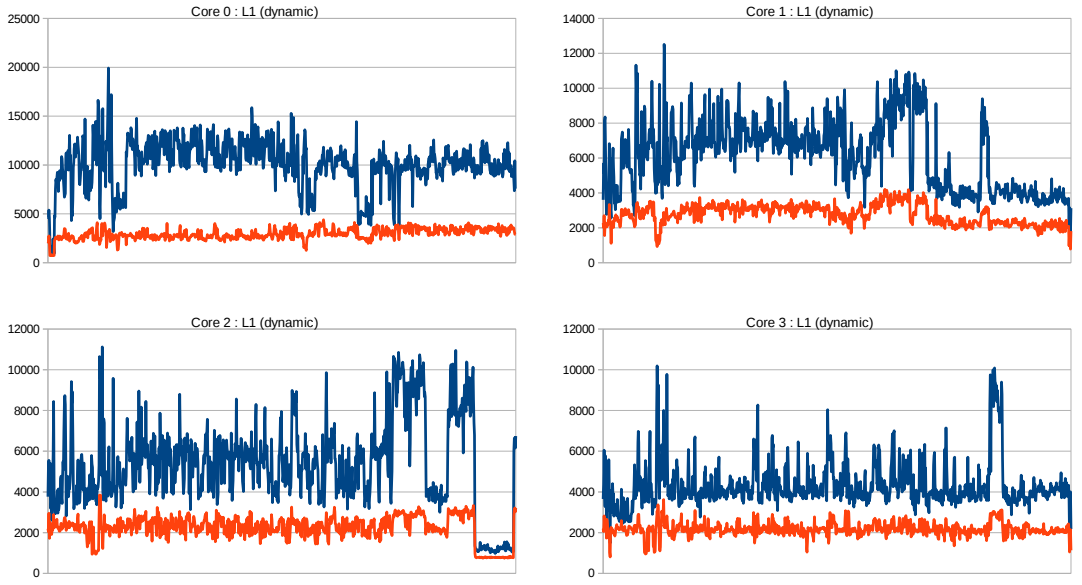

Figure A.8: L1 hits and misses under dynamic-offline schedule.

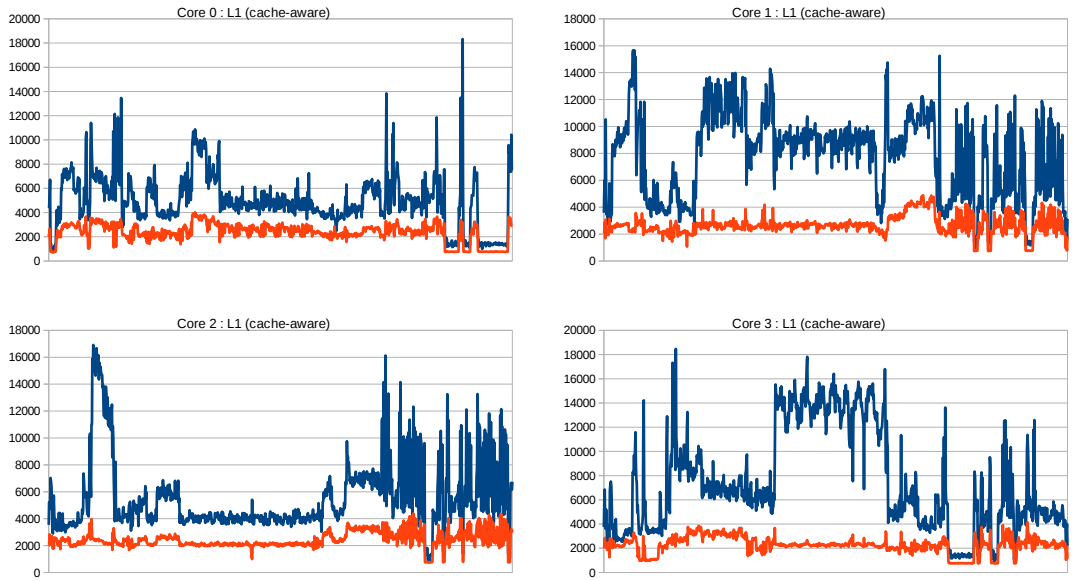

Figure A.9: L1 hits and misses under adaptive cache-hierarchy-aware scheduling.

A.2 L1 Hit Ratio Variations

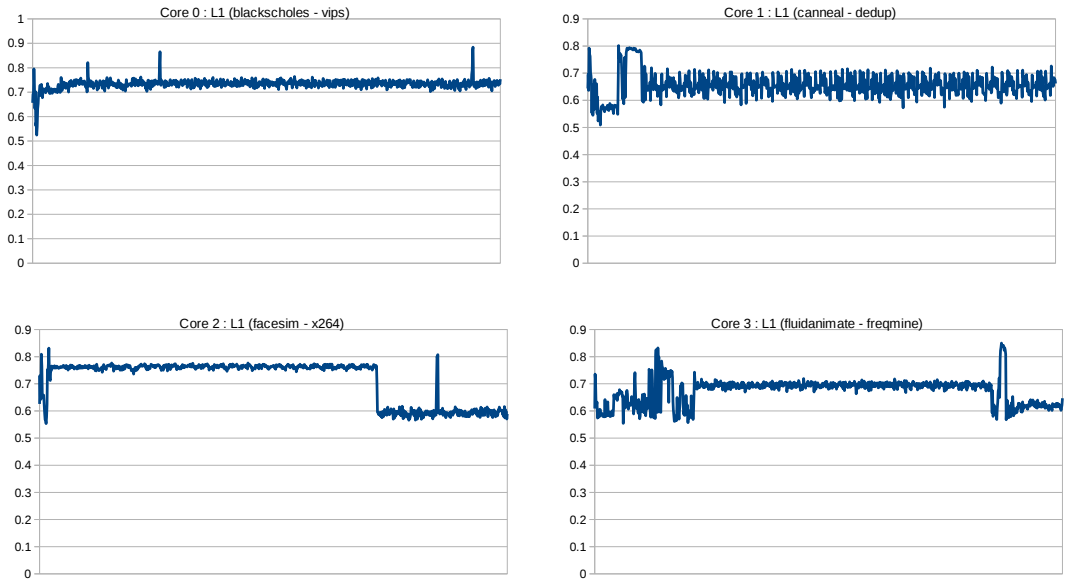

Figure A.10: L1 hit ratio under static schedule 1.

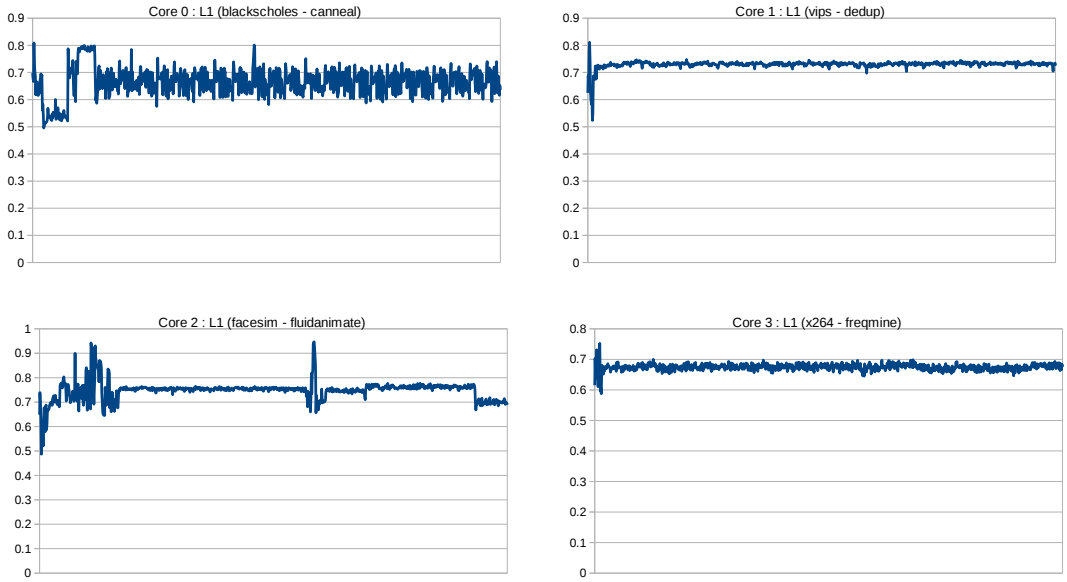

Figure A.11: L1 hit ratio under static schedule 2.

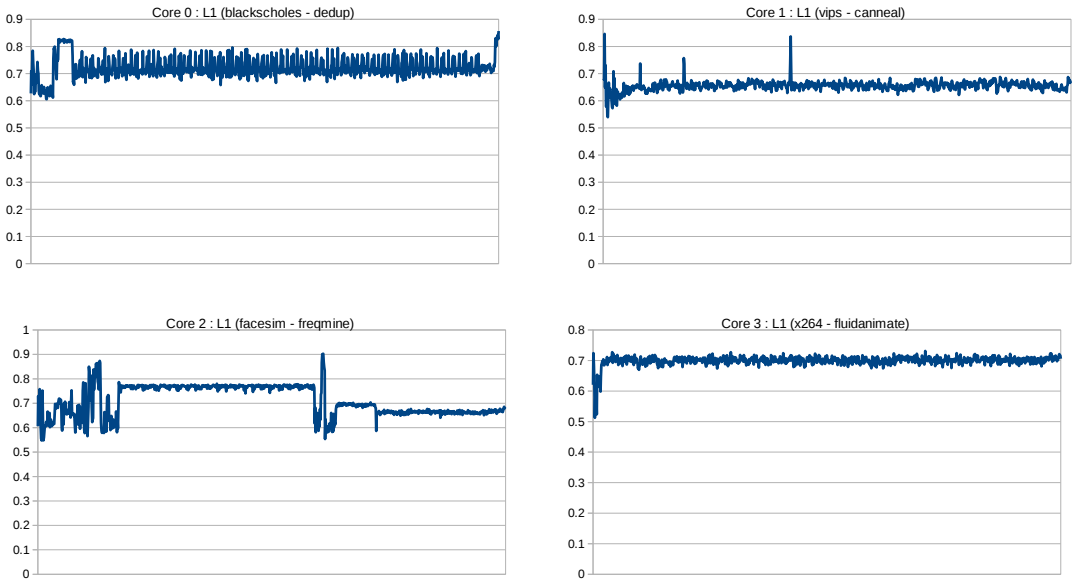

Figure A.12: L1 hit ratio under static schedule 3.

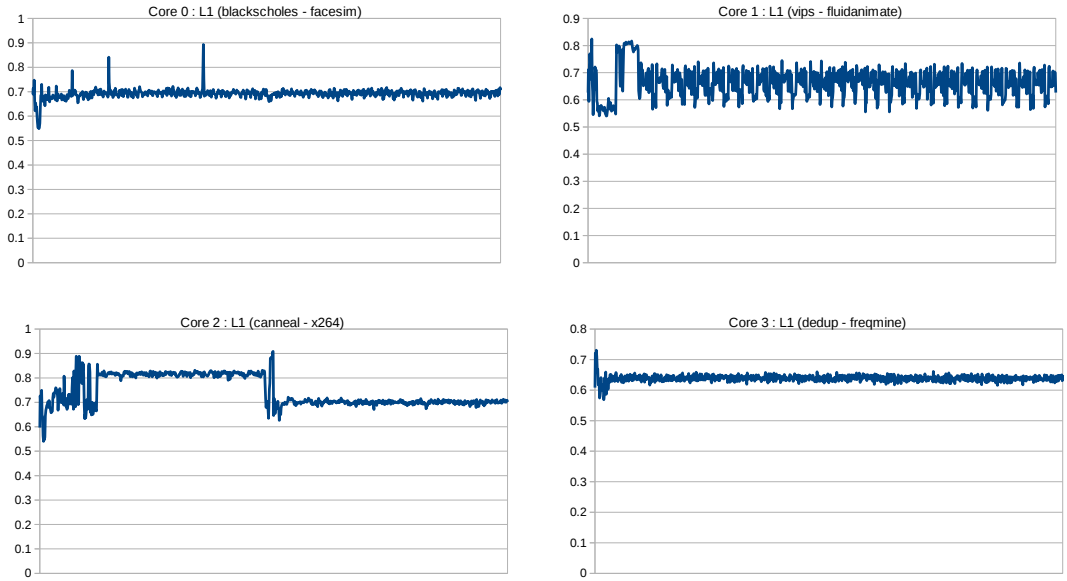

Figure A.13: L1 hit ratio under static schedule 4.

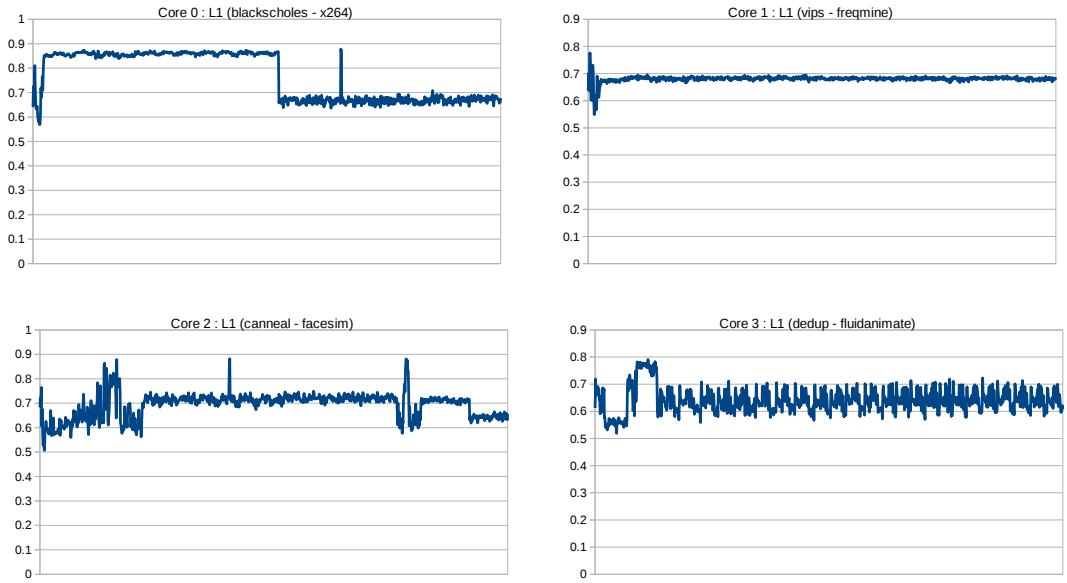

Figure A.14: L1 hit ratio under static schedule 5.

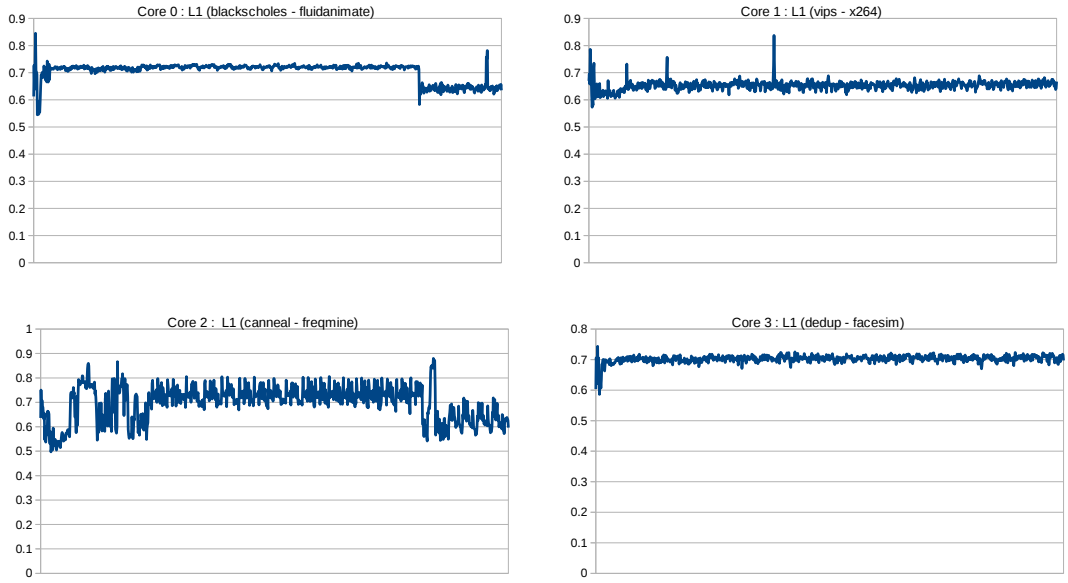

Figure A.15: L1 hit ratio under static schedule 6.

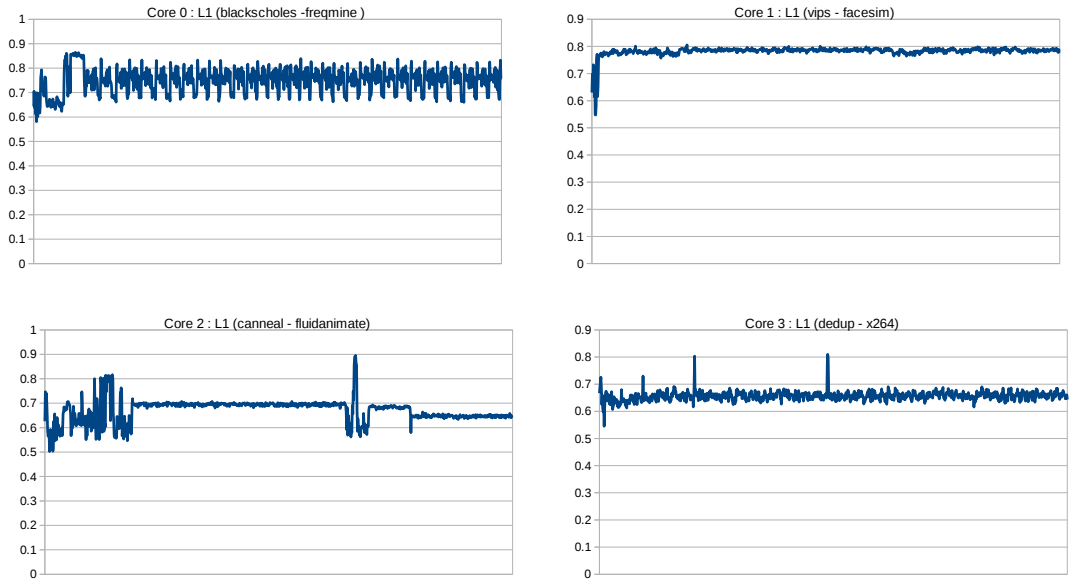

Figure A.16: L1 hit ratio under static schedule 7.

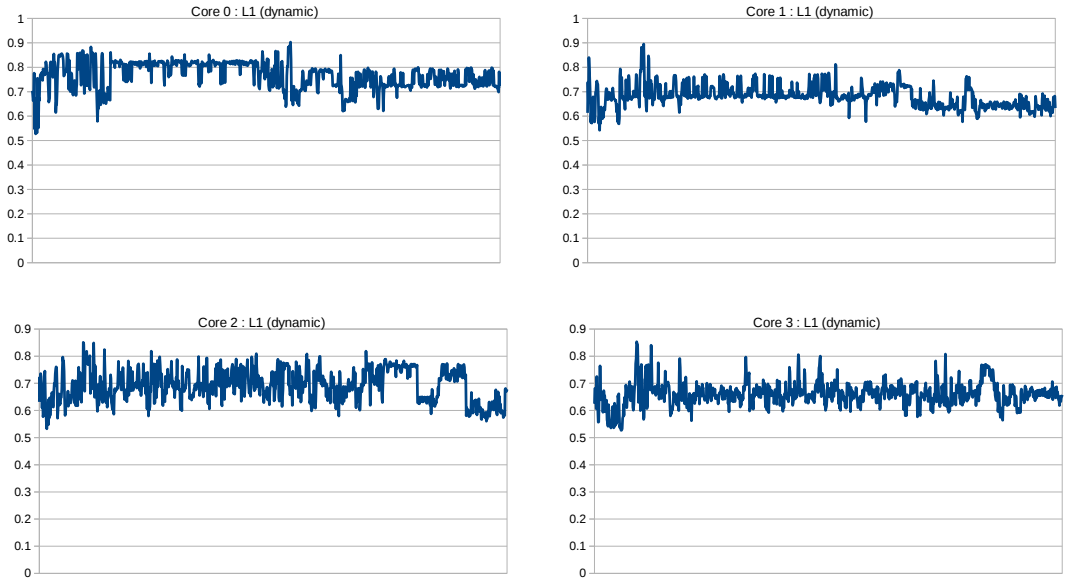

Figure A.17: L1 hit ratio under dynamic-offline schedule.

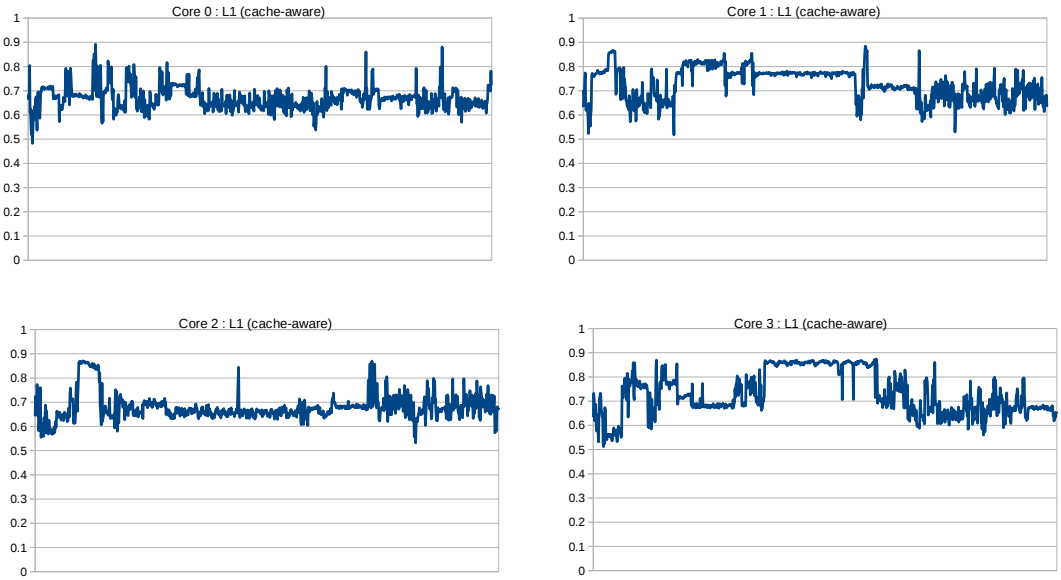

Figure A.18: L1 hit ratio under adaptive cache-hierarchy-aware scheduling.

A.3 L2 Hits/Misses Variations

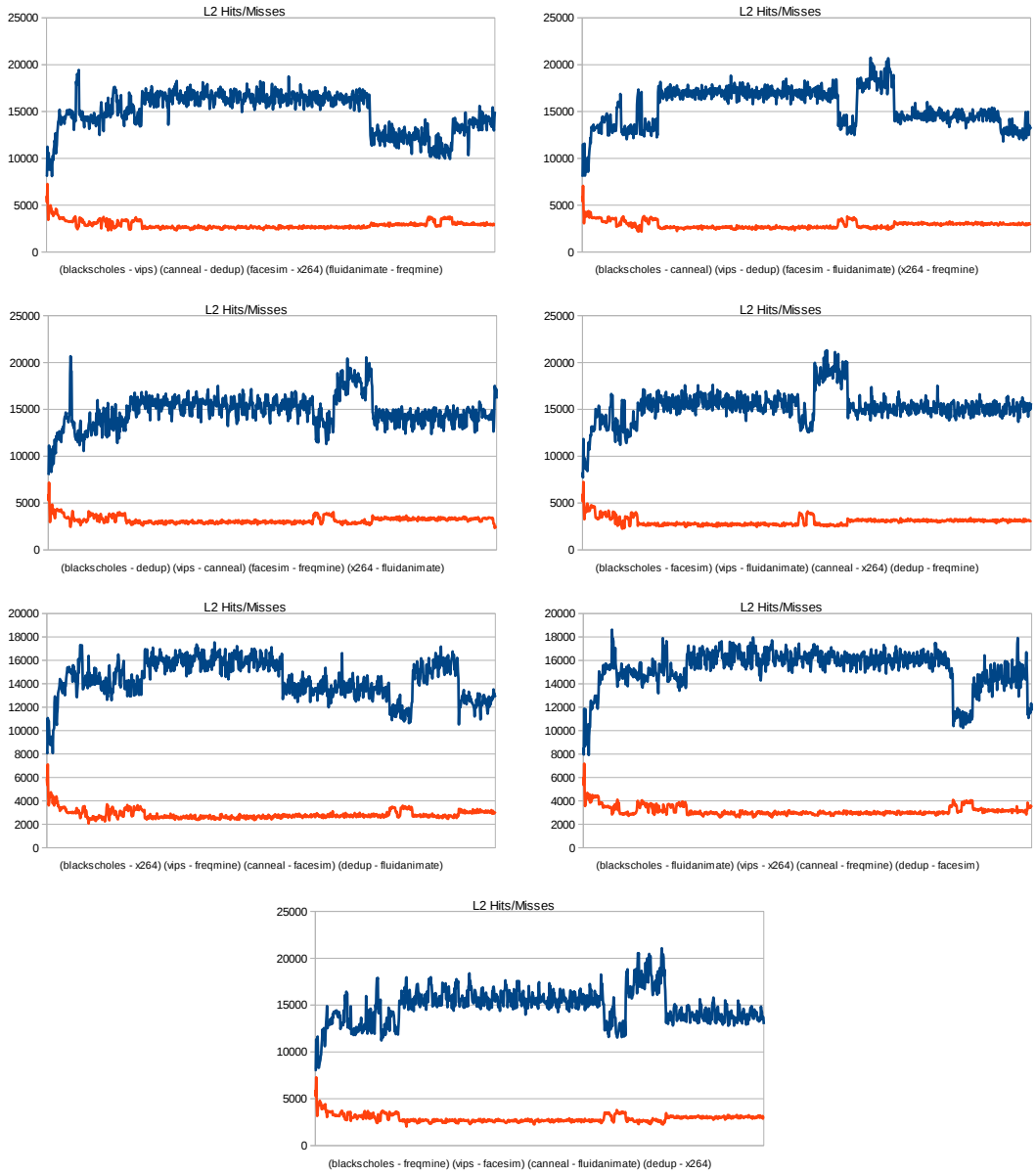

Figure A.19: L2 hits and misses under static scheduling schemes.

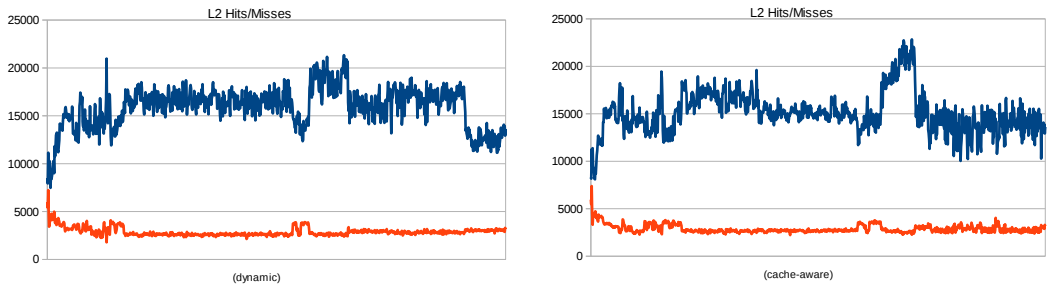

Figure A.20: L2 hits and misses under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

A.4 L2 Hit Ratio Variations

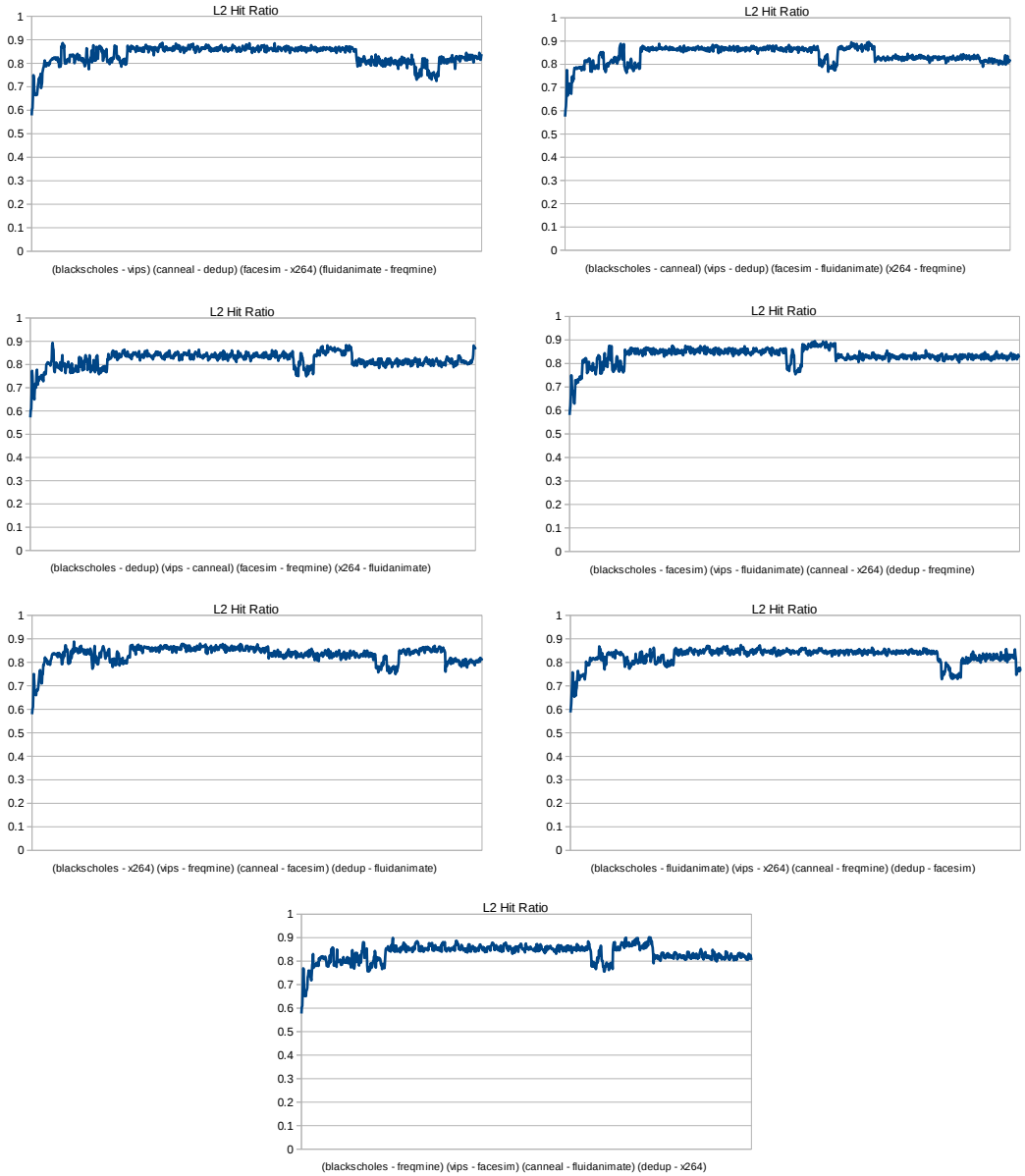

Figure A.21: L2 hit ratio under static scheduling schemes.

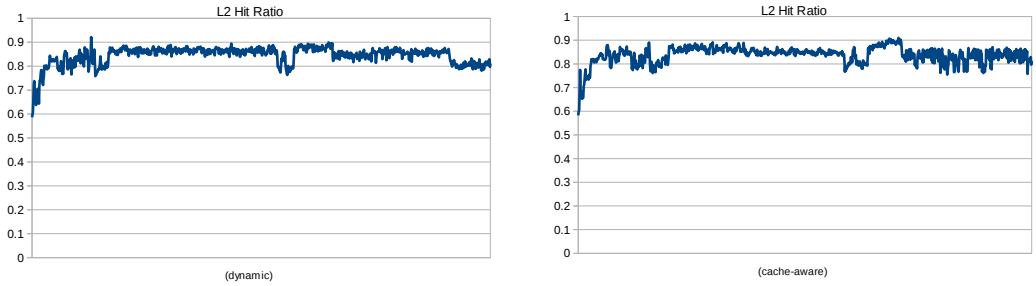

Figure A.22: L2 hit ratio under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

A.5 Performance Variations of Benchmarks

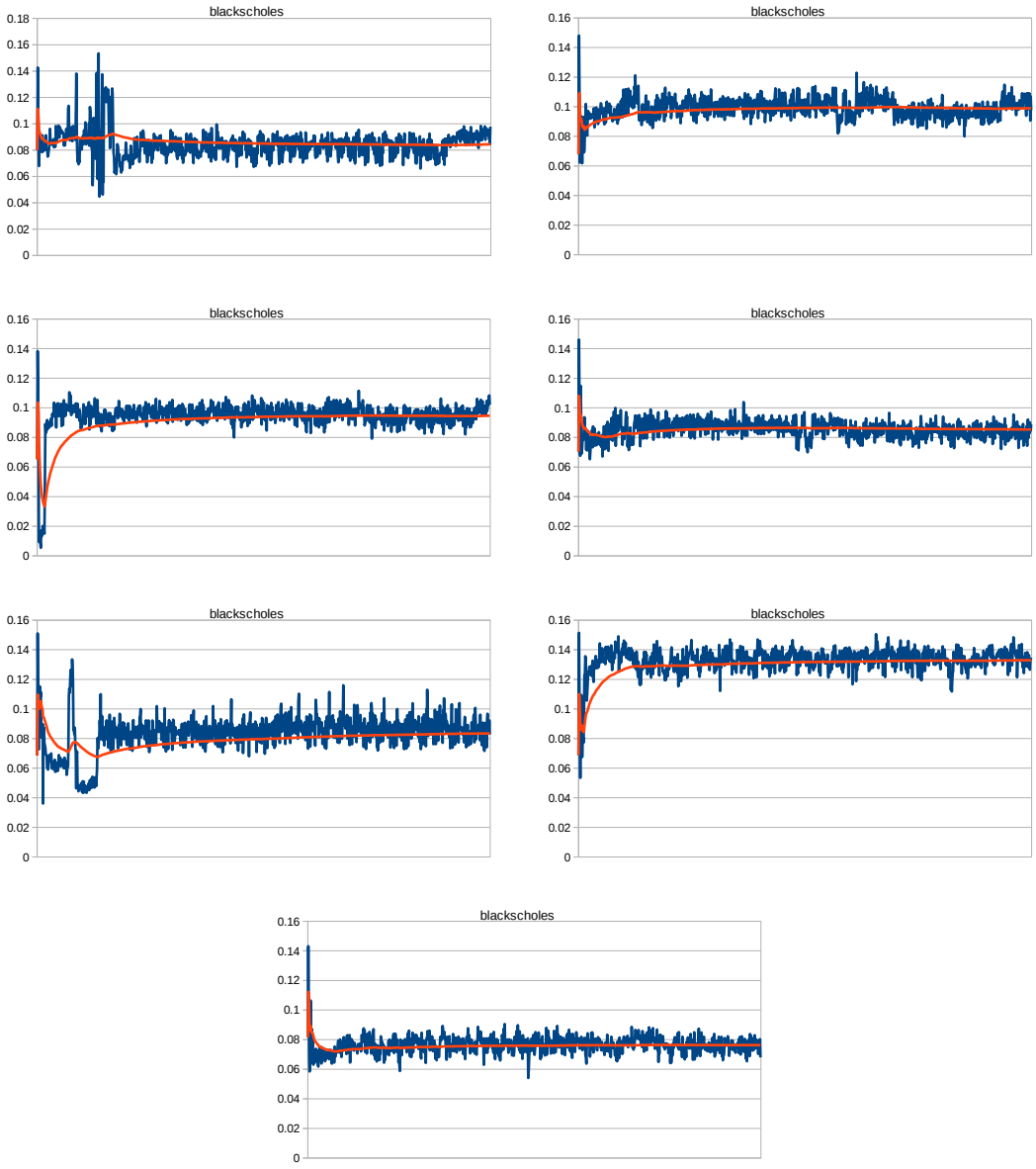

Figure A.23: IPC of *blackscholes* under static scheduling schemes.

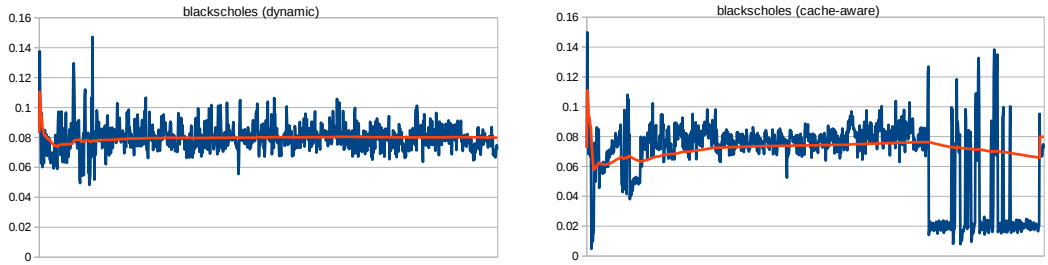

Figure A.24: IPC of *blackscholes* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

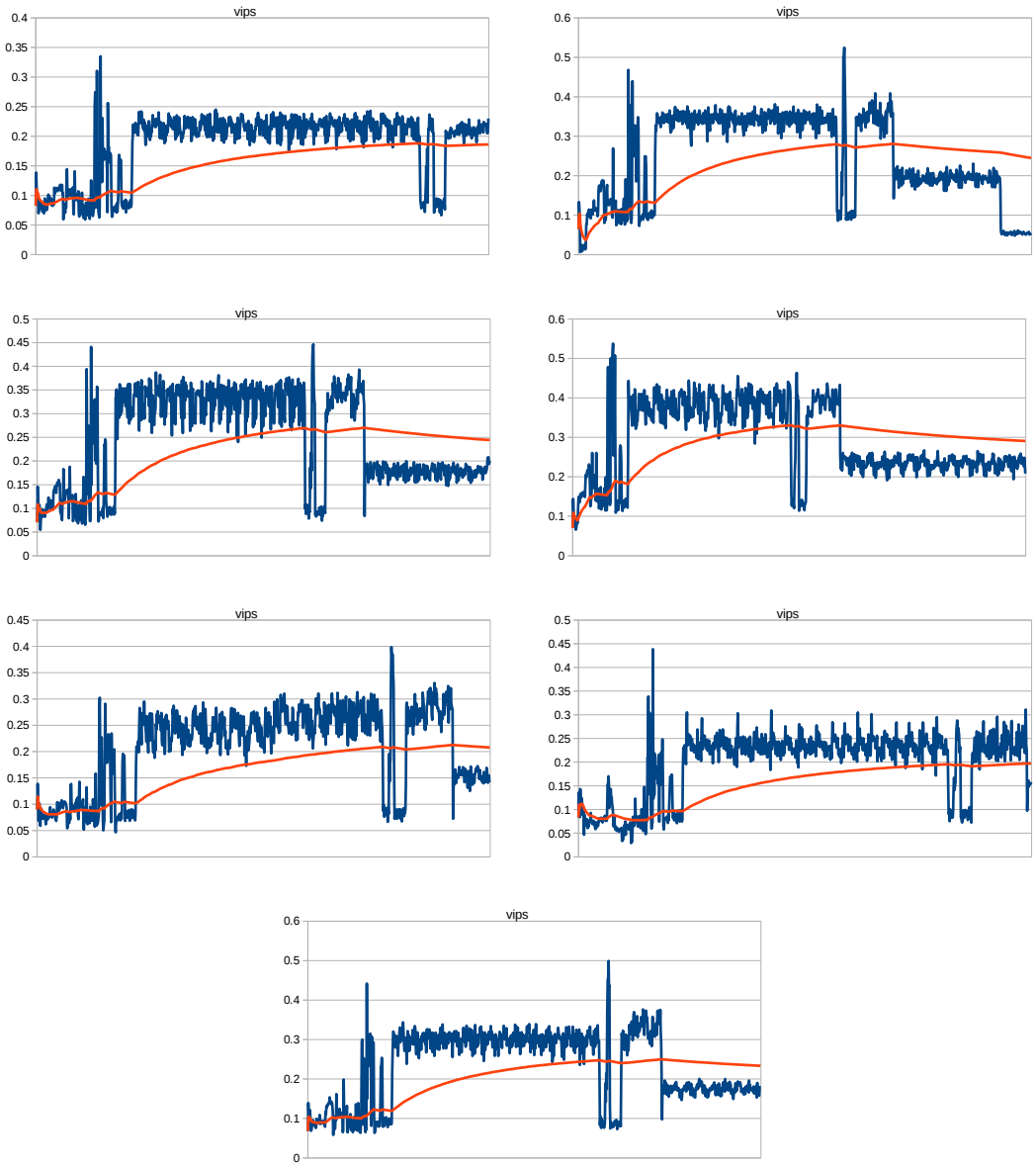

Figure A.25: IPC of *vips* under static scheduling schemes.

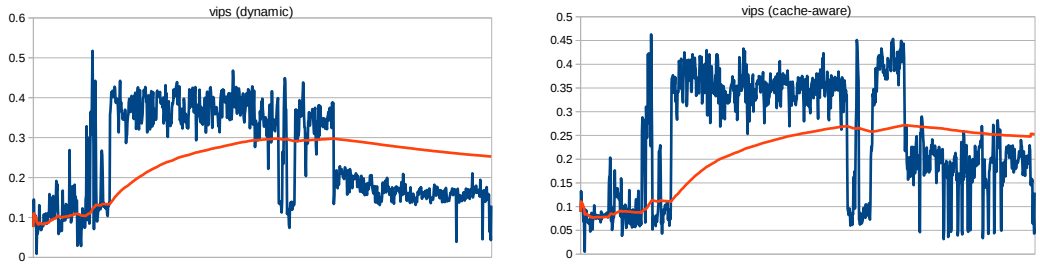

Figure A.26: IPC of *vips* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

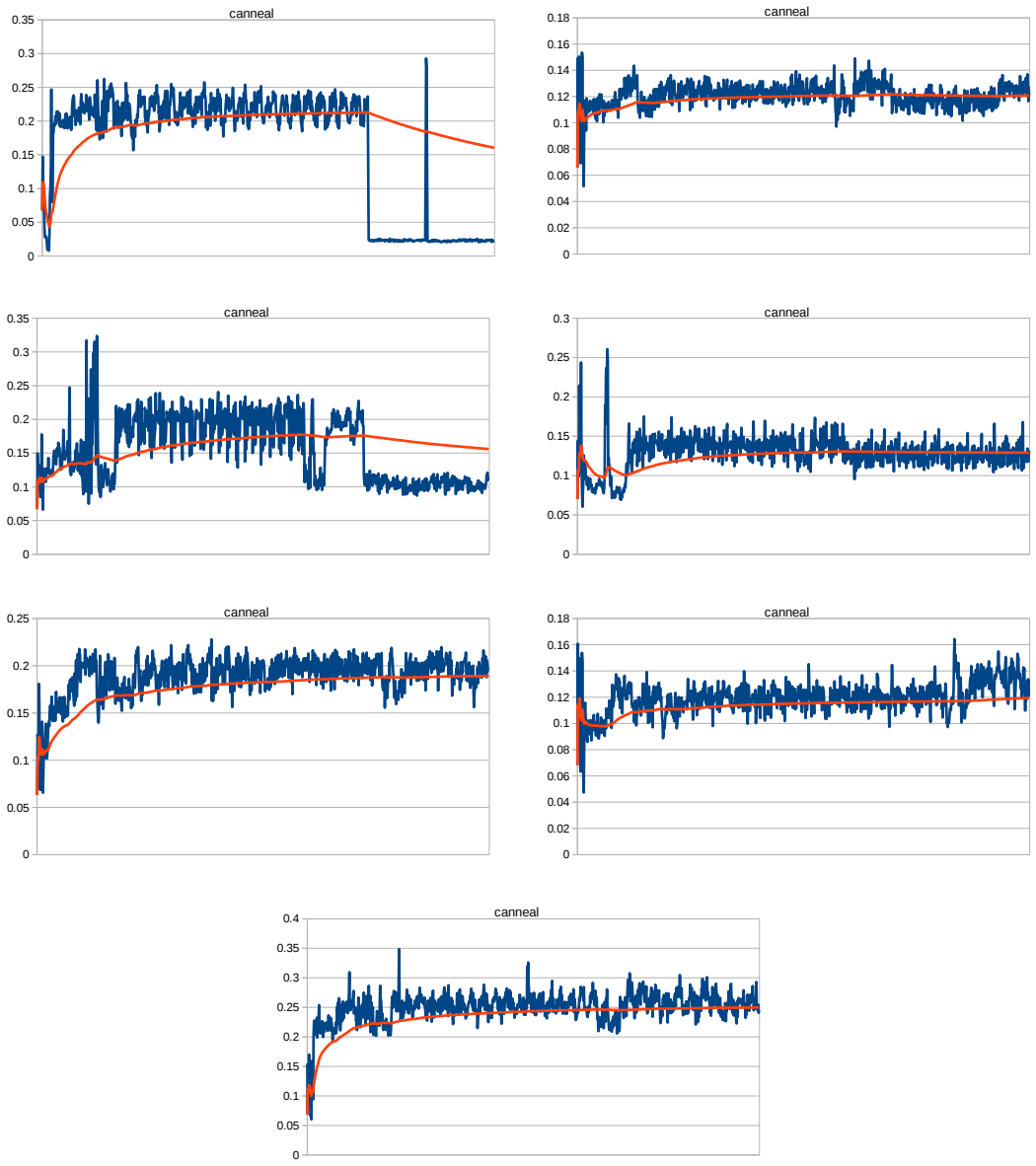

Figure A.27: IPC of *canneal* under static scheduling schemes.

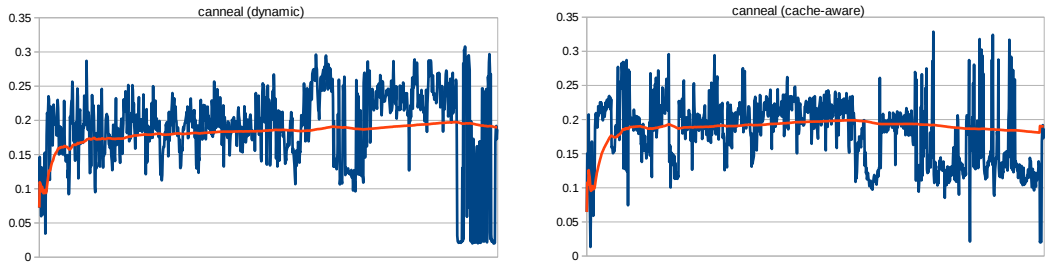

Figure A.28: IPC of *canneal* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

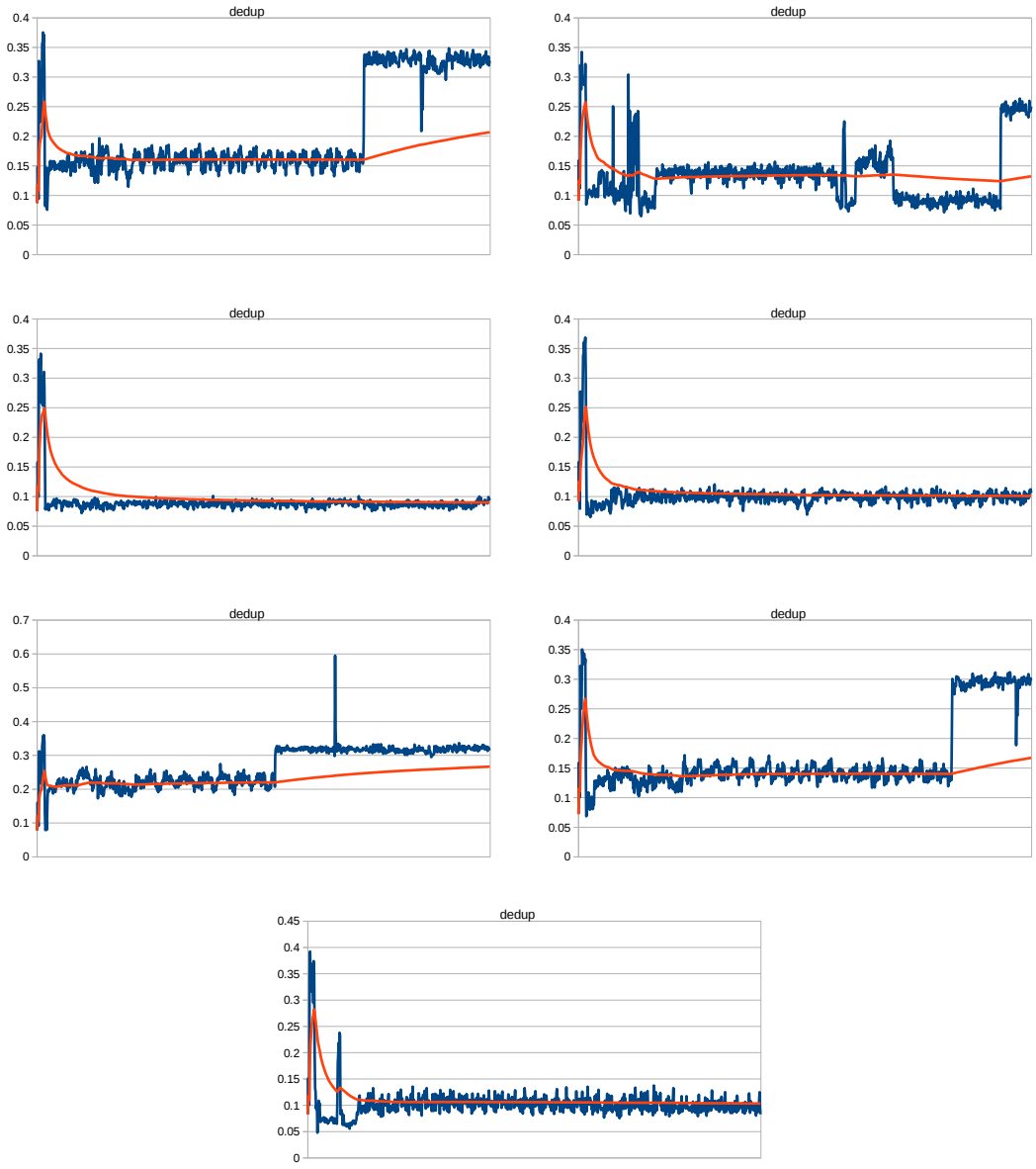

Figure A.29: IPC of *dedup* under static scheduling schemes.

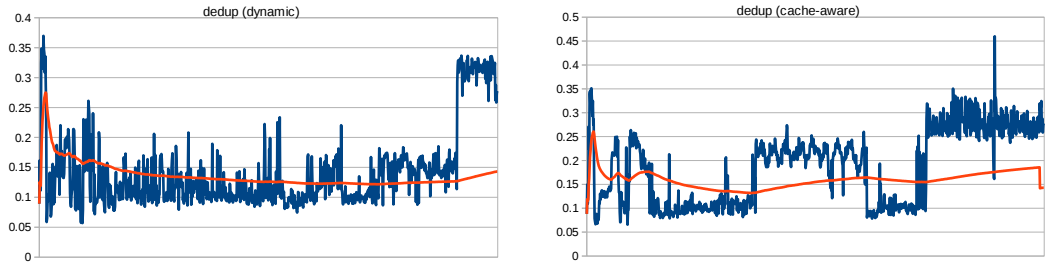

Figure A.30: IPC of *dedup* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

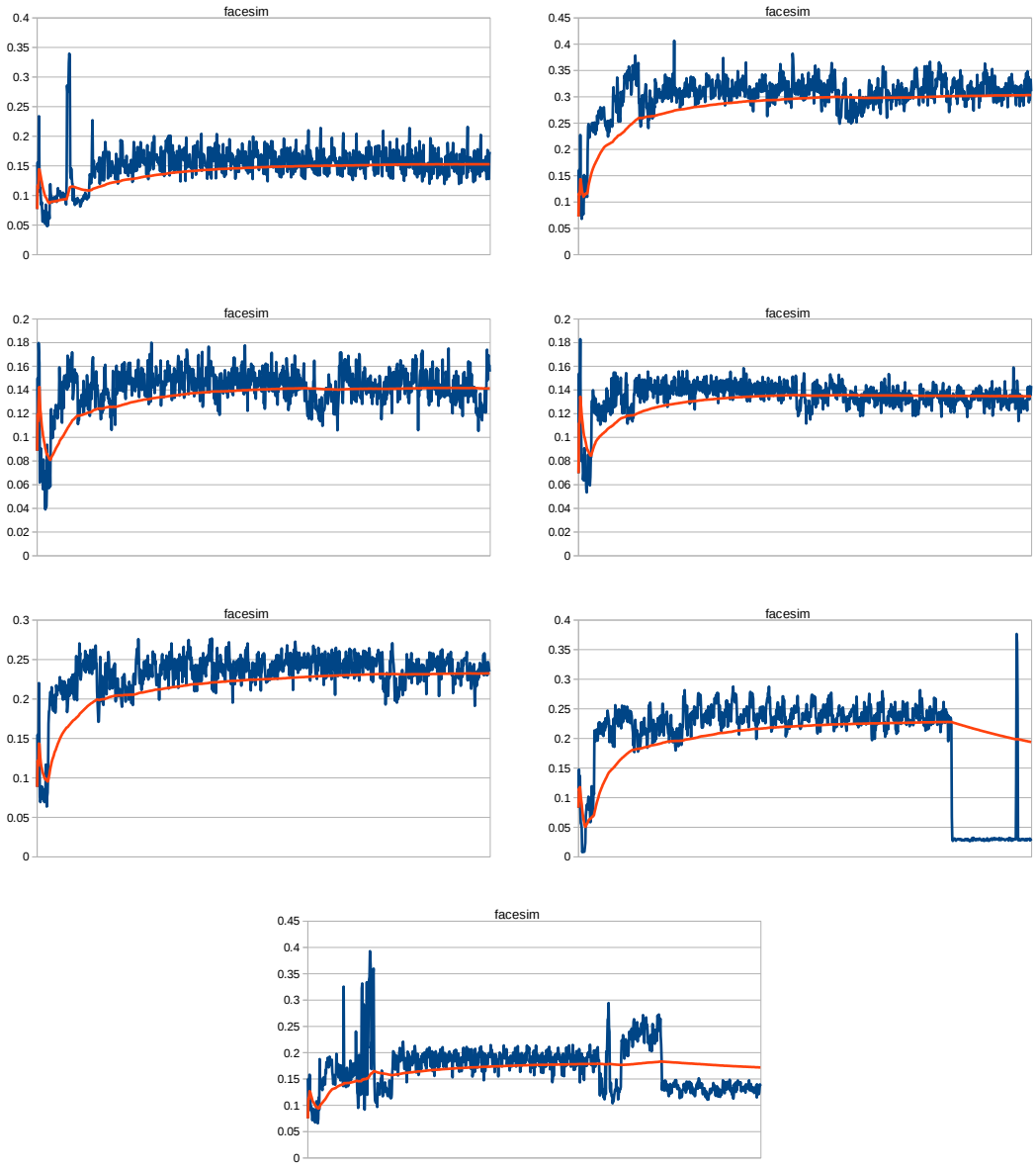

Figure A.31: IPC of *facesim* under static scheduling schemes.

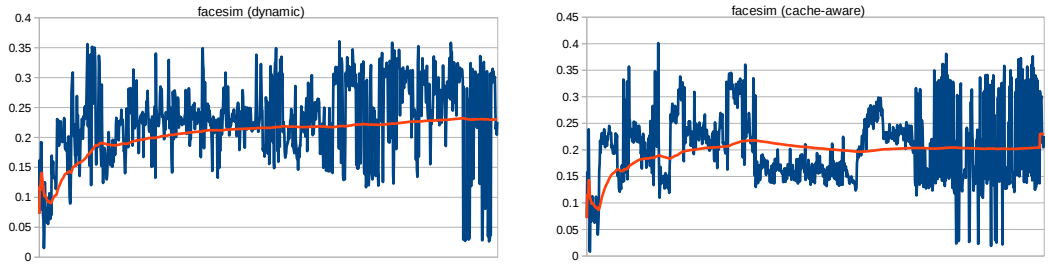

Figure A.32: IPC of *facesim* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

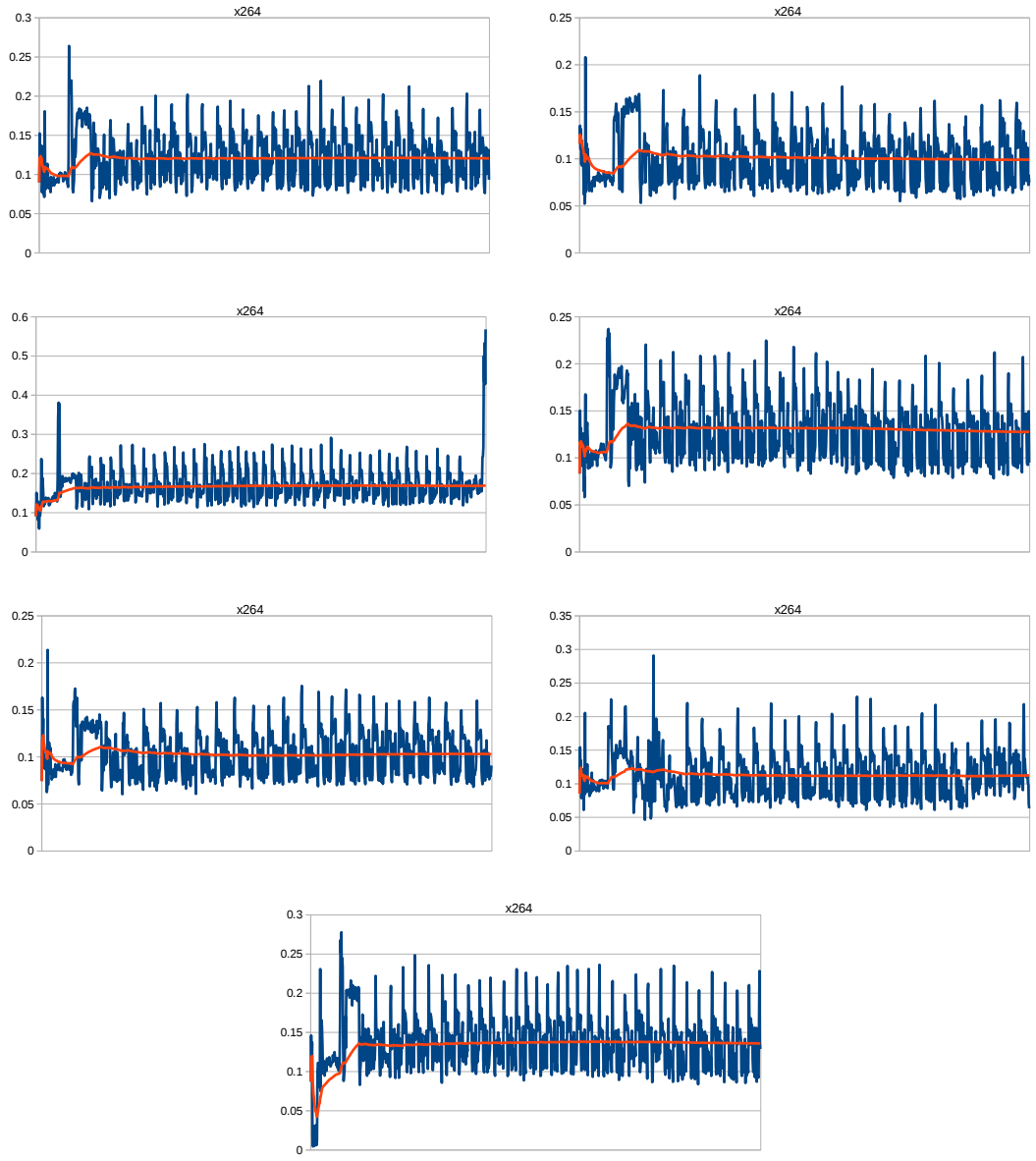

Figure A.33: IPC of *x264* under static scheduling schemes.

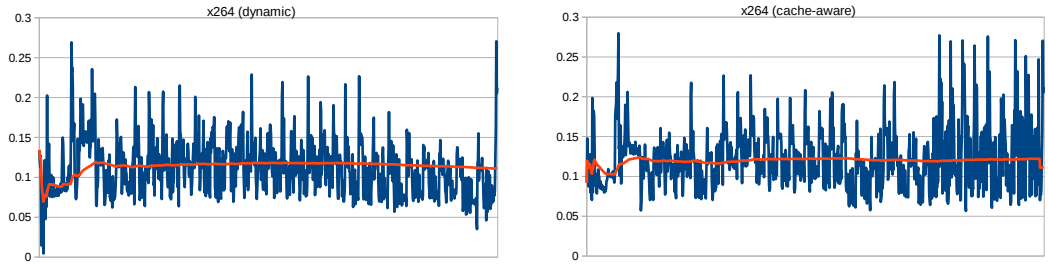

Figure A.34: IPC of *x264* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

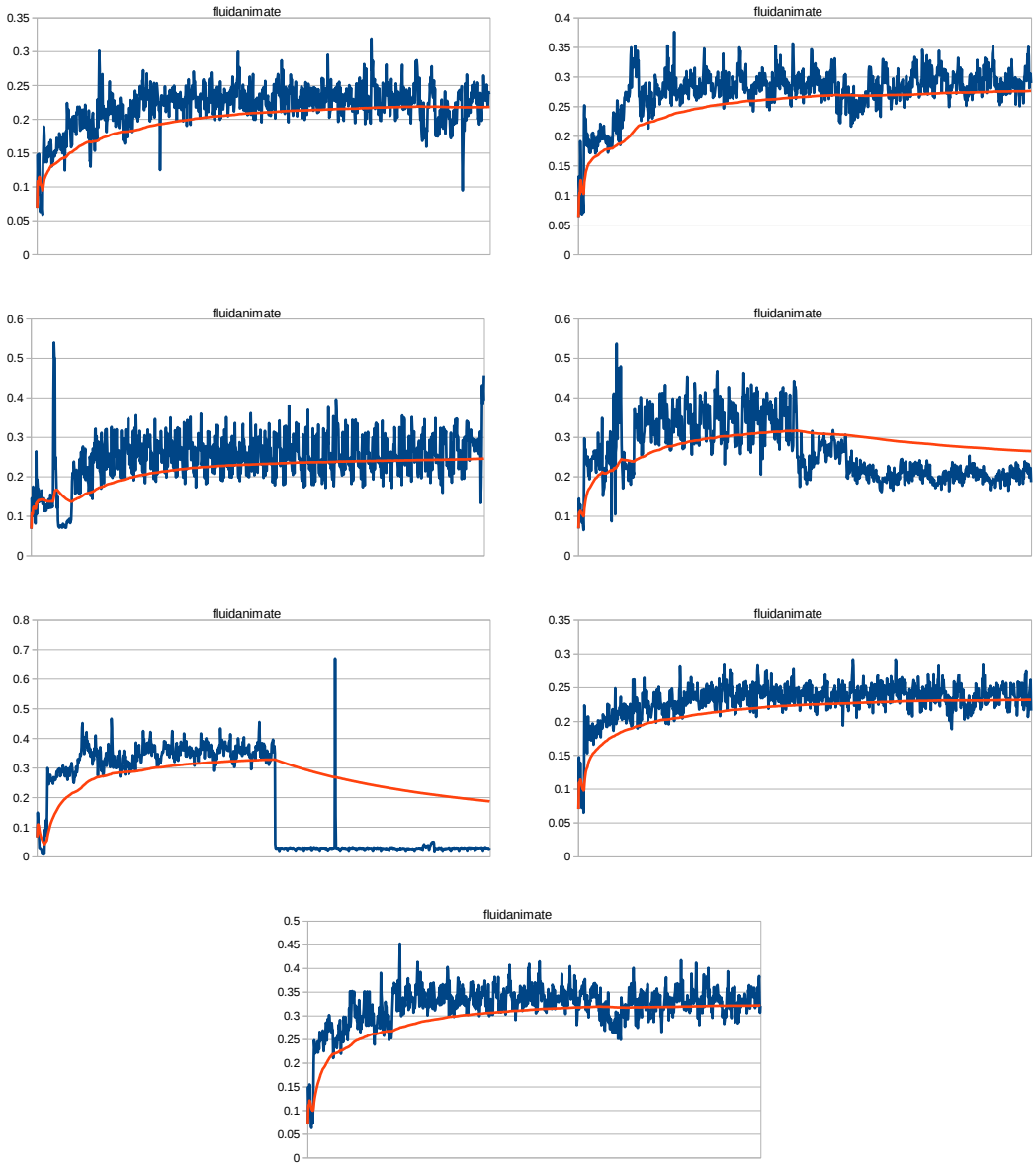

Figure A.35: IPC of *fluidanimate* under static scheduling schemes.

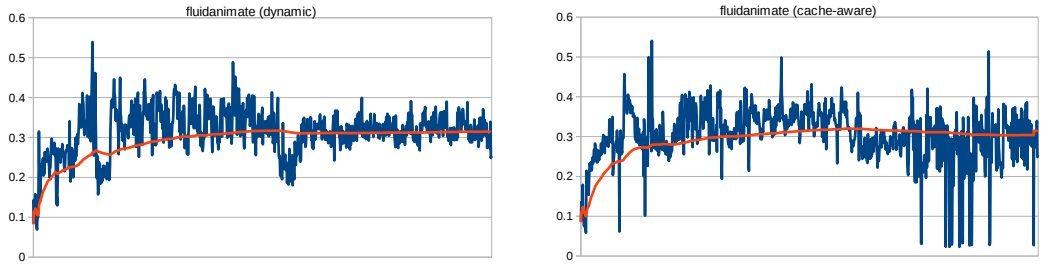

Figure A.36: IPC of *fluidanimate* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

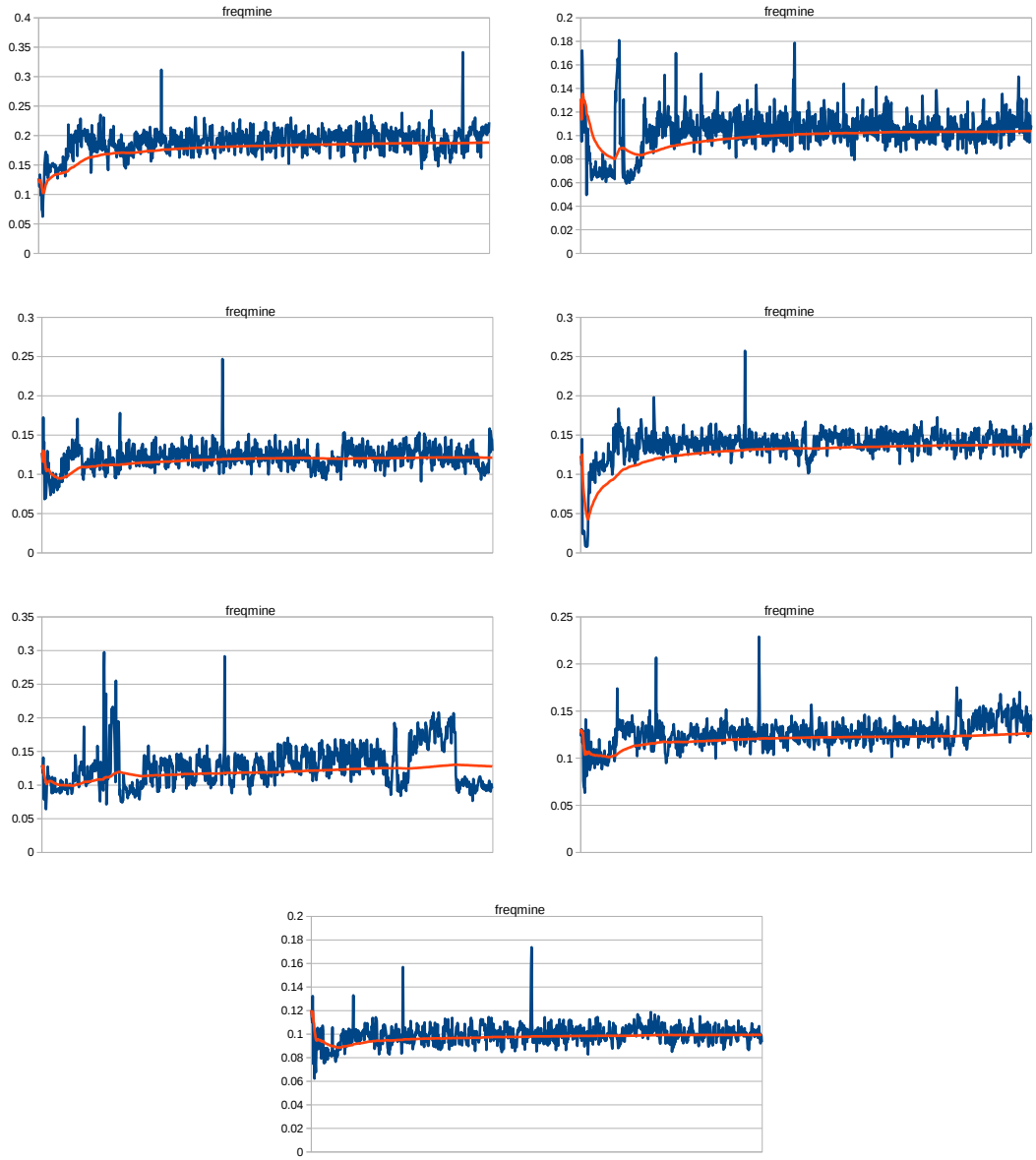

Figure A.37: IPC of *freqmine* under static scheduling schemes.

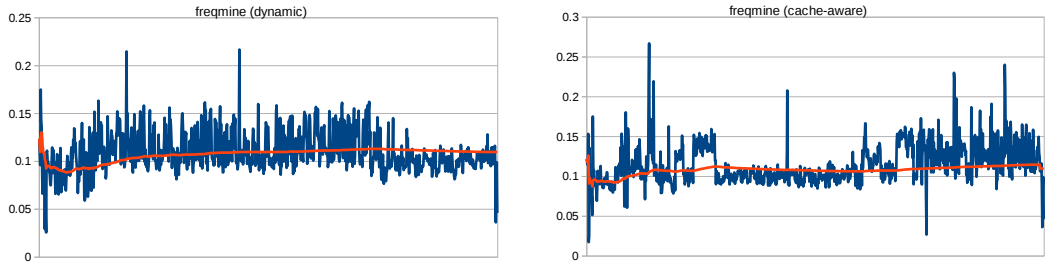

Figure A.38: IPC of *freqmine* under dynamic offline and adaptive cache-hierarchy-aware scheduling schemes.

Vita

Ismail Akturk was born in 1984, in Trabzon, Turkey. He has spent his life in Samsun until he received his high school diploma from Samsun Anatolian High School in 2002. Then, he moved to Istanbul for his college education. He received a bachelor's degree in computer engineering from Dogus University in 2007. After that, he obtained master's degree in Electrical Engineering from Louisiana State University in 2009. He worked in Center for Computation and Technology at Louisiana State University as a graduate research assistant and focused on distributed storage and data management systems. Then, he joined Bilkent University in 2010, where he pursued master's study in Computer Engineering, and worked on computer architecture and memory systems.