Performance, Power Modeling and Optimization for High-Performance Computing Systems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Chi Xu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

John Sartori

Oct, 2016

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school.

I wish to thank my committee members who were more than generous with their expertise and precious time. A special thanks to Dr. John Sartori, my advisor for his countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process.

I would like to acknowledge and thank my school division for allowing me to conduct my research and providing any assistance requested. Special thanks goes to the members of staff development and human resources department for their continued support.

Finally I would like to thank the beginning teachers, mentor-teachers and administrators in our school division that assisted me with this project. Their excitement and willingness to provide feedback made the completion of this research an enjoyable experience.

Dedication

I dedicate my dissertation work to my family and many friends. A special feeling of gratitude to my loving parents, Xinzhi Xu and Yingjie Zhao whose words of encouragement and push for tenacity ring in my ears. My cousin Fan Xu, who has never left my side and is very special.

I also dedicate this dissertation to my many friends who have supported me throughout the process. I will always appreciate all they have done, especially during those hard times in my school life.

Abstract

Heterogeneity abounds in modern high-performance computing systems. Applications are heterogeneous, containing time-varying unbalanced utilization for different resources, and system architectures have become heterogeneous in order to achieve higher levels of performance and energy efficiency. The most powerful, and also the most energy-efficient high-performance computing systems today consist of many-core CPUs and GPGPUs with a variety of specialize on-chip and off-chip memories. These heterogeneous systems provide a huge amount of computing resources, but it is becoming increasingly challenging to use them effectively and efficiently to maximize their potential. This becomes an even more pressing challenge as energy efficiency becomes the primary barrier to achieving higher levels of performance. This thesis addresses the challenges of performance modeling and optimization in heterogeneous high-performance computing systems. Effective system optimization requires understanding of how performance and power change in response to optimizations. Therefore, we begin by summarizing the impact of modern architectural advances on performance and power modeling for chip multiprocessors (CMPs). We present two models that estimate the performance and power in such systems. The first model, CAMP, is a fast and accurate cache-aware performance model that estimates the performance degradation due to cache contention of processes running on cache-sharing cores. We then propose a system-level power model for a multi-programmed CMP environment that accounts for cache contention. We explain how to integrate the two models to enable power-aware process assignment. Then, we propose an off-chip memory access-aware runtime DVFS control technique that minimizes energy consumption subject to a constraint on application execution time.

The second part of the dissertation focuses on improving performance for GPGPUs. After a thorough analysis on CPI breakdown, we lay out all the key factors that govern GPU throughput. In order to improve overall performance for GPGPUs, we propose two approaches that address the key factors, without introducing extra congestion and degradation to the system. We first propose a new two-level priority scheduling policy to improve overall performance by optimizing effective degree of parallelism. Then,

we propose ICMT, a full, detailed solution for intra-core multitasking for GPGPUs, including architectural support and a contention-aware workload scheduling algorithm that improves all the key factors in a balanced fashion. Furthermore, we propose a new contention-aware analytical performance model that provides fine-grained workload scheduling decisions for intra-core multitasking, including detailed resource allocation from co-scheduled workloads.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

High-performance computing systems are commonly seen. Typical high performance computing systems include stationary desktop computers, workstations, and servers. Often coupled with multi-core CPUs and GPUs with abundant on-chip and off-chip memory, these modern computing systems have become immensely powerful platform to handle an variety of heterogeneous applications and tasks simultaneously. However, it is no easy task to formalize and solve the problem of optimizing performance and energy consumption for such heterogeneous, potentially data-intensive, workloads running on modern high performance computing systems composed of heterogeneous complex multi-core resources (CPUs and GPUs). To properly tackle this problem, we need a set of optimizing techniques that can address the following challenges:

1. Shared resource allocation, wisely distribute resource shared among different workloads to avoid significant performance degradation relative to what they could achieve running in a contention-free environment.

2. Workload scheduling, assign workload to different resource sharing clusters to improve shared resource allocation within the resource sharing cluster.

3. Power delivery and voltage control, employ dynamic voltage and frequency scaling to match workload behavior to required performance level for best energy efficiency.

The advent of heterogeneous CPUs and GPUs system greatly complicates the three

challenges above, and they still remain unsolved despite significant research efforts dedicated to the problems. Our goal is to consider them together and provide a comprehensive analysis and technique that allows dynamic performance-aware, energy-efficient management for heterogeneous workloads running in modern high-performance computing system.

We take a holistic view of performance, power and resource management in modern computing systems by considering managements at different levels: workload scheduling among resource sharing clusters (Note that we refer to these resource-sharing clusters as memory domains because the shared resources mostly have to do with the memory hierarchy [1]), resource allocation within resource sharing clusters and DVFS-related control within the same voltage/frequency domain.

Existing work in this area addressed the three levels and tried to solve them individually. To simplify the analysis, they usually ignore the impacts between different levels. For instance, pre-core DVFS control without considering the impact of cache contention that caused by the frequency scaling of the processes running concurrently [2] [3], manage cache partition for throughput optimization without considering local frequency/ voltage scaling [4] To the best of our knowledge, there is very no previous work handles three levels together and very limited work considers the interdependency among each other [1][5].

We believe the following technical contributions are necessary to accomplish the goal of near optimal management of heterogeneous workloads (1) accurate modeling to illustrate the impact of resource contention and indicate the cost of local control policy; (2) efficient local control management and shared resource allocation for optimization in energy consumption and performance; (3) performance- and power-aware workload scheduling, jointly considering multiple metrics such as performance, energy consumption, and fairness etc..

1.1 Modeling High-Performance Computing Systems

Modeling high-performance computer systems is a difficult task. Typically, there are four major challenges when designing such models: (1) models need to be accurate. Although model estimation errors are tolerable or addressable through proper guard

banding in many applications, inaccurate estimation results will reduce the usability of such models. (2) Models need to be fast. Significant performance overhead prevents them from being used during runtime, making them inapplicable to many scenarios. In addition, when integrated with optimization techniques, slow models can lead to diminishing returns, or in extreme cases, render the entire optimization technique unusable. (3) The model construction process should be easy and automatic. Ideally, such modeling techniques should require no changes to the underlying hardware or operating system (OS) so that they can be applied to a variety of systems with different architectures. (4) The models should be scalable. The first requirement implies that the model designers must carefully test the models to ensure that the model estimation errors are small in all cases. A designer can improve model accuracy by incorporating more details into the model and simulating the interactions among different model components. However, this leads to higher computational complexity and therefore conflicts with the second requirement. In addition, the amount of inter-component interactions grows exponentially as more and more cores are integrated into the system. Hence, this approach cannot scale and thus conflicts with the last requirement. Similarly, the model performance can be improved by implementing it on hardware. However, this conflicts with the third requirement. Therefore, designers need to think carefully about the tradeoffs among the aforementioned attributes of the models to develop one that satisfies all the requirements.

Although there are many challenges when designing the models, it is usually worth the effort. Roughly speaking, models can be categorized into design-time models, assign-time models, and run-time models. Design-time models such as power grid models and IC thermal models can help designers to validate the correctness of their decisions during chip design. For example, understanding the thermal implications is essential because early-stage architectural decisions can significantly affect the design of cooling solutions. Assign-time models can predict the impact of process assignment on system metrics such as performance and power, helpful for designing intelligent assignment algorithms. Run-time models such as performance and power models enable system administrators and optimizers to dynamically monitor and predict changes in these runtime parameters, usually with little or no changes to the underlying hardware or applications. Furthermore, all these models have the potential to reveal the bottlenecks

in the system, thus motivating new software and hardware optimization techniques. Finally, modeling techniques is usually the first step toward optimization. In fact, all the optimization techniques proposed in this dissertation are motivated by the modeling techniques, most of which also heavily rely on these models. The ongoing move from single-core to heterogeneous architecture with CMPs and GPUs leads to more complex system architecture and applications, further emphasizing the need for fast and accurate models. In the future, processors are likely to integrate several tens or hundreds of cores on a single chip and probably requires a network on chip. Intel's recently unveiled 48-core chip is one such example. Without modeling and techniques similar to those described in this dissertation, it is very difficult, if possible at all, to develop optimization algorithms for such systems.

1.2 Optimizing High-Performance Computing Systems

Optimization techniques for high-performance computers are equally, if not more, important than modeling techniques. There are numerous attributes in high-performance computers designers attempt to optimize, e.g., performance, power consumption, temperature, and energy. Therefore, optimization techniques have a direct impact on user experience or system monetary cost by optimizing these attributes. It is usually possible to optimize one metric at the cost of another. However, this requires that the designers understand the trade-offs among various system metrics when developing such optimization techniques. There has been extensive studies on system-level optimization techniques for high-performance computing systems (see Chapter 5, Chapter 7, and Chapter 8). However, a large number of existing techniques only optimizes one metric and completely ignores other system metrics. Few algorithms that attempt to optimize a metric while constraining others either make unsubstantiated claims without resorting to accurate models, or rely on over-simplified models that produce inaccurate predictions and degrade the quality of optimization results. In our research, we carefully evaluate the trade-offs among various system metrics and design the optimization techniques based on accurate models when applicable.

1.3 Scheduling High-Performance Computing Systems

The emerging trend of multi-core CPUs and GPU systems allow more tasks running simultaneously, it's also getting more important to have some performance- and power-aware workload scheduling techniques, jointly considering multiple metrics such as performance, energy consumption, and fairness etc.. Such workload scheduling techniques need an accurate performance model to effectively predict the congestion suffered by running multiple tasks concurrently, and performance degradation from each individual tasks. Moreover, moving to the GPU side, workload scheduling technique not only need to determine which workload to scheduling together, but also need to figure out how much resource to be allocated on each workload. Overall, combining a high level workload scheduling algorithm with a set of local optimization techniques, is the ultimate solution to achieve a near optimal result given metrics such as throughput, energy etc..

1.4 Dissertation Overview

The rest of the dissertation is organized as follows.

Chapter 2–Chapter 5 focus on performance, power modeling and optimization techniques for CMPs. Chapter 2 gives an overview of the parameters that influence performance and power. We then describe the performance and power implications in modern CMP system. Chapter 3 describes a shared cache aware performance model for CMPs. We also provide an automated technique to collect process-dependent information needed by CAMP without resorting to simulation. Chapter 4 describes a system-level shared cache aware power model and an integrated model for fast and accurate power estimations during assignment in a multi-programmed CMP environment. Chapter 5 describes a predictive on-line dynamic voltage and frequency control (DVFS) algorithm that achieves close-to-optimal energy savings with a bounded performance degradation ratio.

Chapter 6–Chapter 9 focus on performance modeling and optimization and scheduling techniques for GPUs. Chapter 6 provides a thorough analysis on per-warp CPI breakdown, and identifies out all the key factors that govern GPU throughput from a single warp perspective. Chapter 7 proposes a new two-level priority scheduling policy

to improve performance in single kernel scenario. Chapter 8 proposes ICMT, a full, detailed solution for intra-core multitasking for GPGPUs, including architectural support and a contention-aware workload scheduling algorithm that improves TLP and PLP in a balanced fashion. Chapter 9 proposes a new contention-aware analytical performance model that can provides a fine-grain workload scheduling decision for intra-core multitasking, including detailed resource allocation from co-scheduled workloads.

In the end, we summarize the contributions of the work presented in this dissertation in Chapter 10.

Chapter 2

Multi-core CPU Overview

Performance and power issues are important challenges for the development of high-performance processors. As the industry has shifted their focus from single-core processors to CMPs, new performance and power model are desired. This chapter examines the impact of the current architecture paradigm shift on various modeling techniques and provides insights and motivations for the techniques proposed in Chapter 3–5.

The rest of this chapter is organized as follows. Section 2.1 gives an overview of the fundamental parameters that influences performance and power, many of which must be accounted for in the models to generate accurate estimations. Section 2.2 briefly summarizes CMP architecture background. Section 2.3 provides a motivation example.

2.1 Introduction

The performance of a computer can be defined as the amount of time required to accomplish one unit of work with one unit of resource. Not surprisingly, the performance of a chip is closely related to its clock frequency, which largely depends on the propagation delay of the transistors on the critical path, affected by supply voltage, temperature, and technology [6]. However a computer's performance cannot be solely determined by its chip's clock frequency; other factors such as instruction-level parallelism, thread-level parallelism, off-chip memory access latency, resource contention all contribute to the system performance.

The power consumption can be decomposed into dynamic power and static power.

Dynamic power consumption is caused by the charging and discharging events during voltage transitions in transistors. It scales quadratically with the supply voltage and linearly with the frequency of energy-consuming transitions. Static power, on the other hand, is independent of the frequency of such transitions. However, it has an exponential dependence on the supply voltage and temperature. Researchers have proposed numerous techniques to reduce the soaring power of computer systems, among which are dynamic voltage and frequency scaling and clock modulation [7, 8].

2.2 Background

CMP processor is composed of two or more independent CPU cores, thereby allowing more parallelism than single-core architecture. Each CPU core has its own private L1 caches, with the last-level cache being shared among all the cores to improve performance by supporting on-chip inter-process communication and allowing heterogeneous allocation of cache to processes running on different cores. However, a process may evict the data belonging to other processes with which it shares cache space, known as the cache contention problem. Intuitively, simultaneously running processes may influence each other's performance through sharing the cache. Furthermore, the performance (and indirectly power) impact is non-uniform, as the cache-sharing processes may have distinct memory access patterns. This requires that the performance model and power model for CMP systems explicitly account for the cache contention problem in addition to the time sharing problem in single-core systems.

2.3 Motivation

Cache sharing among processes running on different cores of a CMP can hide inter-process communication latency and improve cache utilization. This improvement is undermined by cache contention among concurrently running processes. To illustrate this effect, we wrote a synthetic *stressmark* that accesses the last-level cache very frequently. The stressmark is intentionally designed to exhibit extreme memory access behavior, for use in characterization. The stressmark is run concurrently with the process under

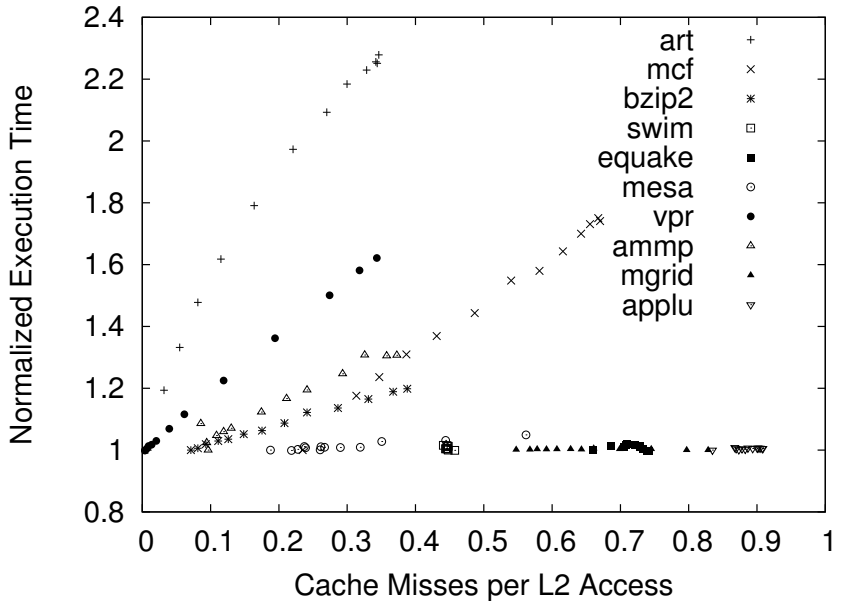

Figure 2.1: Impact of stressmark on performance of processes sharing case with it.

evaluation, on another core sharing the same cache. By varying the memory access behavior of the stressmark, we can change the number of last-level cache misses per cache access (MPA) for the stressmark, thereby controlling and measuring the performance impact on the other concurrently running process.

Figure 2.1 illustrates the relationship between the execution time, normalized to that when running the process alone, and MPA of the stressmark when it is run concurrently with each of 10 SPEC CPU2000 benchmarks. The relationship between MPA and execution time depends on the application. For example, with an MPA value of 0.35, the normalized execution time of *art* increased by 120% while that of *mesa* only increased by 1.5%. This demonstrates that the impact of cache contention on performance is application-dependent. Accurately predicting the performance and power consumption implications of assigning a particular set of processes to a CMP therefore requires a model that captures the variation in cache access and contention behavior among processes.

Chapter 3

Performance modeling on CMPs

The ongoing move to chip multiprocessors (CMPs) permits greater sharing of last-level cache by processor cores, but this sharing aggravates the cache contention problem, potentially undermining performance improvements. Accurately modeling the impact of inter-process cache contention on performance and power consumption is required for optimized process assignment. However, techniques based on exhaustive consideration of process-to-processor mappings and cycle-accurate simulation are inefficient or intractable for CMPs, which often permit a large number of potential assignments.

In this chapter, we propose CAMP, a fast and accurate shared cache-aware performance model for multi-core processors. CAMP estimates the performance degradation due to cache contention of processes running on CMPs. It uses reuse distance histograms, cache access frequencies, and the relationship between the throughput and cache miss rate of each process to predict its effective cache size when running concurrently and sharing cache with other processes, allowing instruction throughput estimation. We also provide an automated way to obtain process-dependent characteristics, such as reuse distance histograms, without offline simulation, operating system (OS) modification, or additional hardware. We tested the accuracy of CAMP using 55 different combinations of 10 SPEC CPU2000 benchmarks on a dual-core CMP machine. The average throughput prediction error was 1.57%.

The rest of this chapter is organized as follows. Sections 3.1 and 3.2 motivate the problem, summarize our contributions and present related work. section 2.3 describes

CAMP. section 3.4 introduces an automated way to characterize process memory access behavior to permit later prediction of cache contention. section 3.5 presents and discusses the experimental validation process and results. Finally, section 3.6 concludes this chapter.

3.1 Introduction

In recent chip multiprocessor (CMP) architectures, last-level caches are often shared among cores. This can improve performance by supporting on-chip inter-process communication and allowing heterogeneous allocation of cache to processes running on different cores. However, a process may cause the eviction of data belonging to other processes with which it shares cache space. This contention for shared cache space can cause simultaneously running processes to influence each other's performance. Moreover, the performance impact is non-uniform: it depends on the memory access behaviors of all processes with which it shares cache space.

The importance of inter-process cache contention for CMPs has been recognized in prior work [9, 10, 11]. However, the problem of predicting the impact of cache sharing on application performance during process assignment has been considered by only a few researchers [12, 13]. Knowing the performance implications of alternative assignment decisions can improve their quality. We therefore seek to build a cache contention model that permits fast and accurate performance prediction of processes on CMPs.

The construction of such a model should be easy and automatic; it should not require modifications to existing operating systems (OS) or hardware. Exhaustive offline simulation of process combinations is computationally intractable and should therefore be avoided. Moreover, prior work does not permit accurate prediction of the steady-state cache partition among arbitrary combinations of processes, which is a prerequisite for accurate performance prediction during assignment.

The chapter describes a fast and accurate shared cache aware performance model for multi-core processors (called CAMP). This model uses non-linear equilibrium equations in a least-recently-used (LRU) or pseudo-LRU last-level cache, taking into account process reuse distance histograms, cache access frequencies, and miss rate-aware performance degradation. CAMP models both cache miss rate and performance degradation

as functions of process effective cache size, which in turn is a function of the memory access behavior of other processes sharing the cache. CAMP can be used to accurately predict the effective cache sizes of processes running simultaneously on CMPs, allowing performance prediction with an average error of only 1.57%. We also propose an easy-to-implement method of obtaining the reuse distance histogram of a process without offline simulation or modification to commodity hardware or OS. In contrast with existing techniques, the proposed technique uses only commonly available hardware performance counters. Finally, we evaluate the generality of CAMP by profiling processes on one CMP and using the resulting models to accurately predict process performance when run on two other CMPs having different cache sizes. All the measurements are performed on real processors.

3.2 Related Work

Past work [14, 15, 16, 17] has considered the problem of adjusting cache partitioning during runtime after process assignment decisions have already been made. In contrast, the goal of our work is to predict the performance implications of process assignment decisions before execution. Other researchers have developed performance prediction models to guide process assignment. However, most [18, 19] addressed cache contention only for uniprocessors on which only a single process may run at a time. The move to CMPs will aggravate the cache contention problem since multiple processes can run on different cores simultaneously.

Resource contention models for simultaneous multithreading (SMT) uniprocessors should be applicable to CMP systems due to the similarity in inter-process resource contention. However, existing work on resource contention modeling for SMT processors either suffers from large performance prediction error (20% of instruction throughput predictions deviate by more than 20% from the actual instruction throughput) [20] or requires modifications to the underlying hardware [21]. To the best of our knowledge, existing performance models for SMT processors do not support accurate runtime performance prediction. Although the similarity of cache effects for CMPs and SMT processors suggests that the modeling technique described in this chapter might also be accurate for SMT processors, we have not yet experimentally tested this hypothesis.

Researchers have also considered addressing the performance prediction problem using offline simulation [22] or modifications to the existing hardware or operating system [23]. For example, Suh *et al.* [16] proposed to add a hardware counter to each cache way and use them to determine the reuse distance histogram. Our goal is runtime prediction of the performance of a process concurrently running on a shared-cache CMP, without requiring prior characterization.

Tam *et al.* [24] previously developed a technique to predict miss rate as a function of cache size by using built-in hardware performance counters, with a primary goal of supporting on-line optimization of cache partitioning among processes. They do not explain how to use miss rate curves to predict instruction throughputs for processes sharing cache space. Also, their approach relies on performance counters peculiar to the POWER5 architecture.

Chandra *et al.* [13] proposed three analytical models to predict miss rates for processes sharing the same cache. Their models use the reuse distances and/or circular sequence profiles for each thread to predict inter-thread cache contention. These models require knowledge of the steady-state L2 cache access frequency of a process when concurrently running with other processes. In reality, obtaining this information without running or simulating all potential combinations of concurrent cache-sharing processes is impractical.

Chen *et al.* [12] proposed a two-phase approach for performance prediction. In the first phase, the access frequency of a process running alone is used to estimate performance. In the second phase, the performance estimates from the first phase are refined to consider the implications of cache contention. The models proposed in each paper require processing circular memory access sequences, which must be obtained by tracing execution with an instruction-set simulator or non-standard detailed access tracing hardware.

3.3 Analytical Model

This section describes the main components in CAMP, namely its performance model, effective cache size estimation technique, and steady-state condition estimator.

3.3.1 Background

In this section, we define some basic terms that will be used throughout this chapter. Our study will consider an N-core processor with an L2 last-level on-chip cache. In the rest of the chapter, we refer to "L2 cache" as "cache". A set-associate cache is broken into *sets*, each of which has space for multiple *lines*, i.e., the minimal unit of data fetched by or evicted from a cache. The number of lines per set is the cache's associativity, i.e., its number of *ways*. A line at a particular location in memory is associated with a set and may be fetched into any line in the set.

Effective Cache Size

When multiple processes share a cache, they compete for limited space. The division of cache space among processes is influenced by characteristics of the concurrently running processes, such as cache access frequency and sequential data access patterns. We define *effective cache size* of process i to be the average number of ways occupied by the process in a set, denoted as S_i. Therefore,

$$\sum_{i=1}^{N} S_i = A, \tag{3.1}$$

where N is the total number of processes sharing the cache and A is the number of ways in the cache. Note that S_i is a real value in our model because it represents the average number of ways process i occupies in a set during prolonged execution. If the cache access behavior of all processes is static, then S_i will be stable. We define this as the *steady-state* condition.

Reuse Distance

We define the *reuse distance*, R_j, of cache line j to be the number of distinct cache lines within the same set accessed between two consecutive accesses to line j. A *reuse distance histogram* represents the distribution of cache line reuse distances for an entire shared cache. Given an A-way set-associative cache, Figure 9.7 shows a reuse distance histogram for the *mcf* application (see section 3.5). The x-axis shows the reuse distance and the y-axis shows the normalized frequencies of the associated reuse distances. The first bar in the histogram, i.e., $hist_1$, gives the probability that a most-recently-used

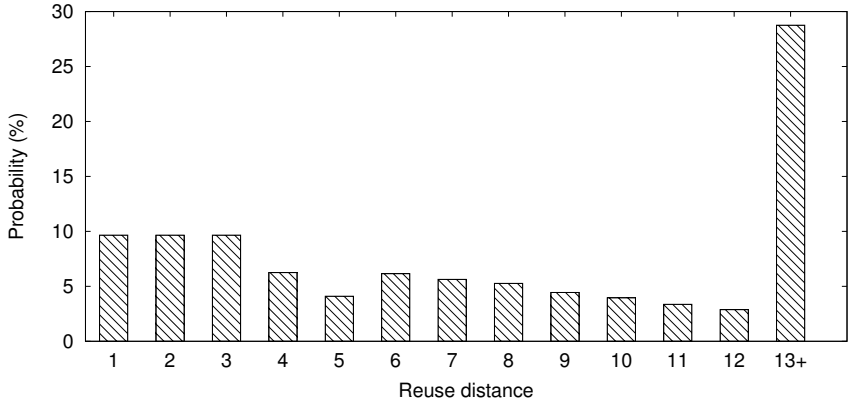

Figure 3.1: Cache line reuse distance histogram for *mcf* application.

line will be accessed again, while the last bar, i.e., $hist_{13+}$, gives the probability that the data for the next cache access does not exist in the most-recently-used 12 lines, which can be denoted as $\sum_{k=13}^{\infty} hist_k$. $Hist_{\infty}$ is the probability that the data in the line is never accessed again. Note that $hist_{\infty}$ can be very large for some streaming applications. For process i with an effective cache size of S_i, all accesses to the cache lines with a reuse distance larger than S_i result in cache misses. Hence, the probability of a cache access resulting in a miss for process i with an effective cache size of S_i can be expressed as follows.

$$\mathrm{MPA}_i(S_i) = \int_{S_i}^{\infty} \mathrm{hist}_i(x)\, dx. \tag{3.2}$$

Note that $\mathrm{hist}_i(x)$ is a continuous function derived using linear interpolation of the discrete histogram to support estimation for non-integer average reuse distances.

3.3.2 Problem Formulation and Assumptions

The cache contention prediction problem can be formulated as follows: given N processes assigned to cores sharing the same A-way set-associative last-level cache, predict the steady-state cache size occupied by each process during concurrent execution. Note that the steady-state cache size can be directly translated to performance, as illustrated by Equation 3.2. Solving this problem is helpful for process assignment and migration in a CMP environment because it allows one to predict the consequences of tentative process assignment and migration decisions. However, accurate prediction of process

performance is challenging because there are many combinations of processes that may share the same cache.

We make the following assumptions.

1. For each process, we assume that data accesses are uniformly distributed across all cache sets. The temporal cache access behaviors such as number of cache accesses per second (APS) and the reuse distance histogram (see section 6.2) are assumed to be stationary. In the case of multiple non-repeating phases with distinct memory access patterns [25], non-repeating phases should be modeled separately.

2. We assume no hardware prefetching. Hardware prefetching predictively fetches cache lines based on access patterns, potentially complicating the model. As such, the model might be inaccurate for systems using prefetching. However, we argue that prefetching is of limited value on CMPs with constrained processor-memory bandwidth. For the 10 benchmarks used in this work, the average improvement was 3.25%, and only *equake* benefitted significantly.

3. We do not explicitly model the effect of kernel thread and instruction accesses on cache contention but note that the resulting technique remains accurate in the presence of these accesses.

4. The cache uses an LRU replacement policy. Although most modern caches use pseudo-LRU policies, assuming LRU still permits high prediction accuracy.

Although these assumptions simplify the explanation of our analysis, we do not rely on them but instead "close the loop" by evaluating the resulting prediction technique on real systems for which the assumptions may not hold. Finally, we consider a multi-programmed environment and therefore neglect communication among processes. Our analysis will hold for applications in which there is little communication among processes assigned to separate cores.

3.3.3 Performance Model

The average number of cache accesses per second (APS) reflects how aggressively a process competes for cache space. All other things being equal, a process with high

APS will generally take up more space in a shared cache than a process with low APS.

$$\text{APS} = \frac{\text{API}}{\text{SPI}}, \tag{3.3}$$

where API is the number of cache accesses per instruction and SPI is the number of seconds per instruction. API is a process property: given the same input data, the API of a process is fixed. On the other hand, SPI is largely affected by the number of cache misses per second (MPS). The latency per instruction, i.e., seconds per instruction, can be decomposed into two parts: on-chip latency due to computation and off-chip latency caused by main memory and disk accesses. When the CPU frequency remains constant, the on-chip latencies per instruction are approximately constant for a process. As shown in Figure 2.1 we experimentally determined that SPI can be expressed as a linear function of MPA.

$$\text{SPI} = \alpha \cdot \text{MPA} + \beta, \tag{3.4}$$

where α and β are parameters that can be obtained during offline characterization.

3.3.4 Estimating Effective Cache Size After n Accesses

In this section, we use the reuse distance histogram of a process to derive its effective cache size. Consider the number of distinct cache lines, s, (i.e., the effective cache size of the process) after n accesses in one set. Note that s is essentially the effective cache size, S_i, as defined in section 3.3.1. Given that $P_{s,n}$ is the probability of having s distinct cache lines after n consecutive cache accesses, $P_{hit,s}$ is the probability that a cache access will result in a cache hit when the process already has s cache lines, and $P_{miss,s}$ is the probability that a cache access will result in a miss when the process has s cache lines, noting s can never be greater than n, the following recursive equation can be derived:

$$P_{s,n} = P_{s,n-1} \cdot P_{hit,s} + P_{s-1,n-1} \cdot P_{miss,s-1}, 1 < s \leq n. \tag{3.5}$$

This can be explained as follows. The fact that n cache accesses result in an effective cache size of s can only be the result of one of the following scenarios.

1. In scenario A, the first $n-1$ cache accesses led to an effective cache size of s and the nth access resulted in a cache hit. Since the nth access did not lead to an

increase in the effective cache size, it remains s. The probability of this scenario, $P(A)$, is $P_{s,n-1} \cdot P_{hit,s}$.

2. In scenario B, the first $n-1$ cache accesses lead to an effective cache size of $s-1$ and the nth access causes a cache miss. In this case, the effective cache size is increased by one, relative to the $s-1$ lines resulting from the first $n-1$ accesses. Thus, the effective cache size will be s after n cache accesses. The probability of this scenario, $P(B)$, is $P_{s-1,n-1} \cdot P_{miss,s-1}$.

Noting that $P_{s,n} = P(A) + P(B)$, we can derive Equation 3.5.

Given that MPA(s) is the probability of a cache access missing, given an effective cache size of s, Equation 3.5 can be written as

$$P_{s,n} = P_{s,n-1} \cdot (1 - \text{MPA}(s)) + P_{s-1,n-1} \cdot \text{MPA}(s-1). \tag{3.6}$$

Note that $P_{1,1} = 1$ because the first cache access always causes a cache miss and replacement and $1 < s \leq n$. Assuming the process reaches steady state after n accesses, and given that $\text{G}_i(n)$ is the effective cache size for process i after n accesses, we have

$$\text{G}_i(n) = \sum_{s=1}^{n} (P_{s,n} \cdot s). \tag{3.7}$$

Note that $\text{G}_i(n)$ is a monotonically increasing function of n. Therefore, given the effective cache size of process i, S_i, we can deduce the number of cache accesses n needed for the process to reach steady state using the inverse function of $\text{G}_i(n)$, i.e., $n = \text{G}_i^{-1}(S_i)$.

3.3.5 Steady-State Conditions

Given a cache with an LRU-like replacement policy, it is reasonable to assume that at time t, we can always find a duration T such that data accessed before time $t-T$ have been evicted and data accessed during $[t-T, t]$ are presently in the cache. To determine the effective cache size, we are only interested in data accessed during $[t-T, t]$. Since none of these accesses will evict any data lines accessed during $[t-T, t]$, it is as if the data were written to an empty cache with no cache misses during $[t-T, t]$. Thus, Equations 3.6 and 3.7 hold. Note that these accesses may still evict cache lines accessed before $t-T$. We assume the partition among processes resulting from data accesses

from all co-running processes within $[t-T,\ t]$ is the same as that when all the processes reach steady state. By computing the cache size of each process resulting from data accesses within $[t-T,\ t]$, we can determine process effective cache sizes. Hence, the effective cache size of process i, denoted as S_i, corresponds to the expected cache size determined by the most recent $\text{APS} \cdot T$ cache accesses for process i. Thus, the effective cache S_i is written as $\text{G}_i(\text{APS}_i \cdot T)$. Conversely, APS_i can be expressed as $\text{G}_i^{-1}(S_i)/T$. From Equation 3.3 and 3.4, we can derive the following equation:

$$\text{APS}_i = \frac{\text{G}_i^{-1}(S_i)}{T} = \frac{\text{API}_i}{\alpha_i \cdot \text{MPA}_i(S_i) + \beta_i}. \tag{3.8}$$

Therefore,

$$T = \frac{\text{G}_i^{-1}(S_i) \cdot (\alpha_i \cdot \text{MPA}_i(S_i) + \beta_i)}{\text{API}_i}. \tag{3.9}$$

Note that Equation 3.9 holds for any process i, where $i \in \{1, 2, \cdots, N\}$, given that N is the total number of processes. Combined with Equation 3.1, we have

$$\frac{\text{G}_1^{-1}(S_1)}{\text{G}_j^{-1}(S_j)} - \frac{\text{API}_1 \cdot (\alpha_j \text{MPA}_j(S_j) + \beta_j)}{\text{API}_j \cdot (\alpha_1 \text{MPA}_1(S_1) + \beta_1)} = 0, \forall_{j=1}^{N}, \tag{3.10}$$

$$\text{and } \sum_{i=1}^{N} S_i - A = 0, \tag{3.11}$$

where $\text{G}_i^{-1}(S_i)$ and $\text{MPA}_i(S_i)$ are application-dependent non-linear functions of S_i. We solve Equation 3.10 using Newton–Raphson iteration, a standard numerical method for finding the roots of non-linear equations. Note that the number of ways in a cache (A) and number of cores (N) are each fewer than 10. $\text{G}_i^{-1}(S_i)$ and $\text{MPA}_i(S_i)$ are monotonic functions of S_i, so we can solve S_i for process i accurately within several iterations, where i ranges from 1 to N. The initial guess also affects the computational cost. In our experiments, we find that initially guessing that the effective cache size of a process i is proportional to its APS allows quick convergence to an accurate solution.

3.4 Automated Profiling

In this section, we first explain how to obtain the reuse distance histogram of a process. We then describe how to derive other parameters such as API and MPA. After that, we

give details about the automated profiling process. Finally, we indicate possible sources of prediction error.

3.4.1 Reuse Distance Profiling

Process reuse distance histograms play a central role in the proposed performance modeling technique. It would be possible to extract the reuse distance histograms of processes via simulation, and CAMP would dramatically improve estimation speed even if simulation were used for initial characterization; however, there is a faster alternative.

Most modern processors have built-in hardware performance counters (HPCs) that record information about architectural events such as the number of instructions retired, number of last-level cache accesses, and number of last-level cache misses [26]. Therefore, we can gather information about parameters such as SPI and MPA accurately. However, existing hardware or software resources do not directly provide reuse histogram data. We now explain the process of deriving reuse histogram data from directly monitored parameters.

Consider two processes running on separate cores sharing an A-way last-level cache. We assume if one process occupies l ways in a cache set, the concurrently running process will occupy $A - l$ ways. Based on Equation 3.2, we can compute the effective cache size of a *stressmark* with a controlled MPA and a known reuse distance histogram. We obtain the reuse distance histogram of a process (denoted as B) as follows. Run the stressmark along with B multiple times. In the lth run, we tune the parameters in the stressmark to change the effective cache size, denoted as $S_{stress,l}$. Record B's MPA in each run, denoted as $MPA_{B,l}$, where $l \in \{1, 2, \cdots, A\}$. Given that $S_{B,l}$ is process B's effective cache size in the lth run, and considering the lth and the $l+1$st runs, we have

$$\begin{aligned} MPA_{B,l+1} &= \int_{S_{B,l+1}}^{\infty} \text{hist}_B(x)dx \text{ and} \\ MPA_{B,l} &= \int_{S_{B,l}}^{\infty} \text{hist}_B(x)dx. \end{aligned} \tag{3.12}$$

See the discussion after Equation 3.2 for the definition of $\text{hist}(x)$. Hence, we can estimate the probability of process B having an effective cache size of $S_{B,l}$ as

$$\text{hist}_B(S_{B,l}) \approx MPA_{B,l+1} - MPA_{B,l}. \tag{3.13}$$

Algorithm 1 Stressmark with k-Way Occupation

1: *Set* is the number of cache sets.
2: *Step* is the number of integers per cache line.
3: $S[Set \cdot Step \cdot \mathrm{k}]$ is an array of integers.
4: $Index \leftarrow \{s_1, s_2, \cdots, s_n\}$
5: The following loop loads a predefined random sequence into *Index*.
6: **for** $j = 0 : n - 1$ **do**
7: $flag \leftarrow Index[j]$
8: $T \leftarrow \&S[flag \cdot Set \cdot Step]$
9: **for** $i = 0 : Set - 1$ **do**
10: read $T[i \cdot Step]$
11: **end for**
12: **end for**

By varying $S_{B,l}$ from 1 to A, we can estimate the probability at each effective cache size, thus allowing us to construct the reuse distance histogram. Since we can not control $S_{B,l}$ directly, in practice we adaptively tune the effective cache size of the stressmark from run to run. $S_{B,l} + S_{stress,l} = A$. Therefore, varying $S_{stress,l}$ changes $S_{B,l}$.

As indicated above, the stressmark should have the following properties.

1. High cache access frequency, i.e., high API. API is related to the degree to which a process competes for cache space. In order to estimate the probability of a process having a small effective cache size, the concurrently running stressmark should occupy a large portion of the cache with few cache misses.

2. A uniform reuse distance histogram, i.e., the probability is the same across all possible reuse distances. This makes it easy to compute the effective cache size given an MPA value. In addition, given a pseudo-LRU cache replacement policy, cache lines other than the least recently used will sometimes be evicted. Having a uniform reuse distance histogram minimizes the impact of this potential problem because the replacement noise will affect cache lines with all reuse distances equally.

The pseudo-code of the stressmark is shown in 1, where *Set* is the number of sets in the cache, *Step* is the number of integers per cache line. *Index[n]* is an integer array whose elements are uniformly distributed from $[1, k]$, which contains a random access location sequence. In order to maintain high cache access frequency for the stressmark,

we pre-generate these arrays. Note that in Line 10 in 1, two consecutive reads are *Step* elements apart to ensure a 100% L1 cache miss rate. Since the stressmark randomly accesses k cache lines within a cache set, the effective cache size of the stressmark is expected to be k. However, this may not be very accurate due to conflict misses between the stressmark and the process of interest. In reality, we use Equation 3.2 to estimate the effective cache size of the stressmark, i.e., $S_{stress} = \text{MPA}^{-1}(\text{MPA}_{stress})$, where MPA_{stress} is the MPA of the stressmark and $\text{MPA}^{-1}()$ is the inverse function for MPA in Equation 3.2 that converts MPA to an effective cache size, i.e., $\text{MPA}^{-1}(\text{MPA}(x)) = x$.

3.4.2 Automated Parameter Estimation

In this section, we describe how we calculate parameters such as API and SPI for a process. Given an A-way associative cache, in order to get the reuse distance histogram for a process, we run the stressmark concurrently with the process A times. In the lth run, we set k to l for the stressmark in 1. Since API is fixed for a process with the same input data, given that API_l is the process's API in the lth run, the average API of the process can be estimated as

$$\text{API} = \frac{\sum_{l=1}^{A} API_l}{A}. \tag{3.14}$$

Similarly, we can get A pairs of a process' MPA and SPI values from the A runs. Given that MPA_l and SPI_l are the average MPA and SPI of the process in the lth run, the α and β in Equation 3.4 can be determined using linear regression, i.e.,

$$\alpha = \frac{A \cdot (\sum_{l=1}^{A} MPA_l \cdot SPI_l) - (\sum_{l=1}^{A} MPA_l)(\sum_{l=1}^{A} SPI_l)}{A \cdot (\sum_{l=1}^{A} {MPA_l}^2) - (\sum_{l=1}^{A} MPA_l)^2} \tag{3.15}$$

$$\text{and } \beta = \frac{(\sum_{l=1}^{A} SPI_l) - \alpha \cdot (\sum_{l=1}^{A} MPA_l)}{A}. \tag{3.16}$$

Note that most programs have repeating phases with periods ranging from 200 ms to 2,000 ms [25]. Numerous works exist on phase detection, i.e., finding the time at which the process switches from one phase to another. Since the process behavior is by definition similar within a phase, one set of parameters per phase is sufficient. In the rest of the chapter, we will treat processes as having a single phase each to simplify explanation. Note that the proposed technique is also suitable for multi-phase processes, for which each phase may have a different set of extracted parameters.

Process characterization can be automated as follows. First, run the stressmark along with the process A times, varying the effective cache size. After A runs, API, α, β, and the reuse distance histogram can be estimated using Equations 3.13–3.16. These four parameters form the *feature vector* of a process. Given the feature vectors of two processes, we can predict their effective cache sizes when sharing a cache, which in turn can be translated to SPI values using Equations 3.2 and 3.4. Note that the SPIs for the two processes are predicted without actually running them concurrently. Hence, given N processes for assignment to N cores, only N feature vectors are needed ($\mathcal{O}(N)$ complexity). These vectors can be used to predict the performance of any subset of the N processes during assignment ($2^N - 1$ combinations). Thus, the proposed technique is dramatically more efficient than one requiring simulation or execution of $2^N - 1$ combinations of processes.

3.4.3 Potential Sources of Error

There are two primary sources of error in the proposed technique: error in histogram estimation and error in linear regression analysis. We will explain these error sources now, but note that even with these error sources, the proposed technique is highly accurate (see section 3.5).

When estimating the reuse distance histogram for a process, it is very difficult to capture the probability corresponding to a reuse distance close to 0 because the concurrently running stressmark cannot consume all of the cache space. Similarly, the estimation for a reuse distance close to A may also have some error. In practice, we assume a uniform distribution for reuse distances close to 0 or A. Linear interpolation, given an assumed miss rate of 1 at an effective cache size of zero, is used for very small effective cache sizes. In addition, the probability of reuse distances larger than A cannot be captured by our technique. Hence, we extrapolate this probability based on the derivative of the probability density function at a sample point close to A.

Error may also be introduced due to noise in sample parameters. When the <MPA, SPI> pairs gathered during profiling are clustered within a small region, linear regression may lead to inaccurate estimation of coefficients due to noise. We addressed this problem by bounding the step size during Newton–Raphson iteration when solving for the effective cache size (see Equation 3.10), permitting convergence.

Table 3.1: Intel P8600 Specification

Item	Specification
Number of chips	1
Number of cores per chip	2
Frequency	2.40 GHz
L1 ICache (Private)	32 KB, 64 B line, 8-way associative
L1 DCache (Private)	32 KB, 64 B line, 8-way associative
L2 Cache (Shared)	3 MB, 64 B line, 12-way associative

3.5 Evaluation Methodology and Results

In this section, we first describe our experimental setup. We then present the experimental results for model validation. We contrast the proposed technique with other potential methods of predicting CMP cache contention among processes and indicate which features of the proposed approach permit high prediction accuracy.

3.5.1 Experimental Setup

We evaluated our technique on a computer equipped with an Intel Core 2 Duo P8600 processor and the Mac OS X 10.5 operating system. The system parameters are listed in Table 3.1. We used Shark, a built-in profiling tool, to sample performance counters at a period of 2 ms. The samples are used for calculating parameters (e.g., API, MPA, and SPI) on each core. We used the SPEC CPU2000 benchmark suite, which contains 26 benchmarks. Since validating all 351 pairwise combinations would be costly, we instead selected a subset containing five CPU-intensive and five memory-intensive benchmarks, and considered all pairwise combinations of these ten. We recorded the program phase information for each benchmark during pre-characterization. Experimental results indicate that all but two benchmarks have only one significant phase, as defined by our parameters of interest. The longest phases in *art* and *mcf* were used. We can thus address the prediction problem one phase at a time using phase detection algorithms, as described in subsection 3.4.2.

Table 3.2: API, α, and β for Different Benchmarks

Benchmark	art	mcf	bzip2	swim	equake	mesa	vpr	ammp	mgrid	applu
API	0.0225	0.0733	0.0044	0.0116	0.0074	0.0013	0.0102	0.0092	0.0018	0.0018
α ($\times 10^{-9}$)	446	134	99.9	-99.6	60.5	30.7	306	243	0.609	3.12
β ($\times 10^{-7}$)	1.34	5.86	1.50	1.97	2.28	1.55	1.65	1.83	1.28	1.15

3.5.2 Pre-Characterization

As indicated in subsection 3.4.2, we first run the stressmark concurrently with each benchmark on two different cores 12 times to derive various parameters such as API, MPA, and SPI. Each run lasts 10 s, which has proven sufficient for characterizing these parameters. Note that the working data set size of the stressmark is incremented by 1 way after each run to construct the reuse distance histogram for each benchmark, as described in subsection 3.4.1.

Analyzing API, α, and β

Hardware performance counter readings are analyzed to determine API, α, and β in Equations 3.3 and 3.4. Table 3.2 shows the value for each benchmark. API indicates an application's capability to compete for cache space. It also indicates whether an application is memory-intensive because higher API is usually associated with more misses per instruction, resulting in more off-chip memory transactions. As indicated in Table 3.2, benchmarks such as *art*, *mcf*, *vpr*, *swim*, and *ammp* are memory-intensive. Their APIs are significantly higher than those of the other benchmarks. α indicates sensitivity to cache misses in terms of performance. Equation 3.4 implies that for the same amount of change in MPA, a larger α indicates a larger change in SPI. As shown in Table 3.2, the performance of memory-intensive applications tends to be more sensitive to cache misses than that of CPU-intensive applications, with *art* being the most cache-miss sensitive benchmark and *mgrid* being the least cache-miss sensitive benchmark. Note that α is negative for *swim*. This is because cache contention has little impact on this benchmark's MPA value, resulting in inaccurate estimation during linear regression when building the SPI model. However, this introduces little error in performance estimation because, as we show later in Figure 3.2, both MPA and SPI are insensitive to effective cache size for this benchmark.

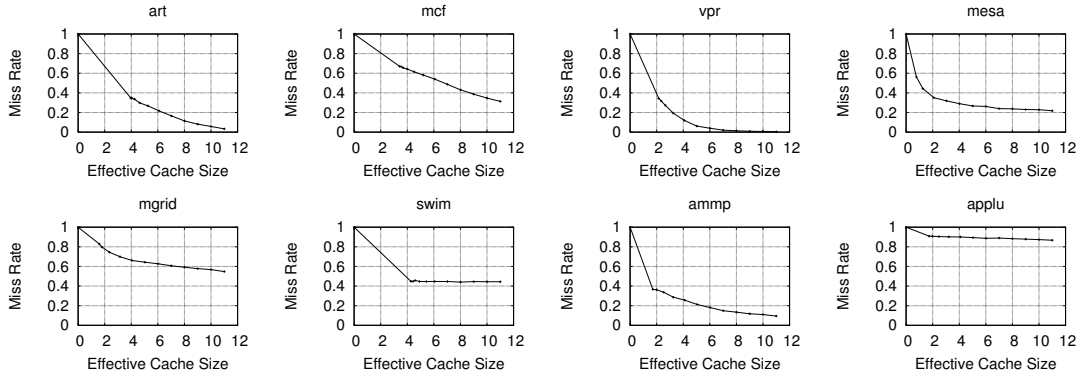

Figure 3.2: Profiled cache miss rate corresponding to effective cache size.

Analyzing Cache Miss Rate

We use the approach explained in subsection 3.4.1 to build the reuse distance histogram for each benchmark, which is then used to predict its cache miss rate as a function of effective cache size. Figure 3.2 illustrates the relationship between cache miss rate and effective cache size for each benchmark. The cache miss rate curves for benchmarks *bzip2* and *equake* are not shown because they are similar to that of *mgrid*. The results, from execution on hardware, are consistent with those obtained from simulation [27]. Note that linear approximation is used for leftmost segment of each miss rate curve, for the reasons given in subsection 3.4.3. However, for the benchmarks with high APIs such as *swim* and *applu*, the solutions of Equation 3.10 always lie outside this linear region. Therefore, we do not consider this region when analyzing the sensitivities of the cache miss rate curves for any benchmarks. As indicated in Figure 3.2, the cache miss rates of benchmarks such as *swim* and *applu* are insensitive to their effective cache sizes in the effective range. Therefore, their performance is only slightly affected when run together with other benchmarks. However, cache miss rates of benchmarks such as *art* and *vpr* are very sensitive to their effective cache sizes. Therefore, their performances will be significantly affected by cache contention, although the impact on their performances highly depends on the memory access patterns of the processes running concurrently with them. This indicates the importance of considering application behavior and cache contention during performance prediction on CMPs.

3.5.3 Model Validation

In this section, we validate our technique by using the *feature vector*, i.e., $<\text{API}, \alpha, \beta>$, and reuse distance histogram of a benchmark to predict the performance when run concurrently with another benchmark. Note that feature vectors are determined during pre-characterization. We compare the performances of the two benchmarks during the evaluation period to the predicted performances using the feature vectors of the benchmarks. Note that the approach proposed by Chandra *et al.* [13] requires the steady-state cache access frequency of a process to be known a priori. We see no practical way to accurately predetermine this value for concurrently running processes. In contrast, our technique determines the steady-state cache access frequency using analysis of performance counter readings, i.e., the proposed technique works correctly using only inputs that are readily available in real systems.

In addition to the proposed technique, we considered and evaluated two alternatives. The first, called Accesses Based (AB), assumes the effective cache size of a process is proportional to APS. Given two processes running on two cores with effective cache sizes of S_1 and S_2, the formula to determine effective cache sizes can be written as

$$\frac{\text{APS}_1}{\text{APS}_2} = \frac{S_1}{S_2} = \frac{\text{API}_1 \cdot (\alpha_2 \text{MPA}_2(S_2) + \beta_2)}{\text{API}_2 \cdot (\alpha_1 \text{MPA}_1(S_1) + \beta_1)}. \tag{3.17}$$

Note that this model only considers APS. It may be inaccurate if the concurrently running processes have different MPAs or reuse frequencies. The second model, known as Misses Based (MB), assumes that S_i is proportional to MPS. Therefore, the equation used to determine S_1 and S_2 is

$$\frac{\text{MPS}_1}{\text{MPS}_2} = \frac{\text{MPA}_1(S_1) \cdot \text{API}_1 \cdot (\alpha_2 \text{MPA}_2(S_2) + \beta_2)}{\text{MPA}_2(S_2) \cdot \text{API}_2 \cdot (\alpha_1 \text{MPA}_1(S_1) + \beta_1)}. \tag{3.18}$$

The model only considers MPS. Thus it may be also inaccurate if the concurrently running processes have different reuse distance profiles.

Analysis of Results

We examined all 55 possible pairwise combinations of 10 benchmarks: each benchmark is paired with every other benchmark (including another instance of itself) and assigned to the two cache-sharing cores. The measured performance data are then compared to

Table 3.3: Prediction Accuracy for Cache Misses and Performance Degradation

Benchmark	CAMP				AB				MB			
	MPA		SPI		MPA		SPI		MPA		SPI	
	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)
art	1.61	0	3.68	40	4.60	50	10.26	80	5.88	70	18.09	90
vpr	0.88	0	1.48	0	4.70	40	7.67	60	5.89	30	9.24	50
mcf	2.10	10	3.70	20	2.82	10	3.97	40	6.79	40	7.72	70
ammp	2.82	20	3.04	20	4.03	30	4.16	30	5.89	60	6.78	90
bzip2	1.86	10	1.17	0	3.17	20	1.89	0	6.09	60	3.63	30
mesa	4.23	50	0.83	0	4.90	30	0.94	0	7.77	50	1.55	0
swim	0.28	0	0.86	0	0.23	0	0.81	0	0.27	0	0.78	0
equake	0.70	0	0.38	0	0.92	0	0.41	0	1.43	0	0.45	0
applu	1.13	0	0.32	0	0.86	0	0.31	0	1.79	10	0.33	0
mgrid	2.79	10	0.28	0	3.35	20	0.28	0	6.00	40	0.30	0
top 5 average	1.86	8	2.61	16	3.86	30	5.59	42	6.11	52	9.09	66
average	1.86	4	1.57	8	2.94	20	3.07	21	4.78	36	4.89	33

those predicted by AB, MB, and CAMP. AB and MB are not past work. They are in fact alternative prediction models we considered.

Table 3.3 presents the average prediction error in cache miss rate and performance for each benchmark when run simultaneously with each of the 10 benchmarks. The first column lists the benchmarks. Columns 2, 6, and 10 show the average magnitudes of cache miss estimation error for CAMP, AB, and MB. Columns 3, 7, and 11 show the percentage of test cases with a cache miss estimation error larger than 5% among all 10 test cases. Similarly, Columns 4, 8, and 12 indicate the average relative estimation error in performance for the three techniques, while columns 5, 9, and 13 indicate the percentage of test cases with a relative performance estimation error larger than 5% among all 10 test cases for the three techniques. The last two rows correspond to the results for the 5 most memory-intensive benchmarks and all 10 benchmarks, respectively.

As indicated in Table 3.3, CAMP has an average of 1.57% performance estimation error over all 10 benchmarks, compared to 3.07% for AB and 4.89% for MB. In addition, only 8% of the cases for CAMP have estimation errors greater than 5%, compared to 21% for AB and 33% for MB. Note that all three models have average performance estimation errors below 5%. This is mainly because all the three models are based on predicting the effective cache size of each benchmark when subject to cache sharing. If one of the two co-running benchmarks are CPU-intensive, e.g., *mesa*, *applu*, or *mgrid*, at least one

Table 3.4: MPA and SPI Prediction when Processes Run with Art

Benchmark	Extra MPA	Extra SPI	CAMP			AB		MB	
			Itera-tions	MPA Error	SPI Error	MPA Error	SPI Error	MPA Error	SPI Error
	(%)	(%)		(%)	(%)	(%)	(%)	(%)	(%)
art	17.40	72.01	1	-1.96	+4.89	-1.96	+4.89	-1.96	+4.89
mcf	16.72	72.62	6	-1.52	+2.38	-7.16	+12.44	+13.60	-41.06
bzip2	6.13	31.48	5	+0.52	-0.13	-2.20	+6.82	+5.97	-17.71
swim	16.20	71.12	6	-4.12	+7.15	-9.35	+15.76	+6.58	-17.39
equake	10.92	48.03	8	+0.60	+0.19	-8.03	+17.47	+10.45	-31.18
mesa	2.33	13.93	4	-0.33	+5.60	-2.56	+11.50	-0.17	+5.18
vpr	8.41	42.24	5	+0.03	-0.66	-0.07	-0.41	+6.00	-18.72
ammp	5.42	32.84	5	-2.33	+4.45	-5.54	+11.80	+3.77	-13.48
mgrid	7.76	37.85	4	+2.17	-5.01	-3.29	+8.67	+5.26	-14.73
applu	9.40	44.74	6	+2.48	-6.38	-5.83	+12.79	+6.90	-20.46
average	10.07	46.69	5	1.61	3.68	4.60	10.26	6.07	18.48

of the two following conditions holds: (1) its cache miss rate is insensitive to its effective cache size or (2) its performance is insensitive to its cache miss rate. Therefore, the large cache miss estimation error may not be reflected in performance estimation error. This also explains why memory-intensive benchmarks have larger estimation error than CPU-intensive benchmarks. In Table 3.3, the bottom 5 benchmarks are either CPU-intensive applications or streaming applications with constant high miss rates, e.g., *swim*. Their performance estimation errors are below 1% for all three models. We thus also list the average performance estimation error for the top 5 benchmarks, which are relatively sensitive to the cache misses. CAMP has an average of 2.61% performance prediction error, compared to 5.59% for AB and 9.09% for MB.

Analyzing One Benchmark–Art

We now examine the accuracy of the three models when a specific benchmark, namely *art*, runs simultaneously with other benchmarks. Table 3.4 presents the estimation error for MPA and SPI using CAMP, AB, and MB when *art* runs concurrently with each of the 10 benchmarks. The first column lists the benchmarks. Columns 2 and 3 present the increase in MPA and in SPI of each of the 10 benchmarks due to cache contention, compared to those when it runs alone. Column 4 shows the number of iterations required to solve for the effective cache size. Columns 5, 7, and 9 show the prediction errors for MPA for each of the three models. Columns 6, 8, and 10 show the prediction errors

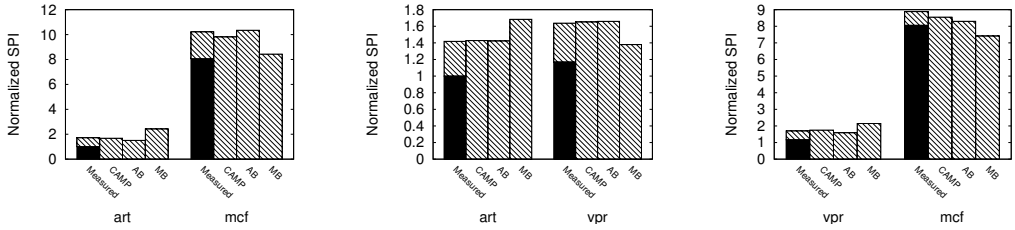

Figure 3.3: Performance degradation for (a) <art, mcf> pair, (b) <art, vpr> pair, and (c) <vpr, mcf> pair.

for SPI for each of the three models. The errors relative to measurements are reported. A positive error indicates an over-prediction and a negative error indicates an under-prediction. The last row shows the average results for all 10 cases.

Table 3.4 indicates that CAMP outperforms AB and MB in terms of both MPA estimation error and SPI prediction error. AB over-predicts the effective cache size of *art*, resulting all 10 under-predictions of cache miss rate and 9 over-predictions of SPI. It achieves an average SPI prediction error of 10.26% and a maximum error of 17.47%. MB under-predicts the effective cache size of *art*, resulting in 8 over-predictions of MPS. It achieves an average SPI estimation error of 18.48%. and a maximum error of 41.06%. In contrast, CAMP achieves an average estimation error of 3.68% and a maximum error of 7.15%. Note that the computation overhead of CAMP is also lower than that of AB and MB because it uses monotonic non-linear functions. This might significantly reduce computational cost when the number of cores is large. In addition, since the three models are based on estimating the effective cache sizes of two processes, they give the same results when two instances of *art* are running together, as indicated in the first row of Table 3.4.

We now explain why AB usually leads to over-prediction and MB usually leads to under-prediction of the effective cache size. Figure 3.3 illustrates the predicted and measured normalized SPIs. The black portion shows the SPI when benchmark is run alone. Figure 3.3(a) shows the results when benchmarks *art* and *mcf* share cache in a dual-core system, with the left part corresponding to *art* and the right part corresponding to *mcf*. We denote this scenario as <*art*, *mcf*>. Similarly, Figure 3.3(b) represents <*art*, *vpr*>, and Figure 3.3(c) represents <*vpr*, *mcf*>. As indicated in Figure 3.3, CAMP achieves the best accuracy in all three cases. We take the left figure as an example

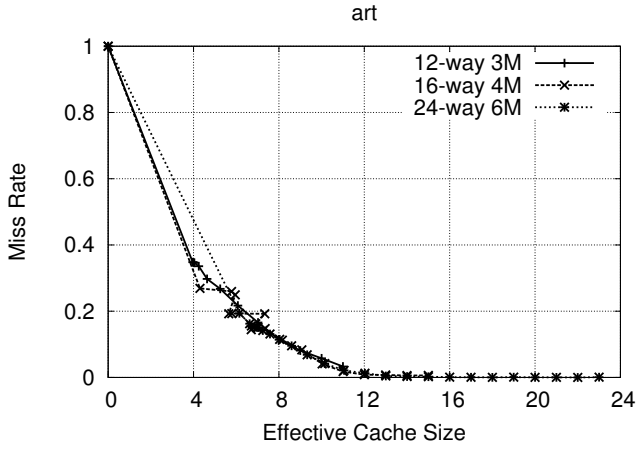

Figure 3.4: Profiled cache miss rate corresponding to effective cache size for different cache configurations.

to explain the reason for variation in accuracy. As indicated in Figure 3.2, given the same effective cache size, *mcf* has a higher miss rate than *art*, resulting in larger SPI than *art*. Therefore, the APS of *art* is approximately twice that of *mcf* when they run concurrently, even though the API of *mcf* is larger than that of *art*. Thus, *art* has a high APS with low MPS, which indicates that *art* can access the cache very frequently with low reuse distances, resulting in few misses. In this case, MB tends to over-predict the performance of *mcf* because it ignores factors such as APS. On the other hand, AB overestimates *mcf*'s performance due to ignoring its high reuse distances. Note that when two processes share the cache in a dual-core system, under-predicting the performance of one leads to over-predicting the performance of the other. CAMP takes both APS and MPS into consideration, and therefore is most accurate.

3.5.4 Generality of Predictor For Different Machines

Figure 3.4 shows the cache miss rate of *art* corresponding to effective cache size profiled under two other cache configurations differing from that in Figure 3.2. CAMP was also validated on two other Intel Core 2 Duo Processors with 4 MB and 6 MB of L2 unified cache. The three cache miss rate curves closely match each other, suggesting that process characterization data derived on one machine might be used to accurately

predict the performance of cache-sharing processes on different types of processors with different cache structures.

3.6 Conclusion

Cache contention among processes running on different CMP cores heavily influences performance. A cache-contention aware assignment algorithm can help improve system throughput and reduce power consumption. However, this requires a model of cache contention behavior that can quickly and accurately determine the impact of different assignments on performance. This is challenging due to the large numbers of potential assignments of processes to CMPs. We have described CAMP, a predictive model that allows fast and accurate estimation of system performance degradation due to cache contention. More specifically, it first determines a process-dependent feature vector and reuse distance histogram via (potentially on-line) pre-characterization. The feature vectors of cache-sharing processes are supplied into a group of non-linear equations that determine the steady-state effective cache size and performance of each process. We also described a method to automate the profiling and performance prediction process. We evaluated the proposed technique on 55 different combinations of 10 SPEC CPU2000 benchmarks on a dual-core machine. The average performance prediction error is 1.57%. We also tested the generality of the proposed technique by profiling processes on one CMP and using the profiling information for performance prediction on two other CMPs with different cache sizes. In contrast with existing work, the proposed approach requires access only to information that is readily available from processor performance counters.

Chapter 4

Power Modeling for CMPs

In the previous chapter, we described CAMP, a shared cache aware performance model for CMPs. By taking advantage of the hardware performance counters, which are available on most modern processors, we can automate the profiling process and gather process-dependent characteristics such as reuse distance histograms, cache access frequencies, and the relationship between the throughput and cache miss rate of each process without exhaustive simulation or modification to the underlying hardware and software infrastructure. CAMP uses those inputs to predict its effective cache size when running concurrently and sharing cache with other processes, allowing instruction throughput estimation. In addition, we demonstrated the generality of the proposed technique by profiling processes on one CMP and using the profiling information for performance prediction on two other CMPs with different cache sizes.

However, to permit an efficient power-aware scheduling and management scheme in a multi-programmed multi-core computing platform, power modeling is another critical building block. In addition, such power model can be easily integrated into our existing performance model.

This chapter describes a fast, automated technique for accurate on-line estimation of the power consumption of interacting processes in a multi-programmed, multi-core environment. The proposed technique does not require modifying hardware or applications. The system-level power model is derived using multi-variable linear regression, accounting for cache contention. We validated the power model on multiple real multi-core systems using SPEC CPU2000 benchmarks to demonstrate the generality of the

proposed model. Finally, we integrate the power model with CAMP to estimate processor power for any tentative assignment without any runtime information. The combined model is validated on a 4-core server using SPEC CPU2000 benchmarks, with an average estimated error within 3.5%. This work is done in collaboration with other researchers. In particular, Xi Chen was the leader on designing and evaluating the power model, the author of the dissertation was responsible for combining the performance and power model, and evaluating results on one of the evaluation platforms.

4.1 Introduction and Motivation

Power modeling in a multi-programmed single-core environment is challenging due to issues such as time sharing among processes. The on-going move to chip multiprocessors (CMPs) permits sharing the last-level cache among cores on the same die but this aggravates the cache contention problem: processes running simultaneously on cache-sharing cores contend for the limited space in the last-level cache, impacting performance and power consumption, which further complicates the modeling problem. Accurately modeling the performance and power consumption in a multi-programmed multi-core environment is necessary for design-time architectural optimization and run-time dynamic resource management [28, 29].

Power modeling in a multi-programmed multi-core environment presents several challenges: (1) the models should be easy to construct without modifications to existing software or hardware. Exhaustive off-line simulation of all process combinations is computationally intractable and thus should be avoided; (2) the models should handle time sharing among processes on the same core and resource contention among processes on cache-sharing cores; and (3) to be useful in on-line process assignment, the models must estimate power and throughput before processes are assigned. To the best of our knowledge, no existing performance and/or power models satisfy the requirements mentioned above.

This chapter makes the following contributions: (1) we propose a modeling framework that generates fast, accurate, on-line estimates of power consumption for any process-to-core mapping during runtime; (2) the system-level power model can handle time sharing among processes on the same core and cache contention among processes

on cache-sharing cores; (3) this is the first work to estimate the processor power for any tentative assignment without run-time information by integrating the performance model and the power model; and (4) our models are general enough to accommodate heterogeneous tasks and processors. Both models have been validated on different machines with different architectures and nominal power consumptions. Note that although constructing a performance model requires profiling each process of interest, this does not limit the generality of our approach because profiling can be done on-line. When a new application makes up a significant percentage of the workload, we force it to run alone on an idle machine and record profiling information. Therefore, the approach can be used (in different ways) for both embedded and general-purpose computing systems.

4.2 Related Work

Researchers have also developed simulation-based power models [28]. However, such models impose significant performance overhead and are therefore inappropriate to use during runtime. Other researchers have proposed performance-counter-based power models for on-line power estimation [30, 29]. However, such models only estimate the power consumption of a single application; it is not straightforward to extend them for power estimation in a multi-programmed, multi-core environment. Singh *et al.*proposed a performance counter based power model in a multi-programmed CMP environment [31]. This work is related to ours. However, their power model construction process is ad hoc and requires the user manually tune the model parameters and fitting functions. In addition, their power model cannot handle time sharing among processes on the same core. In contrast, the model building process for our power model can be fully automated. As demonstrated in section 5.4, it can handle time sharing among processes and applies to CMP systems with different architectures without any changes to the model construction process.

4.3 Power Modeling

In this section, we first formulate the power modeling problem. We then explain the model construction process. Finally, we describe how we handle time sharing among

processes sharing cores and cache contention among processes running on multiple cores.

4.3.1 Problem Formulation

The power modeling problem in a multi-programmed multi-core environment can be formulated as follows: given k processes running on N cores with some of the cores having multiple processes and some of them being idle, estimate the core and processor power consumption during concurrent execution.

It is natural to decompose core power consumption into idle power consumption and the active power consumptions of individual architectural blocks. Given that there are M components in a system, the total power consumption is $P = P_{\text{idle}} + \sum_{i=1}^{M} P_i$, in which P_{idle} is the idle power consumption when no process is actively using the core and P_i is the power consumption of component i. In order to make online estimates of P_i, we again use HPCs: by carefully choosing the HPC-detected hardware events monitored, we can map an event rate, i.e., number of events per second, to the power consumption of the corresponding architectural block. We first choose the HPC event rates that are most correlated to core power consumption. We omit the details here due to space limitations. The top 5 event rates with the highest correlation coefficients are L1RPS, L2RPS, L2MPS, BRPS, and FPPS, which represent the number of L1 data cache references per second, number of L2 cache references per second, number of L2 cache misses per second, number of floating point instructions retired per second, and number of branch instructions retired per second, respectively.

It remains unclear how to map the event rates to the corresponding component power: the power consumption of a component may be nonlinearly dependent on the event rate associated with it. We first wrote a micro-benchmark with 6 phases, each of which lasts 80 s. In the first phase, the core idle power is recorded, whereas one of the aforementioned 5 architectural blocks are explicitly accessed in each of the following 5 phases. Note that the access frequency is the highest at the start of a phase and reduced to a lower level every 10 s, i.e., there are 8 different access frequencies for one component in one phase. We then use 8 SPEC CPU2000 benchmarks (see section 5.4) and the micro-benchmark for model construction. Given an N-core processor, we run N instances of one benchmark on N cores (one instance per core) and gather the HPC values along with the processor power throughout the execution, assuming each core

has the same power and HPC values. We then evaluate the modeling results based on two different algorithms, the multi-variable linear regression (MVLR) algorithm and a three-layer sigmoid activation function neural network (NN). Experimental results indicate that the MVLR-based model achieves an accuracy of 96.2% while the NN-based model reaches an accuracy of 96.8%. Given an accuracy comparable to NN-based model and the simplicity in model construction and evaluation, MVLR-based model is chosen. Hence, the core power P_{core} can be expressed as

$$\begin{aligned} P_{\text{core}} &= P_{\text{idle}} + c_1 \cdot \text{L1RPS} + c_2 \cdot \text{L2RPS} + c_3 \cdot \text{L2MPS} + \\ &\quad c_4 \cdot \text{BRPS} + c_5 \cdot \text{FPPS}, \end{aligned} \tag{4.1}$$

where P_{idle} and c_1 through c_5 are coefficients determined from MVLR.

4.3.2 Handling Context Switching and Cache Contention

The proposed power model can accurately estimate the core power consumption when a single process is running. However, there are usually multiple processes running on the same core in a multi-programmed environment, limiting the usability of the power model. We define *process power consumption* as the core power consumption when this process is running. Since we assume there are no data dependencies among processes, the major interactions among processes on the same core are contention for resources such as cache. We experimentally determined the average amount of time required to fill the cache after a context switch is only 1% of the timeslice length given a 20 ms timeslice, which indicates the impact of context switches on performance and power is negligible. Therefore, the core power consumption is the linear weighted sum of all process power consumptions with the timeslice length of each process being its weight. In reality, we make the simplifying assumption that every process has the same weight. Hence, assuming there are k processes running on the single core with process i's power consumption being P_i, the core power consumption is simply $P_{\text{core}} = \frac{1}{k}\sum_{i=1}^{k} P_i$.

We now define the *processor power consumption* as the sum of all core power consumptions in a multi-core multi-programmed environment, in which cache contention problem becomes more severe. On one hand, increased cache contention leads to lower processor power consumption because c_3 is negative in Equation 4.1. On the other hand, increased resource utilization implies higher processor power consumption. The amount

of increase in processor power consumption depends on the balance between the two factors. This is consistent with our experimental results (see section 5.4). Therefore, the proposed power model can handle the multi-core environment without any modifications. If there is more than one process per core, given core 1 through core N share the last-level cache and S_i is the set of processes running on core i, the average power consumption of these cores $P_{\text{core-set}}$ can be calculated as

$$P_{\text{core-set}} = \frac{\sum_{p_1 \in S_1} \cdots \sum_{p_N \in S_N} P(p_1, p_2, \cdots, p_n)}{\prod_{i=1}^{N} |S_i|}, \tag{4.2}$$

where $P(p_1, p_2, \cdots, p_n)$ is the sum of power consumptions of core 1 through core N when processes p_1, p_2, $\cdots$, p_n run simultaneously.

4.4 Combining Performance and Power Models

In this section we describe how to combine the proposed performance and power models for use in optimization. One such application is power-aware assignment. More specifically, if we can accurately estimate the processor power consumption for each tentative assignment decision, we can choose the one that optimizes power or energy usage. However, such power estimation is usually impossible because the HPC values needed for power estimation are unknown until the processes are assigned. Nonetheless, by integrating the performance model and the power model, we are able to estimate the process power consumption for each assignment, as explained below.

Given the power model in Equation 4.1, we can decompose the process power P_{process} into two parts:

$$\begin{aligned} P_1 &= P_{\text{idle}} + (c_1 \cdot \text{L1RPI} + c_2 \cdot \text{L2RPI} + \\ &\quad c_4 \cdot \text{BRPI} + c_5 \cdot \text{FPPI})/\text{SPI}, \\ P_2 &= c_3 \cdot \text{L2MPS} = c_3 \cdot \text{L2MPR} \cdot \text{L2RPI}/\text{SPI}, \text{and} \\ P_{\text{process}} &= P_1 + P_2. \end{aligned}$$

Here, P_{idle} is the power consumption of an idle core, L1RPI represents the number of L1 data cache accesses per instruction, L2RPI represents the number of L2 cache references per instruction, BRPI represents the number of branches per instruction, FPPI represents the number of floating point instructions retired per instruction, and

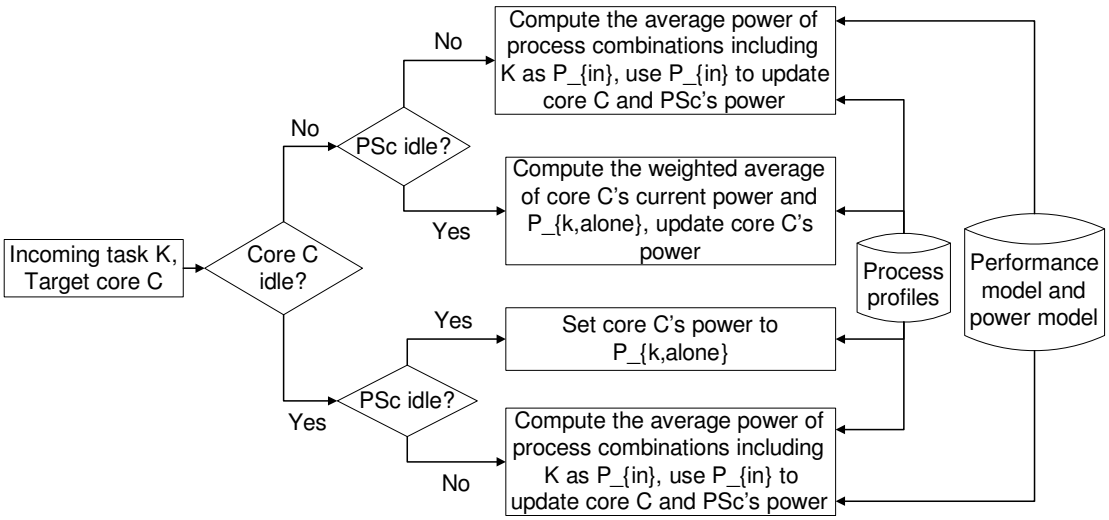

Figure 4.1: Algorithm for power estimation for process assignment.

L2MPR represents the number of L2 cache misses per L2 cache reference. We define a *instruction-related event rate* as the number of events per instruction. L1RPI, L2RPI, BRPI, and FPPI in P_1 are process properties: given the same input data, these instruction-related event rates are fixed and not affected by the execution of other processes. Therefore, the impact of cache contention is only reflected in the change of SPI. However, P_2 is not only influenced by SPI but also L2MPR. Fortunately, both SPI and L2MPR can be determined by the performance model given enough profiling information, as explained in section 3.3. Hence, if we record the instruction-related event rates during profiling for each process and use performance model in section 3.3 to predict SPI and L2MPR whenever cache contention exists, we can estimate P_1, P_2, and thus the process power.

We first assume the performance and power model are built as described in section 3.3 and section 4.3. We also assume for each process i, the profiling vector PF_i, i.e., ($P_{i,\text{alone}}$, L1RPI_i, L2RPI_i, BRPI_i, FPPI_i) is recorded during profiling. Note that $P_{i,\text{alone}}$ represents process i's average power consumption when it runs alone with no other active processes. Figure 4.1 illustrates how to combine the performance model, power model, and process profiles for power estimation during assignment. Suppose we want to evaluate the resulting power consumption by assigning process K to core

C. We denote the set of cores that share the last-level cache with core C as core C's partner set PS_C. Depending on the states of core C and PS_C, there are four different outcomes: (1) both C and PS_C are idle, (2) C is busy and PS_C is idle, (3) C is idle and PS_C is busy, and (4) both C and PS_C are busy. We only analyze scenario (1) and scenario (4) since scenarios (2) and (3) are special cases of scenario (4). In scenario (1), we set core C's power consumption to $P_{K,alone}$, fetched from profiling vector PF_K. The processor power consumption is also increased by $P_{K,alone}$. In scenario (4), we assume there are N cores in PS_C numbered from 1 to N, among which core 1 through core m have processes running on them and core $m+1$ through core N are idle. For convenience, we use S_i to represent the set of processes running on core i. We define a *process combination* as an ordered tuple (PC_C, PC_1, PC_2, $\cdots$, PC_m) where $PC_C \in S_C, PC_1 \in S_1, \cdots, PC_m \in S_m$, indicating processes PC_C, PC_1, PC_2, $\cdots$, PC_m run simultaneously on core C and its partners core 1 through core m. For the set of process combinations that do not include process K, denoted as S_{ex}, the average power consumption, denoted by P_{ex}, is the sum of current power consumptions of core C and cores in PS_C. On the other hand, if we use S_{in} to represent the set of process combinations that include process K, for each item I in S_{in}, we use the performance model to predict the SPI and L2MPS for each process that belongs to I, which are then fed into the power model to calculate the corresponding power consumption for the process combination I. We use P_{in} to denote the average power consumptions for all combinations in S_{in}. Hence, the processor power consumption $P_{\text{processor}}$ can be written as

$$P_{\text{processor}} = (N-m) \cdot P_{\text{idle}} + \frac{P_{\text{ex}} \cdot |S_{\text{ex}}| + P_{\text{in}} \cdot |S_{\text{in}}|}{|S_{\text{ex}}| + |S_{\text{in}}|} + P_{\text{rest}}, \tag{4.3}$$

where P_{rest} is the current power consumption of cores that do not share cache with core C. Therefore, by profiling each process individually, we are able to estimate the processor power consumption for any process-to-core mapping, reducing the exponential time complexity for a simulation based approach to linear time complexity.

4.5 Experimental Results

In this section, we first describe the experimental setup for model validation. We then present the validation results for the performance model, the power model, and the

Table 4.1: Power Model Validation on a 2-Core Workstation

Scenarios	Number of assignments	Avg./max. error for power samples (%)	Avg./max. error for avg. power (%)
1 proc./core	36	5.32 / 14.12	3.63 / 13.83
2 proc./core	24	6.65 / 8.84	2.47 / 4.05

Table 4.2: Power Model Validation on a 4-Core Server

Scenarios	Number of assignments	Avg./max. error for power samples (%)	Avg./max. error for avg. power (%)
1 proc./core	24	4.09 / 8.52	3.26 / 7.71
2 proc./core	3	5.51 / 6.25	4.47 / 5.95
4 proc. with unused cores	10	3.39 / 4.73	2.54 / 4.14

combined model.

4.5.1 Experimental Setup

We use PAPI 3.6.2 [32] to sample the HPCs. The sampling period is 30 ms. Our testsuite includes 8 SPEC CPU2000 benchmarks that compiled on the test system using gcc 4.1. This set contains both memory-intensive and CPU-intensive benchmarks. We record the program phase information for each benchmark during profiling. Experimental results indicate all but two benchmarks have only one significant phase, as defined by our parameters of interest. The longest phases in *art* and *mcf* were used (refer to Tam *et al.* [24] for details).

To determine power consumption, we use a Fluke i30 current clamp on one of the 12 V processor power supply lines, the output of which is sampled by an NI USB6210 data acquisition card. An on-chip voltage regulator converts this voltage to the actual processor operating voltage. We assume a fixed regulator efficiency of 90%. Therefore, $P = 0.9V \cdot I = 10.8 \cdot I$, where P is the processor power and I is the measured current. The data acquisition card samples at a frequency of 10 kHz in our experiments.

4.5.2 Power Model Validation

We validated our power models on (1) a Pentium Dual Core E2220 processor with 1 MB L2 cache, which runs Linux 2.6.25 and (2) a 4-core server. For each machine, we first build the power model using 8 SPEC CPU2000 benchmarks and the customized micro-benchmark as explained in subsection 4.3.1. We then validate the power model

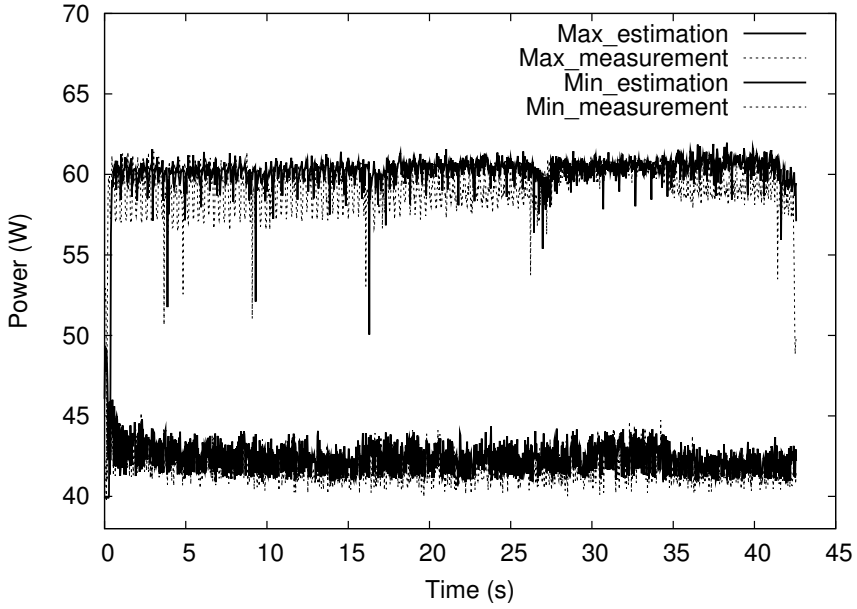

Figure 4.2: Power model validation on 4-core server.

by assigning a combination of several SPEC CPU2000 benchmarks to some or all of the cores and compare the real power consumption with the power estimations using HPC values gathered during runtime. Note that we only analyze the duration in which all processes assigned are running concurrently.

Figure 4.2 illustrates the sample-based power model validation on the 4-core server for the assignments with the maximum and the minimum average power among all test cases. The X axis is time and the Y axis is the power consumption. The solid lines represent power estimations, while the dotted lines represent measured values. They generally overlap, indicating good estimation accuracy. The average estimation errors are 2.46% and 2.51% for the maximum-power scenario and the minimum-power scenario, respectively.

Table 4.1 and Table 4.2 show the validation results for the power model on the 2-core workstation and 4-core server, respectively. Column 1 shows the testing scenario, e.g., "1 proc./core" refers to assignment schemes in which all cores are used with one SPEC program per core. Column 2 represents the number of different assignments evaluated given the testing scenario indicated in Column 1. Note that the processes

Table 4.3: Validating the Combined Model on a 4-Core Server

Scenarios	Number of assignments	Avg./max. error for avg. power (%)
1 proc./core	32	2.84 / 5.78
2 proc./core	10	1.92 / 6.29
4 proc., 1 core unused	16	2.68 / 5.48
4 proc., 2 core unused	16	2.53 / 5.99
4 proc., 3 core unused	9	0.49 / 1.95

in each assignment are chosen randomly in order to test the model on a wide range of scenarios. Column 3 presents the average and maximum error resulting from comparing the estimated processor power with the measured power for all power estimation samples. Column 4 presents the average and maximum error resulting from comparing the estimated average power with the measured average power.

On the 2-core workstation, we tested 36 different assignments with 1 process per core and 24 assignments with 2 processes per core. For a sample-based comparison, the average error for both scenarios are 5.32% and 6.65%, with maximum errors of 14.12% and 8.84%. For an average-power–based comparison, the average error for both scenarios are 3.63% and 2.47%, with maximum errors of 13.83% and 4.05%.

On the 4-core server, we tested 24 different assignments with 1 process per core, 3 assignments with 2 processes per core, and 10 assignments with 1 or 2 cores unused. For a sample-based comparison, the average error for the three scenarios are 4.09%, 5.51%, and 3.39%, with maximum errors of 8.52%, 6.25%, and 4.73%. For an average power comparison, the average errors for the three scenarios are 3.26%, 4.47%, and 2.54%, with maximum errors of 7.71%, 5.95%, and 4.14%. Therefore, we conclude the proposed power model is accurate and is sufficiently general to be used for different architectures, although the limited number of architectures considered is not sufficient to determine the were the limits on generality are located.

4.5.3 Combined Model Validation

We validated the combined performance and power model for average power estimation during assignment on the 4-core server. We first built the performance model and the power model as explained in sections 3.3 and 4.3. We then estimated the power consumption of an assignment following the algorithm in Figure 4.1. Note that only profiling information are used for estimation. The estimated average power is then

compared to the measured average power to determine the accuracy of the combined model.

We tested 32 assignments with 1 process assigned to each core, 10 assignments with 2 processes assigned to each core, 16 assignments with 4 processes assigned to 3 cores, 16 assignments with 4 processes assigned to 2 cores, and 9 assignments with 4 processes assigned to a single core. The average errors for the 5 scenarios were 2.84%, 1.92%, 2.68%, 2.53%, and 0.49%, while the maximum errors were 5.78%, 6.29%, 5.48%, 5.99%, and 1.95%. We thus conclude that the combined model is effective in estimating the processor power consumption during assignment.

4.6 Conclusions

Accurately modeling the performance and power consumption in a multi-programmed multi-core environment is challenging but essential for optimizing process assignment and migration. This chapter describes an on-line performance and power modeling framework that rapidly and accurately estimates the power consumption and performance implications of particular process-to-core mappings. This process requires no changes to existing operating system or hardware. The individual models and the combined model have been validated on multiple CMP machines with distinct architectures and nominal power consumptions. We conclude that the proposed framework is effective for performance and power estimation during both process assignment and execution.

Chapter 5

Memory access aware on-line voltage control for performance and energy optimization

In this chapter, we will explore the impact of the memory hieracrchy on one of the most significant metric: energy. We proposed an off-chip memory access-aware runtime DVFS control technique that minimizes energy consumption subject to a constraint on application execution time. We consider application phases and the implications of changing cache miss rates on the ideal power control state. We first propose a two-stage DVFS algorithm that formulates the throughput-constrained energy minimization problem as a multiple-choice knapsack problem (MCKP), assuming a priori (oracle or profiling-based) knowledge to an application's behavior. This algorithm builds on an application phase-dependent power model, taking advantage of processor hardware performance counters. Solutions to this problem provides an upper bound on the energy savings achievable under a performance constraint. We then propose P-DVFS, an DVFS algorithm for on-line minimization of energy consumption under a performance constraint during runtime without requiring a priori knowledge to an application's behavior. In addition to the power model, P-DVFS also relies on a performance model that characterizes the performance of a running application using hardware performance counters. It predicts remaining execution time during runtime in order to optimize voltage and frequency for

the best application energy consumption and performance results. Like the two-stage DVFS algorithm, P-DVFS supports formulation as a multiple-choice knapsack problem, which can be efficiently and optimally solved online. We evaluated P-DVFS using direct measurement of a real DVFS-equipped system. When bounding performance loss to at most 20% of that at the maximum frequency and voltage, P-DVFS leads to energy consumptions within 1.83% of the optimal solution on average with a maximum deviation of 4.83%. The most advanced existing DVFS control algorithm results in energy consumptions with 9.8% average deviation and 29.86% maximum deviation from optimal. In addition to producing results approaching those of an oracle formulation, P-DVFS reduces power consumption by 9.93% on average, and up to 25.64%, compared with the most advanced existing work.

5.1 Introduction and Related Work

Energy consumption is important in both portable computer systems, due to its impact on battery lifespan, and high-performance stationary computers, due to its impact on energy and cooling costs. Prior work has considered minimizing processor energy consumption. Chang *et al.*proposed a dynamic programming energy minimization technique for multiple supply voltage scheduling in both pipelined and non-pipelined datapaths [33]. Zhang *et al.*developed a two-phase technique that integrates task assignment, task scheduling, and voltage selection for energy minimization [34]. Varatkar *et al.*proposed communication-aware task scheduling and voltage selection to minimize the overall system energy consumption in a multiprocessor environment [35]. However, the goal of these techniques is to minimize energy without affecting performance; trade-offs between performance and energy consumption were not considered.

Other researchers have considered power management mechanisms that trade off performance for power consumption. One of the most promising of these is dynamic voltage and frequency scaling (DVFS). A well-designed DVFS control policy can reduce system energy consumption while maintaining the same or better performance than alternative control policies. This requires a policy with two important characteristics: (1) a well-designed DVFS control policy must model and react to the dynamically changing trade-offs between application performance and power consumption. A reduction

in processor voltage and frequency has very different energy and performance impacts on applications that are heavily accessing off-chip memory, and those that are consistently hitting in cache, and therefore have performance constrained only by the current frequency of the processor. A well-designed DVFS policy must continuously monitor and adapt to the behavior of applications. (2) If a DVFS control policy is to guarantee that a particular application consistently runs with adequate performance, e.g., adheres to an instruction throughput constraint, it should maximize energy consumption savings by predicting the distribution of future instructions among different memory access behaviors categories. This allows the control policy to increase processor voltage and frequency when the performance benefit per lost energy unit is the highest and reduce frequency and voltage when the energy benefit per lost performance unit is the highest.

A number of researchers have worked on DVFS-related control to optimize power and energy consumption. Isci *et al.*proposed a runtime phase monitoring and prediction technique to reduce power consumption using DVFS [36]. However, this technique does not bound performance degradation. Wu *et al.*proposed dynamic compiler driven DVFS for controlling microprocessor energy and performance [37]. However, their work requires changes to the underlying compilation infrastructure. In addition, their technique cannot guarantee that performance requirements will be met. Liu *et al.*proposed a technique to optimize peak temperature subject to a performance constraint using DVFS in a real-time system [38]. However, their assumption that the execution time of a task is inversely proportional to CPU frequency is incorrect, as we will demonstrate in subsection 5.2.1. The technique proposed by Choi *et al.*is the closest to ours [39]. The goal of their technique is to minimize energy consumption under a constraint on the total execution time of a program. Detailed comparisons with their work can be found in subsection 5.4.2. Their DVFS policy considers the impact of application phases and off-chip memory accesses. However, it considers only immediate application behavior instead of adaptively controlling power state using predictions based on long-term behavior history.

Our work differs from prior work in the following main ways.

1. We propose a two-stage DVFS algorithm that allows us to formulate the throughput-constrained energy minimization problem as an MCKP problem, solve it optimally, and use the solution to guide online frequency and voltage control. This algorithm

builds on an application phase-dependent power model, taking advantage of processor hardware performance counters. The solutions obtained using the two-stage algorithm determine the optimal energy savings under a performance degradation ratio. However, it assumes access to oracle or profiling-based information about application behavior.

2. We also propose P-DVFS, a predictive online DVFS algorithm that requires no a priori knowledge of application behavior. P-DVFS uses power and performance models that use hardware performance counters to adapt to the behaviors of running application. It predicts remaining execution time online in order to control voltage and frequency to minimize energy consumption under application-level performance constraints. Like the two-stage oracle DVFS algorithm, P-DVFS is also formulated as a multiple-choice knapsack problem. This formulation permits rapid, optimal, on-line solution of real problem instances.

3. In contrast with all related work, except that of Choi *et al.* [39], we consider the dependence of the power consumption performance tradeoffs available via DVFS upon application memory access behavior, i.e., phase. By adapting to application phase, our technique supports more aggressive power management settings when doing so has the least negative performance impact. To this end, we describe a method of modeling the performance and power consumption of the processor using built-in hardware performance counters.

4. In contrast with all past work, our problem formulation supports application-level throughput requirement, not instantaneous instruction throughput requirement. This is supported by on-line monitoring of application behavior as well as prediction of application run times.

We evaluated P-DVFS via direct measurement during operation on a real system. When limiting performance loss to at most 20% of that possible at the maximum frequency and voltage, P-DVFS leads to energy savings within 1.83% of the optimal solution on average with a maximum deviation of 4.83%. It improves energy consumption by 9.8% on average, and up to 29.86%, compared to the most advanced existing DVFS control technique. P-DVFS also reduces power consumption by up to 25.64% (9.93% on average) compared with the most advanced prior work.

5.2 Motivation and Problem Formulation

In this section, we first describe how the trade-offs between performance and energy consumption change depending on application off-chip memory access behavior. We then present the problem formulation for energy minimization given a user-specified constraint on application execution time. Finally, we present a dynamic power state control policy that adjusts CPU frequency based on off-chip memory access patterns.

5.2.1 Trade-offs Between Performance and Energy

The execution time of a task can be decomposed into on-chip latencies and off-chip latencies. On-chip resource use associated on-chip latencies scale linearly with CPU frequency, because the on-chip resources share the same clock with the processor. In contrast, off-chip latencies, caused by accesses to off-chip resources such as main memory and disk, are independent of CPU frequency, because the off-chip resources have their own clocks.

The power consumption of a task can be divided into dynamic power and static power. Dynamic power consumption is caused by transistor switching activities. It generally scales superlinearly with the CPU clock frequency of the computing system [40]. Static power consumption is primarily due to the gate and subthreshold leakage currents of transistors. It is independent of the CPU frequency but depends on the voltage. In general, reducing frequency and voltage reduces both dynamic and static power consumption.

Most modern processors are equipped with dynamic voltage and frequency scaling (DVFS) capability. The typical voltage change overhead for our evaluation platform is 50 µs. Given an application with some phases in which instruction throughput is limited largely by processor core performance and other phases in which instruction throughput is limited largely by (processor frequency independent) off-chip memory access latency, we can maximize energy consumption improvement and minimize performance overhead by using a low CPU frequency during memory-bound application phases and a high CPU frequency during core-bound application phases. What temporal granularity should this control use? The DVFS switching overhead of 50 µs (see section 5.4) implies that adjustments should happen no more frequently than once per hundreds of microseconds,

thus limiting overhead.

5.2.2 Problem Formulation

The performance-constrained energy minimization problem can be formulated as follows: Given that α is the user-specified performance degradation ratio relative to the maximum performance of a given task and T_{fmax} is the execution time of the task running at the highest frequency, find the optimal CPU frequency as a function of time t such that the total energy consumption of the task is minimized and the actual execution time of the task subject to DVFS is no larger than $(1 + \alpha)T_{fmax}$. Note that this constraint is a soft timing constraint, i.e., it is highly desirable to meet the constraint. However, violating the constraint does not mean failure: a cost function may be associated with difference between the constraint and the actual execution time.

As indicated in subsection 5.2.1, the energy saving potential directly relates to the proportion of total execution time resulting from waiting on off-chip data access. In our experiments, L2 cache misses are the dominant type of off-chip access. We assume that each L2 cache miss takes the same amount of time. Hence, the number of L2 cache misses per instruction (MPI), is a good indicator of the potential for saving energy. Intuitively, it is beneficial to assign higher frequencies fir intervals with low MPIs to improve performance and lower frequencies for intervals with high MPIs to save energy. It is thus natural to use MPI distribution variation and assign different frequencies depending on the MPIs.

In real operating systems, power control policies are usually implemented using adjustments at discrete time intervals. We discretize the MPI values and pack them into different MPI slots, each of which has a distinct nominal MPI value. We define a *control point* as the time at which control decisions are made and a *scaling point* as the time at which the CPU frequency is modified. The *control period* is the duration between two consecutive control points and the *scaling period* is the duration between two consecutive scaling points. Note that these periods need not be the same. In fact, it is reasonable to use a much larger control period than scaling period to minimize performance overhead incurred by the controller.

Given an MPI distribution within a control period, we denote the set of all MPI slots with S and the set of all available frequency levels with F. Our goal is to find the

correct frequency level f_i for each slot $i \in S$ such that the total energy consumption E_{total} is minimized and the actual execution time T_{act} satisfies $T_{act} \leq (1+\alpha)T_{fmax}$. Therefore, assuming the distribution is independent of frequency, for each $i \in S$ with frequency f_i, given that $\mathrm{SPI}_i(f_i)$ is the number of seconds per instruction at frequency f_i, $P_i(f_i)$ is the power consumption, and poi_i is the percentage of instruction associated with slot i, the objective function and the constraint can be expressed in terms of total number of instructions I_{total} and total energy consumption E_{total}. The formulation thus follow.

$$E_{total} = I_{total} \cdot \sum_{i \in S} P_i(f_i) \cdot poi_i \cdot \mathrm{SPI}_i(f_i) \text{ and} \tag{5.1}$$

$$T_{act} \leq (1+\alpha)T_{fmax}. \tag{5.2}$$

The problem is to minimize E_r subject to Equation 5.2. Since the DVFS switching overhead ranges from 50 µs to 200 µs, the performance (or energy) overhead due to a switch in frequency is less than 0.7%, given a scaling period of 30 ms. Therefore, we ignore its impact in our problem formulation. Note that $P_i(f_i)$ in Equation 5.1 depends on both the CPU frequency and process behavior such as number of last-level cache misses per second (see subsection 5.3.2).

5.3 System Modeling

In this section, we first explain our task performance and power models. We then translate the energy minimization problem into a multiple-choice knapsack problem (MCKP) and solve it optimally, assuming we know the average SPI at the maximum frequency (SPI_{fmax}) and the exact MPI distribution throughout the program execution. We then relax our assumptions and propose an execution time predictor that is accurate when running at the highest frequency. This allows us to formulate the online DVFS problem again as an MCKP, which can be solved efficiently on-line. Finally, we explain the software system architecture used to control DVFS in order to accurately adjust the trade-off between performance and energy consumption.

5.3.1 Performance Modeling

Equation 5.2 requires a formula that accurately determines the relationship between SPI, MPI, and CPU frequency. Intuitively, the amount of time consumed per instruction can also be decomposed into on-chip latencies and off-chip latencies. On-chip latencies are inversely proportional to frequency, while the off-chip latencies, captured by MPI, are independent of frequency. Prior work has reached the same conclusion [36]. SPI can be expressed as

$$\text{SPI}(\text{MPI}, f) = c_1 \cdot \text{MPI} + c_2/f, \text{or equivalently,} \tag{5.3}$$

$$\text{CPI}(\text{MPI}) = c_1 \cdot f \cdot \text{MPI} + c_2, \tag{5.4}$$

where CPI is the number of cycles per instruction, f is the CPU frequency, and c_1 and c_2 are constants to be determined.

Most modern processors have built-in hardware performance counters (HPCs) that record information about architectural events, e.g., number of instructions retired and cache misses [26]. By gathering these two event counts, we can compute SPI and MPI during application execution. Therefore, given the last N data points reported by HPCs, we can determine c_1 and c_2 using linear regression. The relevant formulæ follow.

$$c_1 = \frac{N \cdot (\sum_{i=1}^{N} x_i \cdot y_i) - (\sum_{i=1}^{N} x_i) \cdot (\sum_{i=1}^{N} y_i)}{N \cdot (\sum_{i=1}^{N} x_i^2) - (\sum_{i=1}^{N} x_i)^2} \text{ and} \tag{5.5}$$

$$c_2 = \left(\sum_{i=1}^{N} y_i - c_1 \cdot \sum_{i=1}^{N} x_i \right) / N, \tag{5.6}$$

where x_i denotes the product of MPI and CPU frequency for the ith data point and y_i represents the CPI for the ith data point. Note that N should be carefully chosen such that it can capture changes in memory access pattern quickly and still support accurate regression-based modeling. In our experiments, varying N between 10 and 50 has insignificant impact on total energy consumption (a variation of 0.5% in total energy was observed). However, if N is smaller than 10, e.g., 4, we see an 4% energy consumption increase due to inaccuracies in the linear regression model. In our experiments, we set N to 20.

5.3.2 Power Modeling

Equation 5.1 indicates the necessity of having an accurate formula that describes the dependency between power consumption and MPI. Since an L2 cache miss takes a relatively long time to finish, intuitively the power consumption is higher for larger MPI values and smaller for lower MPI values. However, the power consumption also depends on other architectural events such as number of floating point instructions executed and number of L1 data cache accesses. We experimented with different combinations of HPC-detected architectural events. Experimental results indicate the following five events were sufficient to permit accurate estimation of overall power consumption: number of L1 data cache references per second (L1DPS), number of L2 cache references (L2PS), number of L2 cache misses per second (L2MPS), number of floating instructions executed per second (FPPS), and number of branch instructions retired per second (BRPS). As a first-order approximation, we assume each access to system components such as L1 caches and L2 cache consumes a fixed amount of energy. Therefore, the total power consumption is linearly dependent on these five events. In addition, the dynamic power consumption is nonlinearly dependent on CPU frequency [6]. Given that f is the CPU frequency, the power consumption P can thus represented as

$$\begin{aligned} P &= b_0 + b_1 \cdot \text{L1DPS} + b_2 \cdot \text{L2PS} + b_3 \cdot \text{L2MPS} + \\ &\quad b_4 \cdot \text{FPPS} + b_5 \cdot \text{BRPS} + b_6 \cdot f^{1.5}, \end{aligned} \tag{5.7}$$

where $b_i, i = 0, \cdots, 6$ are task-specific constants that can be determined during pre-characterization. Note that the exponent 1.5 is determined empirically to ensure a good modeling accuracy. It is worth mentioning that b_0 accounts for system idle power and leakage power. For example, the formula for mcf benchmark (see section 5.4) follows:

$$\begin{aligned} P &= 4.778 + 2.2864 \times 10^{-9} \cdot \text{ L1DPS} + 6.517 \times 10^{-8} \cdot \text{ L2PS} \\ &\quad - 3.596 \times 10^{-7} \cdot \text{ L2MPS} + 0.6342 \cdot \text{ FPPS} \\ &\quad - 3.136 \times 10^{-9} \cdot \text{ BRPS} + 4.308 \cdot f^{1.5}. \end{aligned} \tag{5.8}$$

For all the benchmarks we evaluated, the application-dependent power models have an average error of 6.67% and a maximum error of 12.2% across all four CPU frequencies. Note that if the processor has built-in power sensors [41], the pre-characterization

phase can be eliminated and the constants can be determined during execution using a regression-based approach such as that described in subsection 5.3.1.

5.3.3 Cost Minimization

This section describes the way in which the DVFS power management state control problem is formulated as a multiple-choice knapsack problem (MCKP). Given multiple sets, each containing multiple items, where each item is associated with a profit and a weight, MCKP requires the selection of one item from each set. The selection is optimal when the total profit is maximized and the total weight of the selected items is below a constraint. The DVFS problem instance can be converted into an MCKP by considering each potential frequency level to be an item. The weight of the item is the expected throughput at the associated frequency level. The profit of the item is the associated reduction in expected energy consumption compared to the highest energy at the highest frequency level. Note that depending on whether we have a priori knowledge to SPI_{fmax} and the MPI distribution throughout program execution, the DVFS problem instance can be formulated as different MCKP instances, as we explained in section 5.3.3 and section 5.3.3.

Cost Function

Equation 5.3 and Equation 5.7 can be substituted into Equation 5.1. For each slot $i \in S$ within a control period where S is the set of all MPI slots, SPI_i and P_i depend only upon the frequency level assigned to MPI slot i. However, both are nonlinear functions because both SPI and power consumption are nonlinear functions of CPU frequency. As a result, we face a nonlinear optimization problem, which cannot be efficiently solved online. Fortunately, the number of available frequencies in a processor is usually very limited (4 in our case). Therefore, we select the frequency values associated with each MPI slot from a small or moderate set F. Note that F may include any frequency value between the minimum and the maximum available CPU frequency, which can be approximated by switching between two adjacent available CPU frequency levels. For simplicity, F only consists of the available frequency levels for the chip used in our experiments.

We use a binary variable x_{ij} to indicate whether the frequency f_j is assigned to MPI slot i.

$$x_{ij} = \begin{cases} 1, & f_j \text{ is assigned to MPI slot } i \text{ and} \\ 0, & \text{otherwise.} \end{cases} \tag{5.9}$$

Note that $\sum_{f_j \in F} x_{ij} = 1, \forall$ slot $i \in S$. Therefore, for each slot $i \in S$, SPI_i can be expressed as follows.

$$\begin{aligned} \text{SPI}_i &= \sum_{f_j \in F} x_{ij} \cdot (c_1 \cdot \text{MPI}_i + c_2/f_j) \\ &= c_1 \cdot \text{MPI}_i + \sum_{f_j \in F} c_2/f_j \cdot x_{ij}. \end{aligned} \tag{5.10}$$

Since constants c_1, c_2, and F are known at the control point, Equation 5.10 can be simplified as follows.

$$\begin{aligned} \text{Letting } s_0 &= c_1 \cdot \text{MPI}_i, \\ s_j &= c_2/f_j, \forall f_j \in F \text{ and} \end{aligned}$$

$$\text{SPI}_i = s_0 + \sum_{j=1}^{|F|} s_j x_{ij}. \tag{5.11}$$

where $|F|$ denotes the number of elements in F. Similarly, the value of the five events in Equation 5.7 are also known at the control point. It is worth mentioning that the five event counts are also frequency dependent. We therefore normalize event count to instruction count instead of time. For example, for L1 data accesses, we record the number of L1 data cache accesses per instruction (L1DPI), which is independent of frequency. Hence, for MPI slot i with frequency f_j, we have

$$\text{L1DPS}_i(f_j) = \text{L1DPI}_i/\text{SPI}_i(f_j) \triangleq m_{ij,1}. \tag{5.12}$$

Similarly, we use $m_{ij,2}$, $m_{ij,3}$, $m_{ij,4}$, and $m_{ij,5}$ to represent $\text{L2PS}_i(f_j)$, $\text{L2MPS}_i(f_j)$, $\text{FPPS}_i(f_j)$, and $\text{BRPS}_i(f_j)$, respectively. If we define $w_0 = b_0$ and $w_{ij} = \sum_{k=1}^{5} b_i \cdot m_{ij,k} + b_6 \cdot f_j^{1.5}, \forall f_j \in F$, the power consumption for MPI slot i can be expressed as

$$P_i = w_0 + \sum_{j=1}^{|F|} w_{ij} x_{ij}. \tag{5.13}$$

Combining Equations 5.11 and 5.13, Equation 5.1 can be rewritten as follows.

$$E_{total} = I_{total} \sum_{i \in S} poi_i \cdot (w_0 + \sum_{j=1}^{|F|} w_{ij} x_{ij})(s_0 + \sum_{k=1}^{|F|} s_k x_{ik}). \tag{5.14}$$

Note that poi_i is known at the control point. In addition,

$$x_{ij} \cdot x_{ik} = \begin{cases} x_{ij}, \text{if and only if } j = k \text{ and} \\ 0, \text{otherwise.} \end{cases} \tag{5.15}$$

Therefore, Equation 5.14 can be simplified as follows.

$$\begin{aligned} \text{Letting } e_0 &= I_{total} \cdot w_0 s_0, \\ e_{ij} &= poi_i(w_0 s_j + w_{ij} s_0 + w_{ij} s_j) \text{ and} \\ E_{total} &= e_0 + \sum_{i \in S} \sum_{f_j \in F} e_{ij} x_{ij}. \end{aligned} \tag{5.16}$$

Performance Constraint – the Optimal Solution

We first assume that we have a priori knowledge of SPI_{fmax} and the MPI distribution throughout the program execution and demonstrate we can solve this problem optimally. This solution technique has two stages: profiling and evaluation. During profiling stage, we record the necessary information, e.g., SPI_{fmax} as well as the percentage of instructions and the hardware performance counter values for each MPI slot. This allows an optimal solution to the problem. During evaluation, we use the optimal solution obtained in the profiling stage to adjust the frequency dynamically to maximize energy savings without violating the performance constraint. Although this technique could be used directly if profiling-based application precharacterization were permitted, it yields valuable information even for a problem formulation using no a priori knowledge. The formulation we have just described can be viewed to compute the optimal solutions an oracle would yield. It therefore allows us to determine an upper bound on the energy savings given a particular performance constraint. We will later propose an on-line DVFS technique requiring no application precharacterization. We will evaluate the quality of this prediction-based technique, called P-DVFS, by comparing its results with those of the optimal oracle formulation just described.

Assuming the number of instructions associated with MPI slot i is denoted as I_i, Equation 5.2 can be rewritten as

$$\sum_{i \in S} \sum_{f_j \in F} I_i \cdot \mathrm{SPI}_i(f_j) \cdot x_{ij} \leq (1+\alpha) T_{fmax}. \tag{5.17}$$

Dividing both sides by I_{total}, we have

$$\sum_{i \in S} \sum_{f_j \in F} poi_i \cdot \mathrm{SPI}_i(f_j) \cdot x_{ij} \leq (1+\alpha) \mathrm{SPI}_{fmax}. \tag{5.18}$$

Although we can use Equation 5.3 to express SPI as a function of MPI and frequency, in reality we recorded $\mathrm{SPI}_i(f_j)$ during profiling to eliminate the impact of linear regression error on the quality of the optimal solution. More specifically, at each scaling point during profiling, the frequency is reduced to the next lowest level. When the frequency cannot be reduced further, we increase the frequency back to the highest level. This process is repeated until the program under profiling finishes. We then compute the average $\mathrm{SPI}_i(f_j)$ associated with each MPI slot i and each frequency f_j. Hence, we can treat $\mathrm{SPI}_i(f_j)$ as a constant k_{ij} here. Equation 5.18 thus becomes

$$\sum_{i \in S} \sum_{f_j \in F} poi_i \cdot k_{ij} \cdot x_{ij} \leq (1+\alpha) \mathrm{SPI}_{fmax}. \tag{5.19}$$

Noticing that I_{total} and e_0 are constants, this problem can thus be formulated as follows.

$$\text{Minimize } \textstyle\sum_{i \in S} \sum_{f_j \in F} e_{ij} x_{ij} \tag{5.20}$$

$$\text{Subject to } \textstyle\sum_{i \in S} \sum_{f_j \in F} poi_i \cdot k_{ij} \cdot x_{ij} \leq (1+\alpha) \mathrm{SPI}_{fmax} \tag{5.21}$$

$$x_{ij} \in \{0,1\},\ \textstyle\sum_{f_j \in F} x_{ij} = 1, \forall i \in S \tag{5.22}$$

Note that x_{ij} are binary integer variables and e_{ij}, poi_i, and $k_{i,j}$ are positive constants. Therefore, by scaling the constants with a large positive number, we can make the coefficients e_{ij}, poi_i, and $k_{i,j}$ and the right hand side of the constraint in Equation 5.21 all positive integers. Thus, the formulation can be treated as an multiple-choice knapsack problem (MCKP) [42]. We solve this problem optimally using "lp_solve", an existing integer programming solver [43]. We record the frequencies assigned to each MPI value in an $|S| \times |F|$ lookup table. During evaluation stage, we use the current MPI value to look up and adjust the frequency at each scaling point.

Performance Constraint – P-DVFS

For this formulation, we assume that the MPI distribution is unknown. However, our MPI distribution prediction technique relies on the similarity of present and future MPI distributions. It is known that most programs have repeated phases with periods ranging from 200 ms to 2 s [25]. Therefore, this assumption holds given a reasonable time span for gathering MPI values to build the distribution. We also discuss our solutions when used in two scenarios where the total number of instructions are (1) known and (2) unknown. In the rest of the chapter, we will use P-DVFS (*predictive DVFS*) to indicate the online predictive DVFS technique.

Since at each control point, information such as the number of instructions retired is known, it is natural to use the remaining number of instructions I_r and remaining energy consumption E_r instead of I_{total} and E_{total} in our problem formulation. We first note Equation 5.16 is still applicable, except that E_{total} and I_{total} should be replaced with E_r and I_r. Given that T_{elap} is the amount of time elapsed and T_r is the remaining execution time, Equation 5.2 can be written as

$$T_r = I_r \cdot \sum_{i \in S} poi_i \cdot \mathrm{SPI}_i(f_i) \leq (1+\alpha) T_{fmax} - T_{elap}. \tag{5.23}$$

Equation 5.3 allows us to rewrite left side of Equation 5.23 as

$$I_r \cdot \sum_{i \in S} poi_i \cdot \mathrm{SPI}_i(f_i) = I_r \cdot \sum_{i \in S} \sum_{f_j \in F} d_{ij} x_{ij}, \tag{5.24}$$

where $d_{ij} = poi_i / \left(c_1 \cdot \mathrm{MPI}_i + {}^{c_2}/_{f_j}\right), \forall f_j \in F$. Therefore, Equation 5.23 can be simplified as

$$\sum_{i \in S} \sum_{f_j \in F} d_{ij} x_{ij} \leq \frac{(1+\alpha) T_{fmax} - T_{elap}}{I_r}. \tag{5.25}$$

Execution Time Prediction: Equation 5.25 requires an accurate prediction of T_{fmax} at each control point. By comparing T_{elap} with $(1+\alpha)T_{fmax}$, we can roughly estimate how aggressively we should adjust the CPU frequency during the remaining execution time. Intuitively, if $T_{elap} << (1+\alpha)T_{fmax}$, we can reduce the CPU frequency to a much lower level than that if $T_{elap} >> (1+\alpha)T_{fmax}$. However, it is challenging to predict T_{fmax} accurately online because (1) the control algorithm changes the CPU frequency very rapidly, thus resulting in fast and yet significant performance fluctuations

and (2) the prediction algorithm should be efficient enough to avoid imposing significant overhead.

In order to derive a fast accurate prediction model, we fist decompose T_{fmax} into two parts: the amount of time it takes to execute the instructions retired when running at the highest frequency $T_{elap,max}$ and the remaining time to finish execution when running at the highest frequency $T_{remain,max}$. We can derive $T_{elap,max}$ using Equation 5.27: given that f_k is the frequency used for scaling period k, T_{k,f_k} is the amount of time elapsed at frequency f_k, f_{max} is the highest frequency, and MPI_k is the average MPI value, the amount of time required to execute the same number of instructions in period k when the highest frequency is employed, i.e., $T_{k,max}$ can be written as

$$T_{k,max} = T_{k,f_k} \cdot \frac{\text{SPI}(\text{MPI}_k, f_{max})}{\text{SPI}(\text{MPI}_k, f_k)}. \tag{5.26}$$

Therefore, $T_{elap,max}$ can be expressed as

$$T_{elap,max} = \sum_k T_{k,max} = \sum_k \left(T_{k,f_k} \cdot \frac{\text{SPI}(\text{MPI}_k, f_{max})}{\text{SPI}(\text{MPI}_k, f_k)} \right). \tag{5.27}$$

In order to determine $T_{remain,max}$, we first assume the instruction count of the current task is known a priori, e.g., by examining the input file size or history information. This assumption holds for most data processing applications such as image encoding and decoding, data compression, and placement and routing, whose run times are generally functions of input file size. Given that I_{total} is the total instruction count, I_{elap} is the number of instructions retired, I_r is the remaining number of instructions to be executed, and $\text{SPI}(f)$ is the amount of time per instruction at frequency f, we can express $T_{remain,max}$ as follows.

$$I_r = I_{total} - I_{elap} \text{ and} \tag{5.28}$$

$$T_{remain,max} = I_r \cdot \text{SPI}(f_{max}) \tag{5.29}$$

Combining Equations 5.27 and 5.29, T_{fmax} can be written as

$$T_{fmax} = T_{elap,max} + T_{remain,max}. \tag{5.30}$$

We also consider the scenario in which the total instruction count is unknown before the task is executed. We use I_r to denote the remaining number of instructions to

execute, in billions. We start with an I_r of 1. At every scaling point, we subtract the number of instructions retired since the last reset of I_r from the current I_r. If the result is smaller than 1, we reset I_r to the number of instructions retired since the task started. If the resulting I_r exceeds an upper bound I_{up}, we set I_r to I_{up}. I_r is then substituted into Equation 5.29 to estimate the remaining execution time. Note that I_{up} should be large enough to permit aggressive frequency control and yet small enough to preserve accuracy. We use an I_{up} of 30 in our experiments. We experimentally determined that the energy consumption is relatively insensitive to changes in I_{up}: a variation of only 0.8% in total energy consumption is observed when varying I_{up} from 5 to 500. In our experiments, given a performance degradation ratio of 0.2, the energy consumptions only deviate by 2% from those when I_{total} is known beforehand, i.e.., from precharacterization, file size based estimates, or assuming an oracle with knowledge of future application behavior.

Given that T_{fmax} and I_r can be estimated online, the energy minimization problem can then be formulated as an MCKP.

$$\text{Minimize } \sum_{i \in S} \sum_{f_j \in F} e_{ij} x_{ij} \tag{5.31}$$

$$\text{subject to } \sum_{i \in S} \sum_{f_j \in F} d_{ij} x_{ij} \leq \frac{(1+\alpha) T_{fmax} - T_{elap}}{I_r} \text{ and} \tag{5.32}$$

$$x_{ij} \in \{0, 1\}, \ \sum_{f_j \in F} x_{ij} = 1, \forall i \in S. \tag{5.33}$$

Note that we can treat the right hand side of the constraint in Equation 5.32 as positive. Otherwise, the constraint is trivially satisfied. Unlike the oracle scenario, the P-DVFS technique requires solving the MCKP online. Although MCKP is $\mathcal{NP}$-hard, there exist algorithms that can solve it in pseudo-polynomial time [44, 42]. In our experiments, we used "lp_solve" to obtain the optimal solution online. We used 15 MPI slots and 4 frequency levels in our experiments. For each of the benchmarks we evaluated, it took less than 1 ms to obtain the optimal solution, which is fast enough for online control. Note that this also indicates the energy overhead of the MCKP solver is approximately 0.1%, given a control period of 1 s in our experiments. Pisinger's efficient MCKP solver implementation would permit an even more efficient solution in a production version of the control software [44].

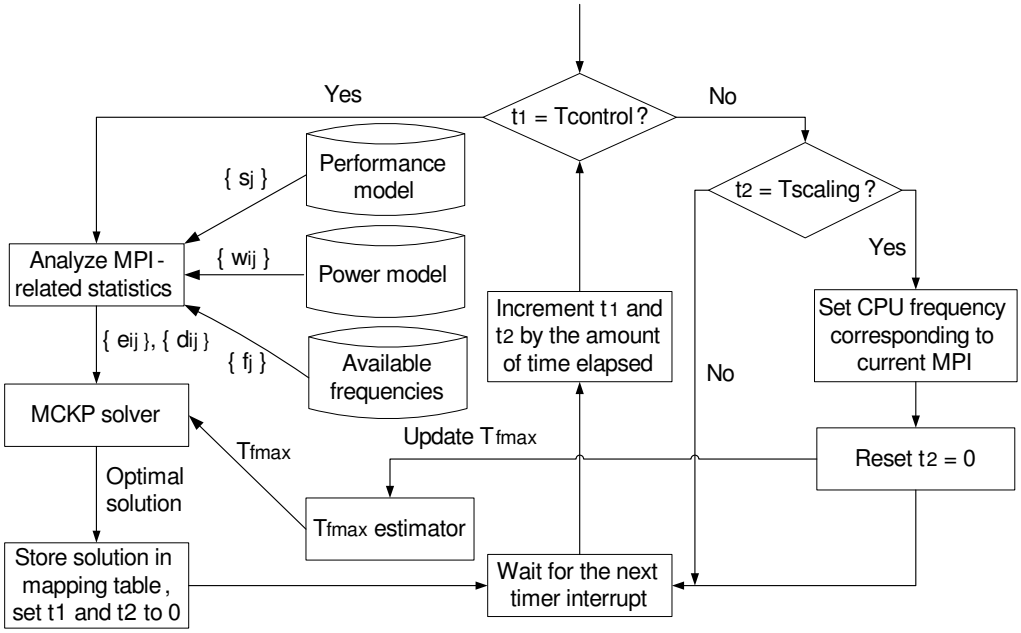

Figure 5.1: System architecture for P-DVFS.

5.3.4 System Architecture for P-DVFS

We have integrated the performance model, power model, execution time predictor, and MCKP solver to accurately control the CPU frequency for a fine-grained trade-off between performance and energy. Figure 5.1 illustrates the system architecture for the P-DVFS technique. We use $T_{control}$ and $T_{scaling}$ to represent the control and scaling periods. As indicated in Figure 5.1, whenever a timer interrupt occurs, we increment the time counters t_1 and t_2. We first determine whether t_1 has reached $T_{control}$. If so, we analyze MPI-related statistics, i.e., dividing the range of MPI values into distinct MPI slots and calculating the percentage of instructions (poi_i) associated with each MPI slot i. We also determine the values of coefficients such as $\{s_j\}$ in Equation 5.11 and $\{w_{ij}\}$ in Equation 5.13 using the performance and power models. We also gather information about the available processors frequencies f_j. These values are translated to $\{e_{ij}\}$ and $\{d_{ij}\}$ in Equation 5.31 and Equation 5.32, which are then provided to the MCKP solver

along with estimations of T_{fmax} and I_r in Equation 5.28 and Equation 5.29. The optimal solutions are then stored in a mapping table and time counters t_1 and t_2 are reset to 0. When $t_1 < T_{control}$, we continue to check whether t_2 has reached $T_{scaling}$ and if so, we set the CPU frequency to that corresponding to the current MPI in the mapping table and reset the time counter t_2. Otherwise, the T_{fmax} estimate is updated. The task then continues executing until the next timer interrupt occurs. Note that the DVFS algorithm is implemented in software and has very low performance and energy overhead (approximately 0.3%).

5.4 Experimental Results

In this section, we first describe the experimental setup and implementation details of the proposed techniques. We then present the experimental results for both P-DVFS and the optimal two-stage solution. Finally, we compare the results produced by P-DVFS with those produced by the optimal oracle solution and the most advanced published work [39].

5.4.1 Experimental Setup

We implemented our techniques on a Pentium Dual Core E2220 processor, which runs Linux 2.6.25 and operates at 1.2, 1.6, 2.0, and 2.4 GHz. We use the *cpufreq-utils* Linux kernel utility, to control CPU frequency. Experimental results indicate the switching overhead ranges from 50 µs to 200 µs. We use PAPI 3.6.2 [32] for HPC measurement and experimentally determined that the performance overhead for accessing HPCs is negligible. Due to the hardware limitations of our processor, we can only sample two architectural events at a time. Therefore, we time multiplex architectural event sampling to obtain all the values needed for power calculation. The switching interval is 10 ms and five architectural event counters are monitored, yielding a scaling period, ($T_{scaling}$) of 30 ms. The control period $T_{control}$ is set to 1 s, i.e., we solve the MCKP formulation every 1 s such that we can obtain a stable MPI distribution and capture changes in memory access behavior quickly enough for accuracy. We use a sliding window of 2 s to build the MPI distribution histogram. 15 MPI slots are used to permit different memory access behaviors to be distinguished while controlling MCKP solver overheard. We

experimentally determined that energy consumption is relatively insensitive to changes in the number of MPI slots: a variation of less than 0.5% in total energy was observed when varying the number of slots from 5 to 30. We note that the same MPI slots are used throughout the execution of a benchmark.

To determine power consumption, we use a Fluke i30 current clamp on one of the 12 V processor power supply lines, the output of which is sampled using a National Instruments USB6210 data acquisition card. This approach permits processor power consumption measurement without requiring printed circuit board rework or access to internal metal layers. An on-chip voltage regulator converts this voltage to the actual processor operating voltage. We assume a regulator efficiency 90%. and converted to power consumption: $P = V \cdot I = 12 \cdot I$, where P is the processor power and I is the measured current. Samples are taken at a frequency of 10 kHz.

5.4.2 Comparison with Prior Work

Choi *et al.* [39] proposed a fine-grained runtime DVFS technique that minimizes energy consumption while meeting soft timing constraints. We will use "F-DVFS" to refer to their technique. In order to take advantage of off-chip accesses, F-DVFS dynamically constructs a performance model and uses it to calculate the expected workload for the next slot; frequency and voltage levels are adjusted accordingly. F-DVFS has several weaknesses. It ignores long-term behavior such as the total application execution time. For example, at each scaling point, it considers only an immediate, local, user-specified performance constraint. However, sometimes even setting the frequency to the lowest level still results in a performance level higher than the user-specified constraint due to large number of off-chip accesses, opening the opportunity to improve energy savings when the MPI becomes lower later during execution. Neglecting total execution time makes it impossible to take advantage of such energy saving opportunities. Note that this sort of time-varying application behavior is very common for scientific computing applications, which commonly read a large amount of data into memory before processing. Moreover, F-DVFS neglects the relationship between frequency and energy consumption, assuming that reducing frequency is always beneficial to energy. However, this is not true when leakage power consumption is significant or the overall optimization goal is the system energy consumption instead of processor power consumption. In

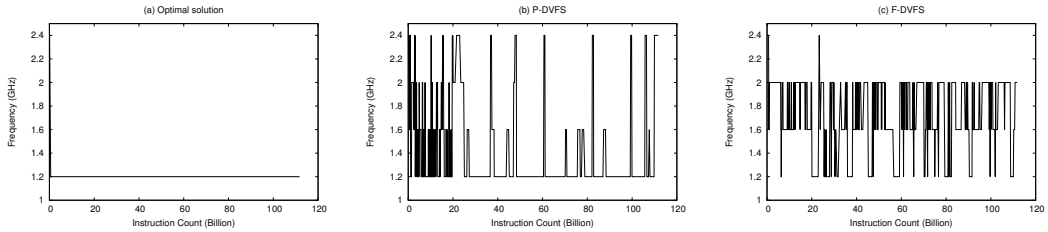

Figure 5.2: Processor frequency as a function of the number of instructions retired for (a) the optimal solution, (b) P-DVFS, and (c) F-DVFS during "mcf" execution with a performance degradation ratio of 20%.

contrast, P-DVFS automatically models and optimizes leakage power consumption and can be easily extended to handle the energy consumptions of other components such as main memory and disk.

5.4.3 Experimental Results

We evaluated P-DVFS on the 8 SPEC2000 benchmarks that compiled on our evaluation platform and 3 ALPBench benchmarks [45]. We did not consider the remaining 2 benchmarks ("MPGenc" and "MPGdec") in the ALPBench benchmark suite because they are very disk I/O intensive: we are presently interested in evaluating the impact of off-chip memory access on energy savings. We considered 3 floating point programs and 8 integer programs. The execution time of each benchmark ranges from 40 s to 425 s. For each benchmark, we specify a performance degradation ratio (the maximum increase in execution time relative to that at the maximum frequency and voltage) ranging from 5% to 20% with a step of 5%. The actual execution time and the average energy savings are reported compared to a scheme without DVFS, denoted as N-DVFS, F-DVFS, and the optimal oracle solution. We use the same window size for F-DVFS, P-DVFS, and the optimal oracle solution to permit a fair comparison. Both techniques use 4 discrete frequency levels.

Table 5.1 shows the actual performance degradation for both F-DVFS and P-DVFS compared with the user-specified performance degradation ratio. The first column specifies the benchmarks we evaluated. The "P-DVFS" and "F-DVFS" columns represent the performance degradation ratios resulting from using the two techniques, with the user-specified performance degradation constraint listed on the second "Goal" row. Given

Table 5.1: Performance Degradations of F-DVFS and P-DVFS

Benchmark	F-DVFS (%)				P-DVFS (%)			
Goal	5%	10%	15%	20%	5%	10%	15%	20%
gzip	0.27	0.34	1.36	10.59	4.74	8.03	10.82	16.62
vpr	0.00	1.91	10.06	11.62	4.83	9.93	14.05	19.39
mcf	2.02	4.51	6.61	7.78	4.50	6.50	13.50	17.00
bzip2	0.51	0.62	0.67	17.9	3.11	6.09	10.76	15.36
twolf	0.0	1.87	16.31	17.9	4.13	7.92	12.40	17.23
art	0.0	4.47	5.20	5.85	3.09	6.85	13.16	16.83
equake	0.0	0.0	0.0	9.64	3.04	7.59	11.72	15.42
ammp	0.23	0.93	7.18	16.13	4.24	10.40	14.41	19.29
facerec	0.0	4.09	10.12	20.2	3.19	7.65	13.65	18.38
sphinx3	0.0	0.54	1.48	9.34	2.80	7.50	11.10	13.84
tachyon	0.0	5.91	6.83	16.4	3.22	8.41	13.57	18.43
Average	0.28	2.29	5.98	13.03	3.72	7.90	12.65	17.10

that the performance constraint is satisfied, a larger performance degradation usually corresponds to larger energy savings; this was confirmed by our experiments. Experimental results indicate that P-DVFS can approach the user-specified constraint more closely than F-DVFS. More specifically, given a user-specified performance degradation percentages ranging from 5% to 20%, P-DVFS can reach a performance degradation percentages of 3.72%, 7.90%, 12.65%, and 17.10%, whereas F-DVFS can only achieve percentages of 0.28%, 2.29%, 5.98%, and 13.03%. P-DVFS has finer-grained control over the trade-offs between performance and energy given a user-desired performance constraint. FDVFS does not reach the user-specified performance degradation ratio partially because the number of available frequencies is limited: whenever the calculated frequency f_{calc} does not correspond to any available frequency, FDVFS uses the closest frequency that is larger than f_{calc} to approximate it. This may reduce the energy benefit when the number of available frequency is small. Switching between two closest available frequencies may address this problem. However, there are more fundamental reasons why FDVFS does not work as well as our techniques, as we explained later in this section. Note that both techniques may violate the soft timing constraint due to inaccuracies in the online performance model. However, for P-DVFS, the maximum violation is less than 1%, which could be eliminated by using a 1% guard band for the constraint.

We compared the energy savings of NDFS, F-DVFS, and P-DVFS with those of the

Table 5.2: Deviation of Energy Consumptions from the Optimal Solution when using using N-DVFS, F-DVFS, and P-DVFS

Benchmark	E_{opt} (J)	N-DVFS (%)	F-DVFS (%)	P-DVFS (%)
gzip	804	7.88	6.88	0.12
vpr	1520	21.91	8.09	3.36
mcf	2401	71.10	29.86	4.83
bzip2	1345	8.18	1.93	0.30
twolf	5281	12.61	1.50	1.38
art	1810	52.49	23.20	4.42
equake	2736	14.58	7.20	1.90
ammp	7344	12.15	2.08	0.14
facerec	2621	12.59	6.37	0.04
sphinx3	1428	19.54	11.13	3.64
tachyon	2210	15.43	9.55	0.05
Average	2682	22.59	9.80	1.83

optimal oracle solution, which might be better than the actual optimal on-line solution. For performance degradation percentages of 5%, 10%, and 15%, N-DVFS generates solutions that deviate from the optimal solution by 9.31%, 12.81%, and 18.46%, with maximum deviations of 22.29%, 33.72%, and 56.55%; F-DVFS leads to energy consumptions that deviate from the optimal solution by 7.1%, 8.23%, and 9.51%, with maximum deviations of 16.84%, 15.89%, and 29.8%; and P-DVFS results in energy consumptions that deviate from the optimal solution by 1.43%, 1.16%, and 1.59%, with maximum deviations of 2.80%, 3.88%, and 4.63%. Since the results are similar for different performance degradation ratios, we only present the energy numbers for a maximum performance degradation ratio of 20% in Table 5.2. The first column specifies the application being evaluated. The second column indicates the optimal, i.e., minimum, energy consumption for each benchmark with a performance degradation ratio of 20%. The third, the fourth, and the fifth columns represent the deviation in energy consumption from that of the optimal oracle solution when using N-DVFS, F-DVFS, and P-DVFS. As indicated in Table 5.2, the energy consumption deviates from the optimal oracle solution by 22.59% on average when no DVFS is used, with a maximum deviation of 71.1%. F-DVFS produces solutions that deviate 9.8% from the optimal oracle solution on average, with a maximum deviation of 29.86%. Among the three candidates, P-DVFS achieves the best solution quality, i.e., an average of 1.83% deviation from the

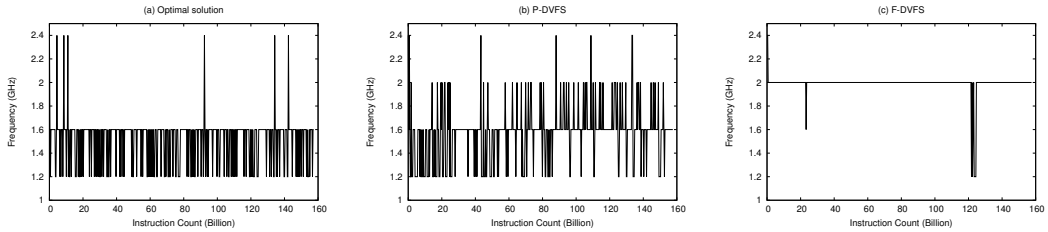

Figure 5.3: Processor frequency as a function of the number of instructions retired for (a) the optimal solution, (b) P-DVFS, and (c) F-DVFS during "art" execution with a performance degradation ratio of 20%.

optimal oracle solution with a maximum deviation of 4.83%. Therefore, we conclude that P-DVFS can very closely approximate optimal solutions. It is also worth noting that for performance degradation ratios of 5%, 10%, 15%, and 20%, P-DVFS has average power savings of 8.3%, 11.31%, 12.3%, and 9.93% and maximum power savings of 15.94%, 12.69%, 27.36%, and 25.64% compared to F-DVFS.

It is interesting that for benchmarks such as "mcf" and "art", F-DVFS leads to solutions that are far worse than those using P-DVFS (25.03% and 18.78% difference, respectively). We now analyze their results.

Analyzing *Mcf* Results

Figure 5.2 illustrates the dynamic processor frequency changes for the optimal oracle solution, P-DVFS, and F-DVFS during execution of the "mcf" benchmark, given a performance degradation ratio of 20%. The X-axis represents the number of billion instructions retired and the Y-axis represents the frequency. Figure 5.2(a) suggests that the optimal solution is to always set the frequency to the lowest level. While P-DVFS yields a near-optimal solution, F-DVFS behaves very differently. We note that "mcf" is a two-phase benchmark: the cache miss rate is very high during the first 20 billion instructions and alternates between a high value and a low value afterwards. In both phases, F-DVFS leads to a higher frequency on average. Recall that F-DVFS requires accurate model estimation and accurate individual coefficients so that it can correctly estimate the ratio of off-chip to on-chip memory accesses. Although the former is generally true for linear regression, the second assumption does not necessarily hold. In this case, since the MPI and CPI values do not change much in the first phase,

the coefficients derived using linear regression can be inaccurate, causing F-DVFS to significantly over-estimate the average on-chip latency and thus limit itself to a relatively high frequency (2 GHz). We analyzed the results and observed this behavior. Note that the output of the performance model, or CPI, is still accurate. In contrast, P-DVFS only requires that the output of the model match the real CPI value: the individual coefficients in the regression formula do not matter. Therefore, P-DVFS allows the CPU frequency to be decreased to a lower level, alternating between 1.6 GHz and 1.2 GHz most of the time. The frequency does not stay at the lowest level due to inaccuracies in the online performance model and the remaining execution time predictor. In the second phase, F-DVFS increases the frequency when the cache miss rate is lower and decreases the frequency when the miss rate is higher. This happens because F-DVFS considers only immediate application behavior and ignores long-term behavior such as total execution time. However, this may result in sub-optimal solutions, as demonstrated by Figure 5.2(b). P-DVFS takes history and long-term behavior into account, allowing it to correctly determine that the frequency can be set to the lowest level even when the cache miss rate is low. Therefore, P-DVFS achieves much larger energy savings in this case, savings that approach those of the optimal oracle solution.

Analyzing *Art* Results

Figure 5.3 illustrates the dynamic processor frequency changes for the optimal oracle solution, P-DVFS, and F-DVFS during the execution of the "art" benchmark, given a performance degradation ratio of 20%. As shown in Figure 5.3, P-DVFS closely approximates the optimal oracle solution and F-DVFS does not. This can be explained as follows. "Art" has periodic cache access behavior with a period of approximately 300 ms at the highest frequency. In each period, the MPI value starts from a low value (0.003 in our experiments) and gradually increases before it reaches the point with the highest MPI (0.005 in our experiments). Then, the MPI value starts to decrease until it returns to the previous value of 0.003. F-DVFS gathers the sampling points within the most recent second to build the performance model. It is likely that the coefficients in the regression formula will remain nearly constant due to the small period and large window size; this was confirmed in our experiments. Therefore, the frequency was set to a fixed number (2 GHz in our case) for all the sampling points in each period.

In contrast, P-DVFS builds the MPI distribution based on the sampling points from the most recent second, translates the energy minimization problem into an MCKP instance, and solve it to get the optimal solution, which indicates we should use high frequency (2 GHz) for sampling points with low MPI and low frequency (1.2 GHz) for sampling points with high MPI. As shown in the experimental results, the overall effect is achieving significant reduction in energy compared to F-DVFS. Since F-DVFS is not distribution-oriented, it cannot know how SPI and power consumption change with MPI. Therefore, it is impossible for F-DVFS to take advantage of the distribution and assign different frequencies to sampling points with different MPIs while still meeting the performance constraint.

For the rest of the benchmarks, P-DVFS slightly outperforms F-DVFS. This is because both consider the effects of off-chip memory access latencies on energy. For benchmarks with relatively few L2 cache misses, e.g., twolf and vpr, the energy consumptions are similar. Therefore, the proposed technique will achieve the greatest energy savings compared to past work for applications with phases during which the energy cost per instruction differ.

5.5 Conclusions

This chapter describes a new power state control technique that adapts to the time-varying memory access behaviors of applications. We first proposed a two-stage DVFS algorithm based on formulating the throughput-constrained energy minimization problem as a multiple-choice knapsack problem (MCKP), assuming a priori characterization-based or oracle knowledge of application behavior. This algorithm builds on an application phase-dependent power model, which can be constructed offline using processor hardware performance counters. We then present an online DVFS technique, called P-DVFS, that predicts remaining execution time in order to control voltage and frequency to minimize energy consumption subject to a performance constraint. P-DVFS requires no information a priori knowledge of application behavior. In addition to the power model, P-DVFS also uses a performance model that accurately captures the relationship between performance and off-chip memory access rate. These two models,

combined with an execution time predictor, allow us to formulate the energy minimization problem again as a multiple-choice knapsack problem, which can be efficiently and optimally solved online. Experimental results indicate that given a performance degradation ratio of 0.2, P-DVFS leads to energy consumptions within 1.83% of the optimal oracle solution on average with a maximum deviation of 4.83%, whereas the most advanced related DVFS control technique (F-DVFS) results in energy consumptions within 9.8% of the optimal oracle solution on average with a maximum deviation of 29.86%. For the same performance constraint, we found that P-DVFS also reduces power consumption by up to 25.64% (9.93% on average) compared to F-DVFS. These energy and power savings are all directly measured on a real system.

Chapter 6

Overview for GPGPUs

The second part of the dissertation will focus on GPUs for general purpose computing. The massive processing capability of GPUs has recently attracted growing attention from general purpose parallel applications. Heterogeneous computing with multicore CPUs and multicore GPUs is emerging as the best performance/cost combination for high-performance computing (HPC) [46]. However, in spite of the great potential for energy efficiency, as well as recent hardware performance improvements, GPUs are still significantly underutilized in comparison with CPUs due to various architectural features that are incompatible with some characteristics of general purpose parallel applications [47]. In this chapter, we will investigate modern GPU architecture, characterize GPGPU applications, perform a thorough analysis on CPI breakdown and identify all the key factors that govern GPU throughput from a single warp perspective.

6.1 Introduction

The design philosophy of GPUs aims to optimize for the execution of a massive number of threads. GPUs are characterized by numerous simple yet energy-efficient computational cores that run thousands of simultaneously-active fine-grained threads, large off-chip memory bandwidth, and simple control logic. However, as the execution resources required by HPC tasks may not always match the characteristics of GPUs, the problem of efficiently managing workloads on GPUs and leveraging their substantial throughput potential has emerged as a significant research challenge.

Several constraints contribute to the inability of GPUs to achieve their peak throughput. First, there is the issue of *thread level parallelism*. Each streaming multiprocessor (SM) supports up to thousands of in-flight threads in order to hide long latencies from arithmetic and memory operations. However, threads are scheduled to cores in units of thread blocks and the amount of resources (register, shared memory, etc.) required by each block sets a hard limit on how many blocks of threads can be scheduled simultaneously. An application that requires more resources per thread/thread block than are available may suffer a significant throughput penalty. Second, underutilization in thread schedulers can result in *scheduling constraints*. Each GPU core, or Streaming Processor (SM), includes multiple Single-Instruction Multiple-Thread (SIMT) pipelines for ALU computations, special functions, and memory operations. However, the throughput of the scheduler and instruction dispatch unit often cannot keep all the pipelines busy, resulting in some of the pipelines being underutilized. If we can judiciously issue more than one instruction into different pipelines every cycle, we may gain throughput benefits. Third, unbalanced utilization among different GPU function units results in uneven usage on various pipelines. We use the term *pipeline-level parallelism* (PLP), to describe the parallel utilization of different function units. An application may have a unique performance bottleneck, e.g., it may be compute-bound or memory-bound, and this leads to substantial underutilization in the rest of the pipelines.

The remainder of this chapter is organized as follows. Section 6.2 provides background on the state-of-the-art GPU architecture and describes the benchmarks suite, application metrics, simulation environments we used for evaluation. Next, Section 6.3 motivates the whole GPU optimization problems by breakdown CPI into several key components, and demonstrates how the number of warps along with CPI per warp impact IPC. Finally, section 6.4 provides an overview of the rest of the dissertation, and how the following chapters tackle the GPU optimization problem from different perspectives.

6.2 Background

6.2.1 Baseline CUDA and Fermi Architecture

CUDA is a parallel computing architecture developed by Nvidia [48]. It abstracts the thread-level parallelism of the GPU into a hierarchy of threads (*grids* of *blocks* of *warps* of *threads*) [49]. These threads are then mapped onto a hierarchy of hardware resources. The basic unit of execution flow, the *warp*, contains 32 threads that execute the same instruction based on the single instruction, multiple thread (SIMT) paradigm.

Figure 6.1 illustrates the detailed microarchitecture of the warp scheduler and SIMT pipelines inside a CUDA SM. Each SM features two warp schedulers and two dispatch units with all the warps evenly divided according to the parity of the warp ID, as shown in the box marked "Scheduler". Each warp scheduler can function independently without dependency checking across the schedulers. Each SM also contains 32 streaming processors (SP) divided evenly into 2 pipelines, 4 special function units (SFU) and 16 load/store units (MEM), as shown in the box marked "SIMT Pipelines". Considering that each pipeline (excluding SFU) has 16 execution units, while a warp contains 32 threads, it takes at least 2 cycles for an instruction to be issued to the pipeline. As a result, the dual warp schedulers run at half of the pipeline frequency, issuing a maximum of one instruction every cycle.

The warp scheduler maintains the status of warps on a per-cycle basis. As shown in Figure 6.1, the warp status in the scheduler can take on one of three values. A warp is *inactive with control hazards* when the next instruction is not stored in the instruction buffer, and thus cannot be issued immediately. This scenario only occurs when the instruction is a branch or function call; in both cases, it is observed that the probability that a warp turns inactive due to control hazards, $P_{inactive\_control}$, remains quite stable and can be considered as a kernel-dependent constant. A warp is *inactive with data hazards* when the next instruction of the warp has a data dependency on a previous instruction which still resides in the pipelines. An *active* warp has no data dependency issues and is ready to be issued immediately.

The scheduler picks an active warp from its own active warp pool in a loosely round-robin fashion, sends the warp to its dedicated SIMT pipeline, and updates the warp status and data dependencies. While inside the dedicated SIMT pipeline, the

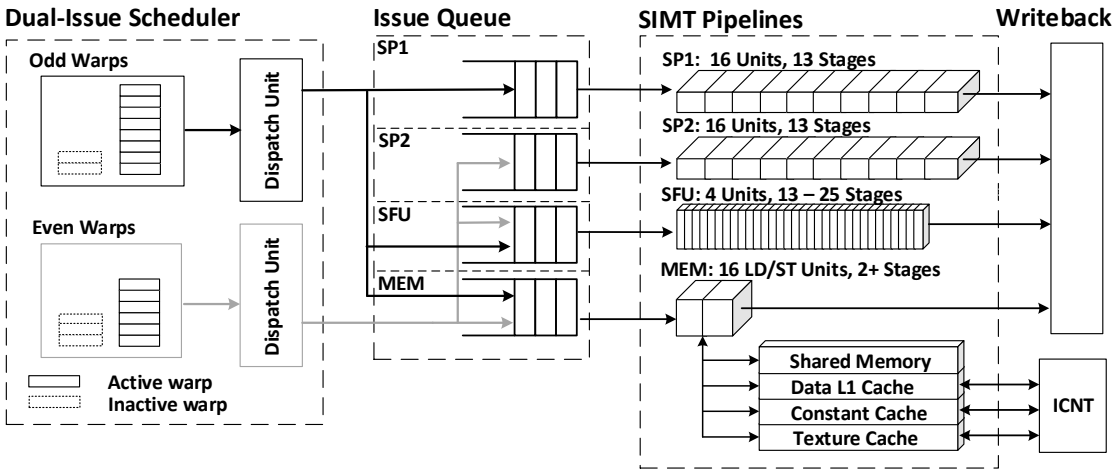

Figure 6.1: Microarchitecture of a GPU core in Fermi GTX 480.

instructions are sent into an operand buffer while waiting for all the input registers to be acquired. Once all inputs are ready, the operand buffer issues the instructions to the execution pipeline in a first-in-first-out fashion. For each arithmetic SIMT pipeline, there are over 20 pipeline stages [50]. Considering extra stalls caused by the dispatch unit and potential registers bank conflicts, a significant amount of warps are needed to avoid stalls in arithmetic pipelines, and particularly in the even more time-consuming MEM pipeline. If no active warp is available, or the warp is issued to another SIMT pipeline, a stall occurs and a bubble is inserted into the SIMT pipeline. At the write-back stage, the instruction is considered finished and the warp status is updated.

6.2.2 Workload and Metrics

Application Suite:

We perform evaluations using the Parboil benchmark suite [51], which contains a wide range of GPGPU applications optimized for CUDA architecture, as shown in Table 6.1, including bimolecular simulation, fluid dynamics, image processing, astronomy, and dense and sparse linear algebra.

Each application consists of one or more kernels. We observed that even kernels from the same applications can exhibit different characteristics. We pick kernels based

Table 6.1: List of GPGPU kernels.

Bench.	**Abbr.**	**Kernel**	**Weight**	**Avg. Kernel Cycles**	**Invocations**	**Avg. Launch Overhead (μs)**
bfs	BFS	BFS_in_GPU_kernel	100%	22	1	-
cutcp	CUT	cuda_cutoff_potential	99.90%	5	26	71
histo	HIS	histo_main_kernel	51.30%	0.3	10000	3
lbm	LBM	performStreamCollide_kernel	100%	3	1	-
mri-q	MRI	ComputeQ_GPU	99.60%	4	2	73
sad	SAD	mb_sad_calc	52.50%	18	1	-
sgemm	SGE	mysgemmNT	100%	3	1	-
spmv	SPM	spmv_jds	99.90%	0.4	50	3.3
stencil	STE	block2D_hybrid_coarsen_x	99.80%	2	100	5.2
tpacf	TPA	gen_hists	100%	7	1	-

Table 6.2: GPGPU-Sim Configuration for Baseline Architecture (Fermi GTX 480).

GPU config.	15 GPU cores, 2.0 Compute Capability
Frequency	1400MHz Core, 700MHz ICNT, 924MHz DDR5
GPU Core Config.	SIMT Width: 16 (SP1, SP2 and MEM), 4 (SFU)
Resources/Core	Max 1536 Threads, Max. 8 CTAs, 48KB Shared Memory, 32768 Registers
Caches/Core	16KB, 128B line, 4-way, 64 MSHR L1 Data Cache 12KB, 128B line, 24-way Texture Cache 8KB, 64B line, 2-way Constant Cache
Unified L2 Cache	768KB, 128B line, 16-way, 256 MSHR
Scheduling	GTO (Greedy-then-Oldest Scheduling)
Interconnect	2D mesh (5x5, 15 cores+6 Memory Controller)
DRAM Model	FR-FCFS, 6MC, Burst Length 8, Buswidth 8B/MC, Total 384bits
GDDR5 Timing	924MHz, 16 Banks, $t_{CCD} = 2$, $t_{RRD} = 6$, $t_{RCD} = 12$, $t_{RAS} = 28$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{CL} = 12$, $t_{WL} = 4$, $t_{CDLR} = 5$, $t_{WR} = 12$, $t_{nbkgrp} = 4$, $t_{CCDL} = 4$, $t_{RTPL} = 2$

on their weight (ratio between kernel execution time and whole application time) in each application and perform evaluations on both GTX 480 hardware and GPGPU-Sim (version 3.2.0) [52]. We model our baseline architecture after Fermi GTX 480 [53] with the configuration shown in Table 6.2.

Table 6.1 shows kernel performance characteristics captured through hardware profiling. Invocation indicates how many times the kernel has been launched in the application. From the kernels we evaluated, different invocations exhibit similar function unit utilization. For simulation simplicity, if a kernel has hundreds of innovations, we repeatedly simulate the kernel with the same input set. In addition, For kernels with

more than one invocation, we measured kernel launch overhead, the gap between when the previous kernel finishes and a new kernel launches. Note that this does not include memory copy time. The overhead is often in μs, but when kernels are very short, it can have a significant performance impact, since the launch overhead becomes larger relative to the kernel execution time. For example, in HIS, the kernel launch overhead is over 1% of the kernel execution time.

Evaluation Metrics:

We use SM IPC, the average number of instructions issued per cycle in one SM, as a performance metric. More specifically, SM IPC in this dissertation stands for the average number of instructions issued from warp schedulers per cycle, which has a direct relation to the pipeline utilization. For the rest of dissertation, IPC indicates SM IPC.

The average number of cycles per instruction, CPI per warp is also used to investigate the stalls each warp suffers due to various reasons. Note in theory, given the number of warps, and the CPI per warp, SM IPC should equal to the number of warps divided by CPI.

6.3 Characterizing CPI Breakdown

In every cycle, the warp scheduler selects a ready warp from the active warp pool for execution. As long as one in-flight warp is ready in every cycle, throughput is maximized. However, there are several reasons that a warp may not be ready [54]: instruction cache misses, barriers, warp finished before the rest of the warps in the same CTA, control hazards, data hazards, and structural hazards. To evaluate the effectiveness of the GPU's latency hiding ability and explore how it might be improved, we identify and analyze all the significant sources of execution time delays for a warp.

Instruction cache misses: In order to avoid instruction fetch latency, each warp has a two-entry instruction buffer. When no instruction is available in the buffer, additional delay is added before the next instruction can be fetched. This is mainly cased by instruction cache misses.

Barrier: Barrier synchronization allows all the threads within the same CTA to wait for each other before moving forward. Once a warp hits a barrier, it stalls until the rest of the warps within the same CTA reach the barrier. The more warps each CTA has, the more likely a warp will stall at a barrier. So, it's important to keep all the warps within a CTA progressing at the same rate.

Function done This is similar to a barrier stall. When a warp finishes before the rest of the warps in its CTA, it stalls until the CTA finishes, at which time a new CTA is issued. When there are no more CTAs available, the stall due to function done is also considered as tail effect.

Control hazards: Unlike CMPs that are often equipped with sophisticated branch prediction logic, GPUs rely on massive parallelism to hide latency from control hazards. However, from a single warp's perspective, if a branch or function call instruction executes, the warp stalls until the target address is calculated.

Structural hazards: Structural hazards are caused by the unavailability of functional units when there are active warps ready to issue or unavailability of miss status holding registers (MSHRs) in the memory system. In modern GPU architectures such as Fermi [53], the memory pipeline is unavailable if it suffers stalls when MSHRs are full. Structural hazards often occur in SFU or MEM pipelines in GPUs, as the throughput of SP is usually much larger than the throughput of MEM and SFU. For instance, the throughput ratio between SP, SFU and MEM is 16:1:8 in Fermi.

Data hazards: Data dependency can introduce stalls when the next instruction of a warp depends on a result from a previous instruction. Currently, the GPU does not support data forwarding, so a warp stalls until all data dependencies have been resolved. If an instruction depends on a load instruction that goes to global memory (DRAM), the warp might stall for hundreds of cycles before the dependency is resolved.

6.3.1 Analyzing CPI Breakdown

To illustrate how different stall factors can contribute to the CPI of a warp, we developed an algorithm to count and categorize the cycles per instruction for each warp. In this section, we use the latency characterization algorithm introduced by Lee *et al.* [54]. In every cycle, profiling increments one of the stall counters for each warp if no instruction is issued from the warp. If there is overlap among multiple stall factors, we increment

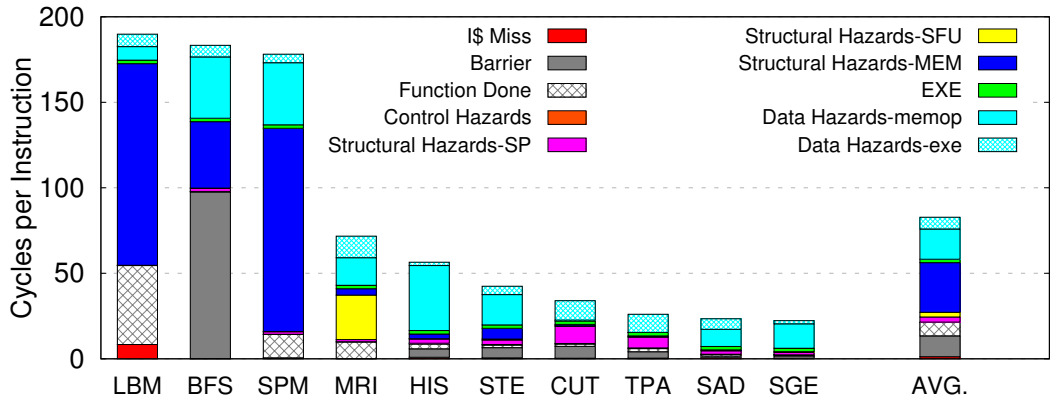

Figure 6.2: The CPI per warp breakdown for Parboil benchmarks with GTO scheduling.

the first stall counter following the order in section 6.3, which defines the order that stalls occur in the pipeline (e.g., and instruction cache miss would happen before other types of stalls, etc.).

Figure 6.2 presents the average CPI breakdown for Parboil applications. Each bar shows the CPI contributed by various stall factors described in section 6.3. To better investigate the CPI breakdown, we further break down structural hazards into structural hazards due to SP, SFU, and MEM function units and data hazards into data hazards due to load instructions and execution instructions. The CPI breakdown results are the average across all the warps among all the SMs throughout the kernel execution. The total CPI of each kernel indicates the effectiveness of its latency hiding ability when we launch as many warps as possible, which also shows how many warps are needed to completely hide the latencies of the kernel. Kernels toward the left do not hide latencies well, whereas the kernels on the right have smaller latencies that can be easily hidden with sufficient warps. We can derive the IPC of an SM by combining CPI with the number of warps each kernel issued per SM. Figure 6.3 shows how CPI per warp and the number of warps determines IPC for Parboil applications. The x-axis represents the average IPC of each kernel, and the y-axis is the number of warps in-flight per SM divided by per-warp CPI. The figure also list the number of warps each kernel issues per SM. The data trend confirms that $IPC = N_{warps}/CPI$. I.e., we can improve IPC by

reducing CPI per warp and improving warp occupancy. Since it is difficult to change warp occupancy without modifying the GPU architecture or the existing scheduling scheme, we first investigate how to reduce each component that contributes to CPI.

The most dominant CPI components in Figure 6.2 are *structural hazards-MEM*, which contribute 35.09% of the total CPI, primarily due to contention in MSHRs and other resources that can mark the MEM function unit unavailable. SPM and LBM in particular experience a significant number of stalls from MSHRs. This is because both kernels are memory bandwidth-intensive with many L1 cache accesses/misses. As a result, the performance is degraded significantly due to structural hazards from MEM. The structural hazards due to SP and SFU components are relatively small, contributing 3.62% and 3.30% of the total CPI, respectively. Note that structural hazards indicates the unavailability of certain function units, so they cannot be improved by increasing the degree of parallelism (adding more warps). In addition, if one kernel suffers significant structural hazards due to one of the function units, it also indicates significant under-utilization in the rest of the function units. Moreover, it's also hard to improve the utilization balance among different function units, since each kernel consists of many identical threads, so execution characteristics remain relatively stable. It is worth noting that the scheduling policy can sometimes impact structural hazards. The scheduler is responsible for picking the right warp among the active warp pool in every cycle. If there is a phase in which kernels are heavily utilizing one of the function units, a good scheduling policy would be able to reduce structural hazards by keeping warps moving at different paces so that different warp spread out their intensive utilization.

The next two most significant CPI components are *data hazards due to mem and exe operation* stalls caused by waiting for data to be ready from previous load or arithmetic instructions. Kernels, such as MRI, CUT, and TPA suffer from data hazards due to arithmetic instructions. For SPM and HIS, this is due to data hazards from previous load instructions. When there are sufficient warps, the scheduler can easily hide those latencies because, unlike structural hazards, data hazard latency does not increase as the degree of parallelism increases. Furthermore, note that scheduling policy cannot reduce CPI portions due to data hazards.

Stalls due to *barrier* and *function done* correspond to 14.64% and 9.52% of the total CPI in Figure 6.2. Despite having a high degree of parallelism, stalls due to barriers

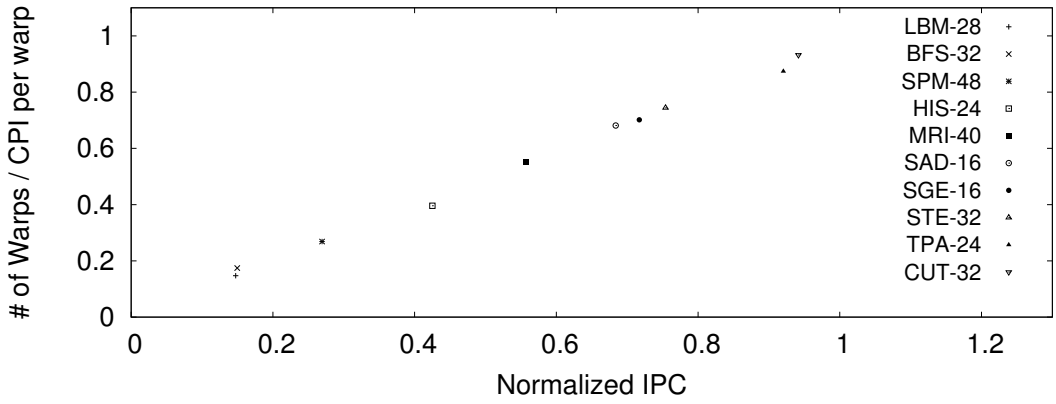

Figure 6.3: This figure shows the relationship between number of warps, CPI, and IPC

and function done can greatly reduce the number of active warps, leaving warps waiting for the rest of warps in the same CTA if the warps are in different pace. For instance, in LBM and MRI, stalls due to function done contribute 24.31% and 13.61% of the the total CPI, and BFS suffers 53.27% of the stalls due to barrier. Scheduling policy might be able to keep different CTAs progressing differently to avoid an overlap of such stalls from different CTAs, but the effect is very kernel-dependent, especially for those kernels with bigger but fewer CTAs per SM. Furthermore, given kernels with the same characteristics, the more warps each CTA has, the harder it is to keep all of them in the same pace, so stalls due to barrier and function done could be longer. As a result, the scheduling policy plays a key role here to reduce the CPI components due to barrier and function done. By keeping all the warps in a similar pace throughput execution, in theory, we can easily reduce this CPI portion. However, current scheduling policies such as LRR and GTO do not have such awareness [54].

In this section, we laid out all the key factors that govern GPU throughput from a single warp perspective. To sum up, in order to improve GPU throughput, we need to improve the degree of parallelism, reduce structural and data hazards, and improve stalls due to barrier and functions done. The following chapters are focus on approaches that can tackle one or some of the aspects.

6.4 GPU optimization overview

The rest of the dissertation is organized as follows, In chapter 7, we proposed an priority scheduling scheme that can reduce CPI stalls due to barrier and function, as long as improve structural hazards.

chapter 8-9 present a new approach, intra-core multitasking, by allowing multiple kernels running simultaneously, we tackle all the factors described in subsection 6.3.1. chapter 8 proposes a run-time intra-core multitasking for GPGPUs, coupled with minor architectural modification and new scheduling scheme, we can greatly improve GPU throughput. Allowing multiple kernels running simultaneously brings new challenges, we observe that it's very costly to dynamically adjust the resource allocation between kernels. Therefore, chapter 9 develops a contention-aware performance model for intra-core multitasking. This allows us to find the optimal thread partition among kernels before-hand, which further improves overall throughput when combining with intra-core multitasking. Finally, we summarize the contributions of the GPU part in chapter 10.

Chapter 7

Priority Scheduling for GPGPUs

Chapter 6 lays out all the key aspects in order to improve GPU throughput, including improve the degree of parallelism, and reduce per-warp CPI through some of the key stall factors, such as structural hazards, data hazards, and function done *et al.*. Given limited resource on each GPU SM, it's hard to add more warps to the SM in a single kernel scenario. This chapter focuses on proposing a new priority scheduling scheme that is more sensitive to the key CPI components, and scheduling the active warps wisely to avoid stalls such as barrier, function done, and structural hazards *et al.*.

7.1 Introduction

The scheduler is responsible for picking the right warp among active warp pool every cycle. From the CPI breakdown analysis in subsection 6.3.1, we know scheduling policies can effectively optimize some of the key CPI components such as barrier, function done, and structural hazards *et al.*. In this chapter, we explore multiple scheduling policies that mainly focused on optimizing those CPI components. CPI due to structural hazards, barrier and function done contribute 47.64%, 3.68% and 12.23% of the total CPI, as shown in Figure 6.2, certainly we cannot eliminate all of them. In addition, there might be overlaps among various CPI components, which means reducing one type of the CPI can make the other CPI components larger. However, for CPIs due to structural hazards, barrier and function done, we observed they are sensitive to

scheduling policies, and we can still get substantial benefits if our scheduling policy are aware of those factors. As discussed in subsection 6.3.1, the key to reduce stalls due to barrier and function done is to keep warps in the same thread block in the same pace, such as LRR; and the key to improve structural hazards is to avoid all the warps in the same pace, such as GTO. In this chapter, we propose GTLS-TAWS, a greedy and thread block based, but also tail-aware warp scheduling policy that aims to improve IPC by reducing stalls due to structural hazards, barrier and function done.

Related Work

Various warp scheduling techniques have been proposed to improve data hazards and structural hazards. Rogers *et al.* [55] propose a cache-conscious warp scheduling policy that aims to reduce cache contention. Jog *et al.* [56] propose OWL, a series of CTA-aware warp scheduling techniques to reduce contention in both cache and DRAM. Jog *et al.* [57] propose a prefetch-aware warp scheduling policy to improve memory tolerance. The technique regroups threads according to their data spatial locality and pairs with a prefetching mechanism to effectively reduce memory latency. Kayiran *et al.* [58] propose DYNCTA – a runtime CTA modulation scheduling strategy to improve degree of parallisim and reduce contention in the memory hierarchy. We observe that this technique can effectively reduce structural hazards stalls due to memory contention. However, CTA modulation also reduces degree of parallelism and results in an additional 10% data hazards stalls, resulting in limited overall improvement in system performance.

7.2 Exploration of Scheduling Policies

Loose Round-robin (LRR): as the name suggests, the round-robin policy schedules the warp in equal portion and in circular order. Thus , all the warps are treated equally, and all the warps are likely maintained in the similar progress.

Greedy-then-oldest (GTO): GTO runs a single warp until it stalls then picks the oldest ready warp. The age of a warp is determined by the time it is assigned to the core. For wavefronts that are assigned to a core at the same time (i.e. they are in the same thread block), warps with the smallest threads IDs are prioritized. Other greedy

schemes (such as greedy-then-round-robin and oldest-first) were implemented and GTO scheduling had the best results.

Greedy-then-least-scheduled (GTLS): GTLS runs a single warp until stalls then picks the warp with longest waiting time. The waiting time of a warp is defined as the number of cycles since last instruction of the warp is selected and issued by the scheduler. Note different warps in the same thread block can have different waiting time. This scheme is aimed to take advantage of both short-term data locality within a warp (GTO) and long-term fairness among all warps throughout execution (LRR).

Thread block based Tail-aware warp scheduling (TAWS): this scheduling policy aims to improve all the CPI stalls due to barrier and function done, by keeping all the warps within the same thread block in the same pace. Meanwhile, with multiple thread block assigned, we prioritize thread blocks based on the number of warps that are currently stalled due to barrier and function done. By giving higher priority to the thread block with the most warps stalls due to barrier and function done, we can allow such thread block finishes faster, such that hardware resources become available to other new thread blocks (initially without any barrier and function done warps). Consequently, we can significantly reduce stalls due to barrier and function done, and different thread blocks are maintained in separate pace, which alleviates contention. TAWS is a scheduling scheme that focuses on prioritizing thread blocks, which after our investigation, works best when combined with other scheduling policies that also specifies the priorities of warps within a thread block. In this chapter, we combine TAWS with both GTO and GTLS. To be more specific, GTO-TAWS runs a single warp until stalls, then fetches the thread block according to TAWS ranking, while GTLS-TAWS runs a single warp until stalls, then fetches the thread block according to TAWS ranking, and issues the active warp with longest waiting time according to GTLS.

7.3 Implementation of Priority Scheduling Policies

We design a two-level per-warp priority counter that indicates the scheduling order. Top level counter determines the issue priority between thread blocks, and the 2nd level counter ranks the warps within a thread block. For example, given there are 48 active warps in the pool from 6 thread blocks (8 warps per thread block), we first rank 6

thread blocks based on the thread block ranking algorithm, once the scheduling order among thread blocks are determined, we apply our 2nd level ranking algorithm on the warps within the same thread block. Note, every time the value of warps are updated in the ranking algorithm, we need to update the priority counter and recalculate the scheduling order. Therefore, we also want a simple yet effective ranking algorithm that does not update the scheduling order unless necessary.

7.3.1 Ranking Algorithm

A ranking algorithm determines which warp is more important and should be issued more often. For GTO-TAWS and GTLS-TAWS, we explore an absolute two-level priority ranking. First, we rank thread blocks based on the number of warps that are currently stalled due to barrier and function done. If there are more than one thread blocks having the same value, we further rank them based on the thread block id. Once the thread block order is done, the order within thread block is determined in GTO or GTLS fashion: we consider the last issued warp in the thread block, if not available, we pick the warp with the oldest timestamp (GTO) or longest waiting time (GTLS). This 2-level ranking algorithm can effective reduce the cost in sorting the warps, as thread block order only changes if a warp suffers/resolves a barrier/function done, which happens every hundreds cycles. and we only need to update the order within one thread block every cycle if an instruction is issued.

7.4 Result Analysis

In this section, we evaluate five different scheduling policies on 9 Parboil benchmarks:GTO, GTO-TAWS, LRR, GTLS, and GTLS-TAWS. We use the CPI breakdowns and IPC speedup compared with GTO to represent the effectiveness of each scheduling scheme in improving certain key stall factors and IPC.

7.4.1 Overall Performance

Figure 7.1 presents the average CPI breakdown for Parboil applications on five scheduling schemes. The x-axis presents different kernels sorted by the sum of the CPI, and five bars for each kernel represents 5 different scheduling policies, which are (from left

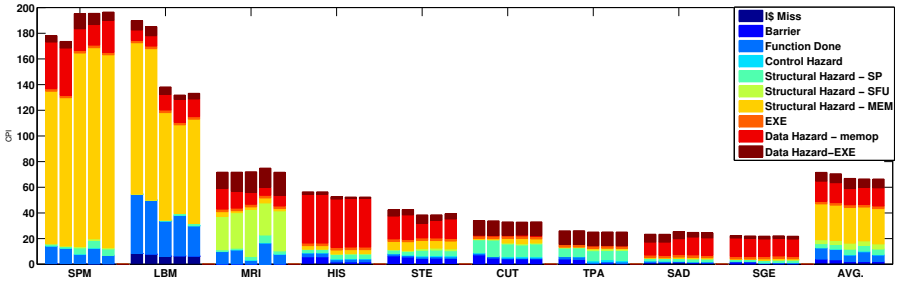

Figure 7.1: The average CPI breakdown of Parboil benchmarks with different scheduling policies: 1. GTO; 2. GTO-TAWS; 3. LRR; 4. GTLS; 5. GTLS-TAWS.

to right), GTO, GTO-TAWS, LRR, GTLS and GTLS-TAWS. The y-axis shows the CPI breakdown attributed by various stall factors described in section 6.3. To better investigate the CPI breakdown, we further breakdown structural hazards into three parts: structural hazards due to SP, SFU and MEM function units; and data hazards into two parts: data hazards due to previous load instruction and previous execution instruction. The CPI breakdown results are the average across all the warps among all the SMs throughout the kernel execution. In general, we observed that kernels on the left exhibit bigger variation against different scheduling policies, and stalls due to structural hazards and function done are most sensitive to scheduling policies, e.g. in LBM, GTLS-TAWS effectively reduces stalls due to structural hazards-MEM from 117.94 cycles per instruction (GTO) to 81.23. Overall, the average CPI goes down from following 5 scheduling schemes from left to right, resulting in 71.59, 70.56, 66.92, 66.48, 66.41, respectively. And the CPI portion due to barrier and function done from those 5 scheduling schemes are, 11.39, 10.42, 5.88, 8.56, 6.02. Thus most of the CPI improvement from LRR and GTLS-TAWS, compared to GTO, comes from barrier and function done stalls. Both LRR and GTLS based scheduling policies are significant better than GTO based scheme, and most of the benefits come from improved stalls due to barrier and function done. This is expected as both LRR and GTLS are designed to maintain similar pace among threads from the same CTA, while GTO is less effective in keeping those threads in the same pace. Furthermore, both TAWS based schemes perform better in reducing stalls due to barrier and function done, and TAWS contributes an

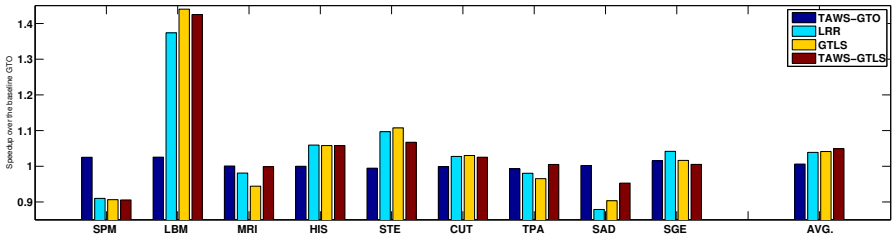

Figure 7.2: The IPC speedup of Parboil benchmarks of different scheduling policies compared with GTO.

average of 8.52% and 19.67% improvement in barrier and function done stalls for GTO and GTLS. Besides CPI breakdown, let's look at IPC improvements.

Figure 7.2 shows the average IPC speedup of 9 Parboil benchmarks compared with GTO. In general, GTO-TAWS, LRR, GTLS and GTLS-TAWS achieve an average of 0.62%, 3.89%, 4.13% and 4.92% IPC speedup compared with baseline GTO. Among all the scheduling schemes, GTLS-TAWS is the best based on the benchmarks we evaluated, it yields the highest average IPC speedup, and individually, it has only 3 benchmarks with IPC slow down, with the biggest slowdown of 9.43% (SPM), all of those metrics are the best among all the scheduling schemes we evaluated. Note it's extremely hard to come up a new scheduling policy that outperforms GTO on all benchmarks. In addition, TAWS adds an additional 0.62% and 0.79% IPC speedup to GTO and GTLS. In the following sections, we will further investigate the CPI breakdown and IPC speedup for some kernels that respond well or badly to GTLS-TAWS.

7.4.2 GTLS, LRR vs. GTO

LBM has the best IPC speedup with GTLS-TAWS compared with GTO, resulting in an average of 42.50% improvement. As shown in Figure 7.1, most of the speedup comes from two CPI components: structural hazards due to MEM and function done. To further investigate how CPI changes for each warp, we count the CPI breakdown from each warp. Figure 7.3 presents the average CPI breakdown from a SM with 5 different scheduling schemes. We notice that the CPI decreases significantly in LRR

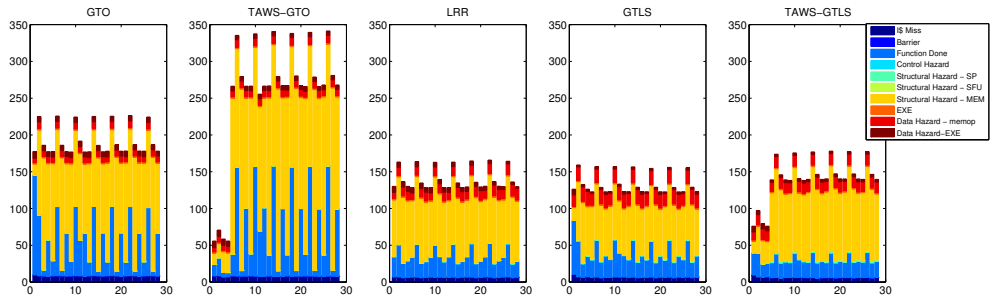

Figure 7.3: The CPI breakdown of LBM for 28 warps, 4 warps per CTA.

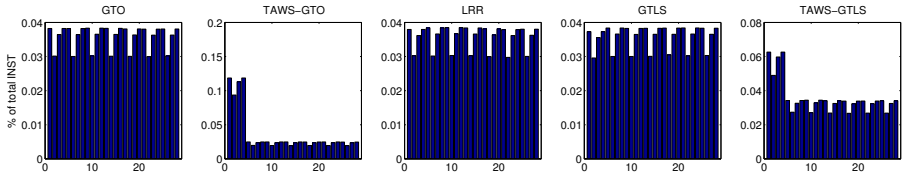

Figure 7.4: The instruction issue percentage of LBM according to warp ID

and GTLS-based policies, this is mainly due to similar pace within CTA in LRR and GTLS. LBM is a memory-intensive kernel that suffers substantial structural hazards due to MEM from MSHRs congestion. Because of data locality among threads within the same CTA, threads from the same CTA often have memory requests that result in the *same* outstanding memory requests to lower memory hierarchy. If those warps progress in a similar pace, all the memory requests from those warps tend to occur around the same time, which results in a lot of *hits* in outstanding memory requests. This greatly saves the limited memory bandwidth. On the other hand, if warps from the same CTA process differently, and memory requests occur at different time, GPU has to address the same outstanding memory requests repeatedly. As a result, this often leads to a waste of memory bandwidth, and even introduces congestion. LBM is an ideal example for this issue. From the CPI distribution in GTO and GTO-TAWS in Figure 7.3, we can see one or two warps often suffer much bigger stalls due to function done, compared to the rest of warps from the same CTA. This indicates warps within the same CTAs are progressing significantly differently, thus resulting more structural hazards due to

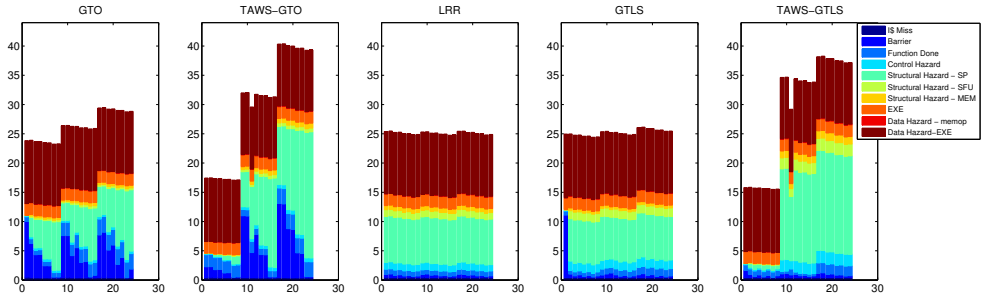

Figure 7.5: The CPI breakdown of TPA for 24 warps, 8 warps per CTA.

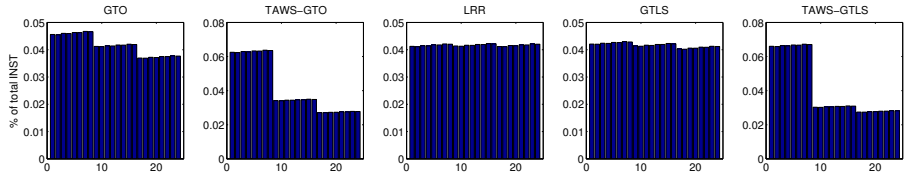

Figure 7.6: The instruction issue percentage of TPA according to warp ID.

MEM, and much lower overall IPC.

Another benefit of keeping warps within CTAs in the same pace is the ability to improve stalls due to barrier and function done. Figures 7.5 and 7.6 show the average CPI breakdown and instruction issue percentage from a SM with 5 different scheduling schemes for TPA. In Figure 7.5, the CPI components due to barrier and function done are very significant, contributes 21.32% of the total CPI in GTO, and even GTO-TAWS only reduces such stalls to 20.21%. However, LRR and GTLS appear to be very effective at keeping such stalls low: such stalls are only 4.68% and 5.47% of the total CPI in LRR and GTLS-TAWS. Note we will not get that many improvement in performance here, as TPA has a relatively high IPC already, then reducing one type of the CPI components might increase some other stalls. For TPA, we observed more structural hazards and control hazards instead, and the overall performance speedup is -1.97% (performance loss) and 1.05% for LRR and GTLS-TAWS respectively.

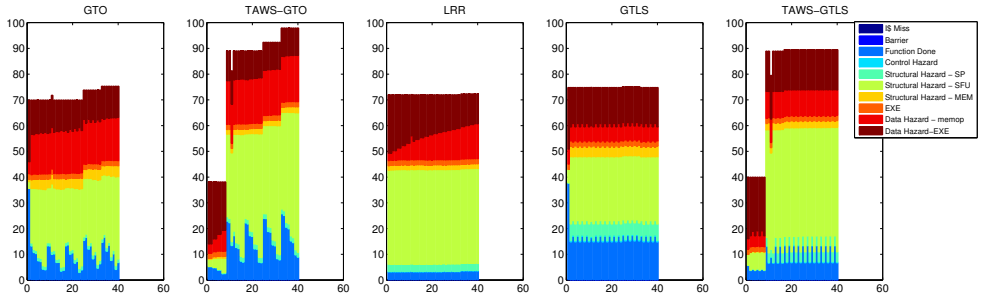

Figure 7.7: The CPI breakdown of MRI for 40 warps, 5 warps per CTA.

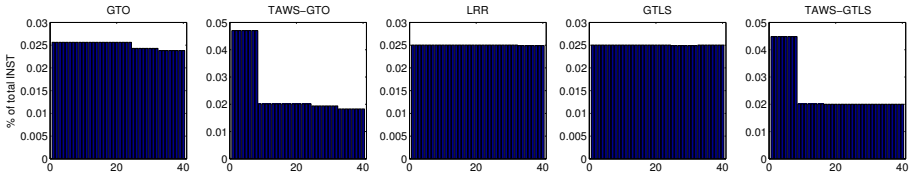

Figure 7.8: The instruction issue percentage of MRI according to warp ID.

7.4.3 TAWS effects

One more observation we get from figures 7.3 and 7.5, is that CPI breakdowns from different warps IDs can be vastly different, this usually happens to TAWS-based schemes, one or more CTAs are having much lower total CPI compared to the rest of CTAs, this will keep different CTAs in different pace. Figure 7.4 shows the instruction issue percentage from different warp ID. Clearly, warps with lower total CPI have much better chance to get issued. However, LBM does not benefit from such characteristics, now we will show how benchmarks get performance boost from such scheme.

As previously discussed, TAWS can effectively reduce the stalls due to barrier and function done, and almost all of the CPI improvement comes from it. Besides, it will keep different CTAs at different pace, we use MRI to evaluate the effect of that. Figures 7.7 and 7.8 show the average CPI breakdown and instruction issue percentage from a SM with 5 different scheduling schemes for MRI. First of all, TAWS favors CTAs that suffer more barrier and function done, and let them finish faster so new CTAs (without barrier and function done stalls) can be issued. Ultimately, this increase the effective number

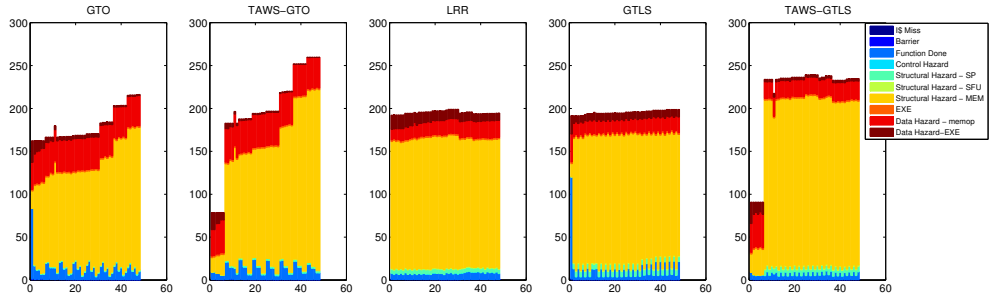

Figure 7.9: The CPI breakdown of SPM for 48 warps, 6 warps per CTA

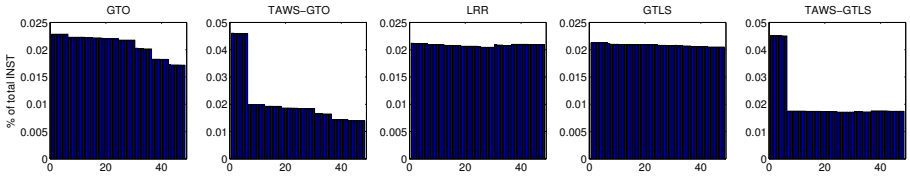

Figure 7.10: The instruction issue percentage of SPM according to warp ID.

of active warps per cycle. In Figure 7.7, GTLS-TAWS improves CPI portion due to barrier and function done from 21.83% (GTLS) to 10.68%. Moreover, because different CTAs are progressing at different pace, we improve structural hazards by spread out the congestion more evenly throughput the execution time, GTLS-TAWS improves CPI portion due to structural hazards from 49.30% (GTLS) to 46.36%. Of course, increased stalls due to data hazards weaken the overall performance boost from TAWS, but TAWS achieves 5.75% performance speedup overall.

7.4.4 SPM

SPM is the only benchmark suffers significant performance loss, resulting in 9.43% slowdown with GTLS-TAWS compared with GTO. Figures 7.9 and 7.10 show the average CPI breakdown and instruction issue percentage from a SM with 5 different scheduling schemes for SPM. Both LRR and GTLS exhibit performance loss, mainly due to increased stalls from structural hazards-MEM, and TAWS does little to improve that. Under further investigation, such structural hazard stalls are not caused by congestion

in MSHRs, instead, they are due to *reservation cache fail* (RCFail) in L2 cache. RCFail happens when all the slots in a cache set are marked "reserved" (waiting for data to be served from lower memory hierarchy) and thus the cache fails to reserve a slot. Once one cache set suffers this stall, the whole memory pipeline has to stall until it's resolved. Therefore, if the access pattern of the L2 cache from all SMs is not evenly distributed to all cache sets, and one of the cache sets fails to reserve a new slot, we will suffer structural hazards. Note this is directly related to the memory access pattern from SMs. For MRI, the memory access patterns from GTLS and LRR are more likely to saturate one cache set, resulting in longer stalls due to RCFail. To sum up, we believe the slowdown of SMP in GTLS-TAWS is just a rare case: 1) very few structural hazard stalls are caused by RCFails; 2) RCFail is access pattern dependent, there is no strong link between longer RCFail stalls with GTLS and LRR.

7.5 Conclusion

In this chapter, we propose GTLS-TAWS, a new two-level priority scheduling scheme, which ranks CTAs based on the number of warps suffering stalls due to barrier and function done, then prioritize warps within CTAs in a greedy then least scheduled fashion. By keeping warps within the same CTA at similar pace, while different CTAs at different progress, GTLS-TAWS can effectively improve stalls due to barrier, function done, and structural hazards. Compared with baseline GTO scheduling policy, GTLS-TAWS reduces CPI components due to barrier and function done by 47.15%, and achieves an average IPC speedup of 4.92%.

Chapter 8

Run-time intra-core multitasking for GPGPUs

In previous chapter, we propose a new two-level scheduling policy that can improve stalls due to barrier and function done *et al.*. However, only limited speedup achieved due to the fact that many applications cannot exploit this massive parallelism due to various resource constraints. Meanwhile, allowing only one kernel running on SMs lacks the ability to balance the availability of heterogeneous resources such as streaming processors (SP), and special function units (SFU), as well as specific components within the memory hierarchy. Analysis of a variety of highly-optimized GPU applications shows that oversubscription of GPU resources limits performance, such that the applications only achieve 35.5% of a GPU's maximum throughput, on average. We observe that since the resource requirements of different applications are different and often complementary, we can improve utilization, reduce contention, and improve performance by simultaneously co-scheduling multiple applications on the same GPU core, i.e., intra-core multitasking (ICMT). We present different ICMT microarchitectures and scheduling mechanisms and demonstrate up to 28.1% average performance benefits for ICMT with only 1.8% area overhead, compared to conventional single-kernel execution with a greedy-then-oldest (GTO) scheduling mechanism. Furthermore, when co-scheduling complementary workloads, the average speedup improves to 39.2%.

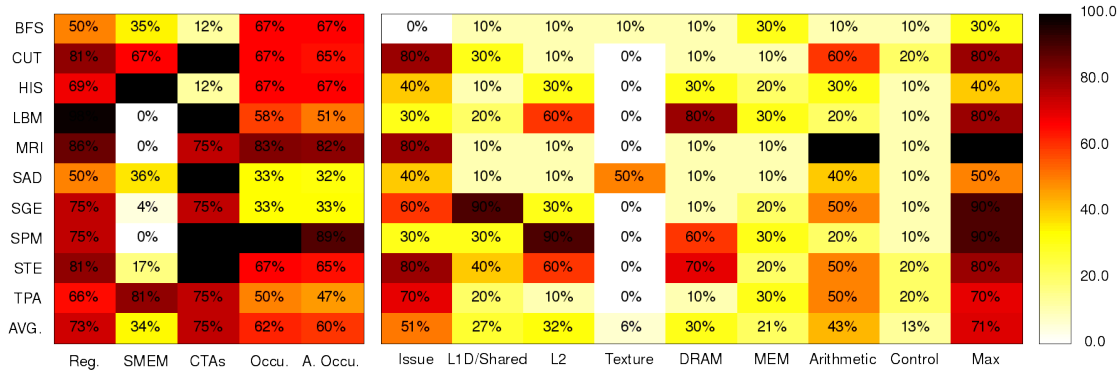

Figure 8.1: Kernels from different Parboil benchmarks exhibit significantly different utilization of hardware resources and function units on a GPU core, possibly indicating that co-scheduling multiple kernels (with complementary resource utilization) on the same GPU core might improve PLP. Occu. and A.Occu. are short for Occupancy and Achieved Occupancy.

8.1 Introduction

General Purpose GPUs (GPGPUs) are a powerful and energy-efficient computing platform for data parallel applications. GPGPUs can accommodate thousands of threads running simultaneously. As such, instead of minimizing latency for an individual thread, GPGPUs exploit *thread level parallelism* (TLP), and allow execution of other threads when some threads stall [58]. To facilitate massively-parallel general purpose computation on hardware designed for graphics processing, programming models such as CUDA [59] and OpenCL [60] have been developed. In these programming models, GPGPU applications are typically divided into several *kernels* that execute sequentially, each of which is composed of many threads that execute in parallel. When kernels are executed, threads are grouped into basic scheduling units called *Cooperative Thread Arrays* (CTAs) and assigned to available GPU cores. CTAs are subdivided into groups of 32 threads called *warps* or *wavefronts*, the basic unit of execution flow. All threads in a warps execute the same instruction stream based on the *single instruction, multiple thread* (SIMT) paradigm [61].

Despite the massive parallelism of modern GPGPUs [62], their throughput often falls far short of their peak capabilities for many parallel computing applications. There are

three primary reasons for this. First, an application may exhibit inadequate parallelism due to limited threads spawned by the application or over-subscription of limited per-core hardware resources such as registers or shared memory. Second, contention in GPU-wide shared resources (especially in the memory hierarchy) can create a performance bottleneck for parallel execution. Third, even if TLP is maximized, different applications/kernels have different instruction mixes and may not fully utilize all the function units (e.g., ALUs, special function units, and memory units) in the GPU. We use the term *pipeline-level parallelism* (PLP), to describe the parallel utilization of different function units. While insufficient TLP may cause *all the function units* on a GPU core to be idle, insufficient PLP can occur when *one of the function units* is underutilized due to an unbalanced instruction mix in an application. For many applications, these limitations result in a sizable gap between actual and peak GPU throughput. For example, for the set of applications that we studied (see Section 6.2.2), we observed that the GPU achieved only 35.5% of its peak throughput, on average.

Researchers have considered co-scheduling kernels concurrently in GPGPUs to improve TLP and PLP [63, 64, 65, 66]. For instance, Adriaens *et al.* [63] propose inter-core multitasking to statically launch multiple kernels on separate GPU cores. Inter-core multitasking can improve TLP, giving a GPU more threads to execute; however, it lacks the capability to improve PLP within GPU cores and primarily improves throughput only when the TLP of one kernel is insufficient to fill the GPU to capacity.

A better approach than launching multiple kernels across different GPU cores might be to co-schedule multiple kernels on the same GPU cores. This approach would not only have the potential to increase TLP but could also improve PLP by balancing the mix of instructions on a GPU core.

Figure 8.1 shows the utilization of the various hardware resources and function units on an Nvidia GTX 480 GPU for different kernels in the Parboil benchmark suite [51]. The figure shows that different kernels can have substantially different utilization profiles. Thus, it might be possible to improve PLP by co-scheduling kernels with complementary resource utilization on the same GPU core.

Previous work explores intra-core kernel co-scheduling on real hardware via offline kernel merging in software [67, 68, 64]. While the work indicates potential for increased parallelism with intra-core co-scheduling, a static approach based on offline software

merging can limit a GPU core's ability to extract PLP. While work on inter-core and intra-core co-scheduling exists, all the previous co-scheduling techniques are based on existing GPGPU architectures and scheduling mechanisms that are designed for homogeneous simultaneous multithreading within a GPU core, where all warps on a core are from the same kernel and exhibit similar behavior, with relatively stable resource requirements.

Evaluating existing proposals for inter-core and intra-core multitasking reveals that existing approaches are unable to significantly improve performance, primarily because they introduce substantial extra contention in shared resources, and partly due to inefficient scheduling mechanisms. Limited memory bandwidth is the main reason for GPU underutilization [58]. Figure 8.2, which characterizes all possible pairings of the 10 kernels listed in Table 6.1 (details in Section 8.6), shows that intra-core multitasking leads to more memory stalls, resulting in 3.38% performance *loss* on average. The problem is not as significant in inter-core multitasking, where average performance improves by 7.50%. Due to the prominent impact of increased memory contention when co-scheduling kernels, for the results in Figure 8.2, we group kernels into two sets based on whether they experience significant memory stalls (MEM-S) or few memory stalls (Non-MEM-S), according to the classification in subsection 8.4.1. Intra-core multitasking introduces 27.72% more memory stalls, on average, than inter-core multitasking when co-scheduling MEM-S with Non-MEM-S and 6.67% more memory stalls when co-scheduling MEM-S with MEM-S. Essentially, co-scheduling a MEM-S kernel with a Non-MEM-S kernel results in a pairing that is memory-constrained (MEM-S). The existing intra-core multitasking only improves performance for 64.0% of the kernel pairs, while 16.7% of the pairs actually suffer over 30% performance degradation.

Based on the results above, co-scheduling different kernels on the same GPU cores exhibits potential to increase TLP and PLP; however, a static approach that does not provision for the extra resource contention introduced by co-scheduling may not effectively exploit the potential benefits and may in fact degrade performance. In this work, we propose an architectural solution for intra-core multi-tasking on GPU cores and explore how to optimize the GPU microarchitecture to enhance the benefits of kernel co-scheduling. This chapter makes the following contributions.

- We perform thorough performance analysis for a set of applications in a modern

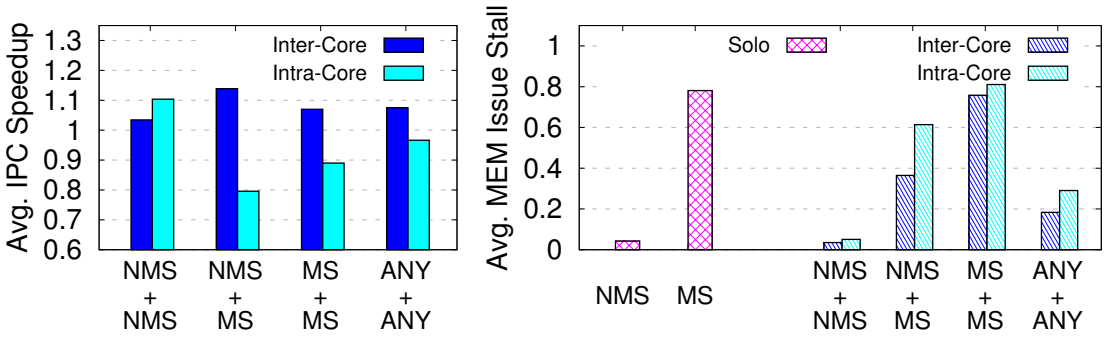

Figure 8.2: Average throughput speedup (G-Mean) and average memory stall rate for existing inter-core and intra-core multitasking. Note *MS* and *NMS* are short for MEM-S and Non-MEM-S.

GPGPU environment, identify their key performance bottlenecks, and demonstrate the potential for throughput improvement from intra-core multitasking.

- We propose architectural support for intra-core multitasking in GPGPUs which alleviates the extra memory contention introduced and show how it can be implemented with minimal changes (1.79% area overhead) to existing microarchitecture.
- We observe 26.01% performance benefits from intra-core multitasking (ICMT) in the baseline microarchitecture. We also show that average speedup can improve to 36.11% with more intelligent co-scheduling (from complementary workloads).
- We perform a simulation-based design space exploration to determine the GPU microarchitecture that maximizes throughput for ICMT-based execution. We find that increasing the front-end by 100% results in an average speedup of 28.07% for ICMT with an area cost of 1.79% with respect to the baseline (GTO).

To the best of our knowledge, this is the first work to propose a complete solution (including hardware and scheduling algorithm) for intra-core multitasking for GPGPUs that is compatible with all GPGPU applications without software modification.

8.2 Related Work

Multitasking in GPGPUs:

The GPU spatial multitasking technique proposed by Adriaens *et al.* [63] alleviates system bottlenecks and improves TLP by partitioning GPU cores among multiple applications, with each core executing in the normal single-kernel fashion. This strategy does not address underutilization (e.g., low PLP) *within* GPU cores and still applies homogeneous simultaneous multithreading per core. In our work, most of the performance improvement comes from the improved TLP, PLP, and memory contention afforded by intra-core co-scheduling.

Gregg *et al.* [67] and Guevara *et al.* [68] first demonstrate the throughput potential of intra-core kernel co-scheduling on real hardware via off-line kernel merging in software. Such software-based approaches are not applicable to all workloads and suffer high overhead. Pai *et al.* [64] implement concurrent kernel execution on real hardware by merging two instruction traces of kernels running alone. Due to the in-order-issue feature of GPUs, merging two instruction traces serializes two kernels with pre-determined instruction ordering. Such merged traces cannot accurately reflect how two kernels interact given different CTA partitions. Lee *et al.* [65] also illustrate the benefit of intra-core multitasking, but their detailed hardware implementation and CTA partition is unclear. Our work is closest to interleaved thread block scheduling proposed in [66], but we offer a complete solution, including hardware modification, and show significant performance improvement over their approach (see INTRA in Section 8.7).

Simultaneous Multithreading:

ICMT in GPGPUs shares some characteristics with simultaneous multithreading (SMT) in CPUs [69, 70, 71]. Like SMT, ICMT has the potential to increase throughput by co-scheduling multiple independent threads of execution onto the execution resources of a single core. In the case of SMT, the main motivation is that independent threads of execution exhibit fewer dependencies, and thus, more ILP. For ICMT, the main motivation is that threads from different kernels may have different and complementary

resource usage such that co-scheduling can improve utilization of varied execution resources (PLP) and also allow threads from one kernel to make progress while threads from another kernel are stalled due to oversubscription of resources.

8.3 Background

This section provides an overview of the framework for intra-core multitasking (ICMT).

8.3.1 High-Level View of Intra-Core Multitasking Framework

Figure 8.3 provides a high-level view of ICMT, showing the changes that are made on top of the baseline GPU architecture. For simplicity, we only consider the scenario of co-scheduling two kernels in this chapter. In principle, though, the proposed approach could be extended to three or more kernels.

When a new kernel is assigned to the GPU, it is placed in the active kernel pool. The *kernel management unit* determines which kernels should be co-scheduled together and decides how many CTAs of each kernel to mix to optimize system throughput. Once the scheduling decision is made, the GPU allocates the CTAs to each core, just as in the single-kernel case. No additional hardware is required to issue CTAs from different kernels, compared to issuing CTAs from the same kernel. However, simultaneous multitasking may place more pressure on device resources, such as the memory system. We investigate the impact of ICMT on microarchitecture and system throughput in Section 8.7.

8.3.2 Evaluation Metric

We use IPC speedup and utilization for the various function units to evaluate the performance of kernels in different GPU configurations.

We use geometric mean (G-Mean) of IPC speedup to measure throughput improvement. G-Mean has been used in previous works on SMT, since it does not favor unfair system configurations in which a kernel with high-throughput is allowed to monopolize system resources at the expense of a low-throughput kernel [72]. For baseline configurations, we consider the performance of two workloads running alone. For function units, we define utilization as the fraction of total execution cycles that a unit is not

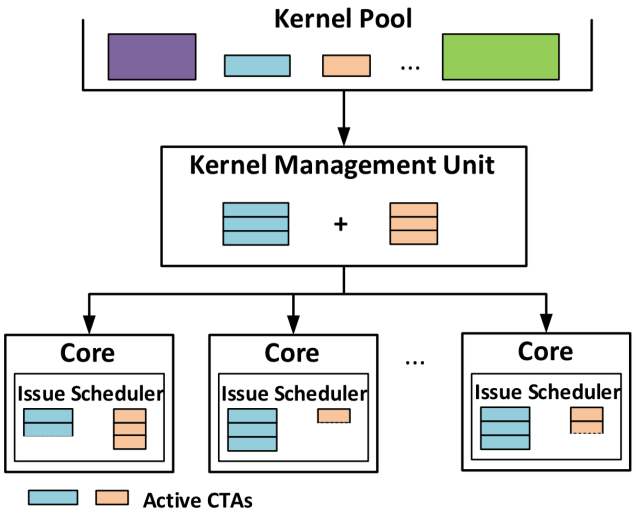

Figure 8.3: High-level view of proposed intra-core multitasking technique.

idle or stalled. We characterize the utilization of the scheduler and all three pipelines (SP, SFU, and MEM).

8.4 Detailed Analysis of TLP and PLP Stalls

8.4.1 Primary Performance Constraints

As discussed previously, underutilization of GPGPUs can be caused by inadequate TLP (i.e., inadequate active warps) and/or PLP. To evaluate the impact of these two factors, we characterize both TLP stalls and PLP stalls for the benchmarks in Table 6.1. We further break down PLP stalls based on the type of function unit (SP, SFU or MEM) that causes the stall. Figure 8.4 shows the breakdown of average stall rate for kernels executing on the baseline architecture, where all the kernels use the maximum number of CTAs allowed per core, with the goal of maximizing TLP. We observe significant variation in performance, TLP stalls, and PLP stalls across the set of kernels. Even kernels from the same application can exhibit significant variation. On average, GPU cores are stalled 43.5% of the time, and PLP stalls contribute 54.0% of the total stalls, primarily due to limited on-chip and/or off-chip memory bandwidth (MEM stalls). These results agree with findings of previous work [56]. As MEM stalls is the primary

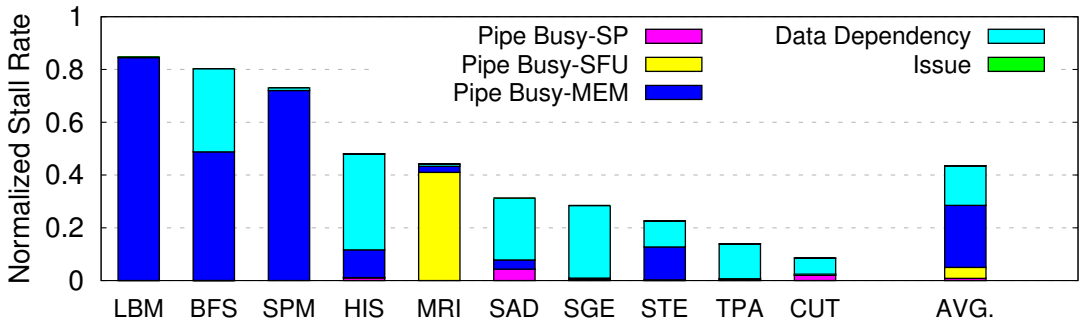

Figure 8.4: Breakdown of average GPU stall rate for kernels executing on the baseline architecture. *TLP stalls* occur when no active warp is available. *PLP stalls* occur when active warps are available but the scheduler cannot issue an instruction to a particular pipeline due to a structural hazard (e.g., the pipeline is stalled due to excessive unresolved off-chip memory accesses or a full pipeline is still busy executing previously-issued instructions).

performance constraints in GPGPUs, for future reference, we categorize all the kernels into two groups: kernels experience significant memory stalls (**MEM-S**) or few memory stalls (**Non-MEM-S**). We consider kernels suffering over 40% MEM stalls as MEM-S. Thus we have *LBM*, *BFS*, and *SPM* in that category. In addition, five kernels suffer stalls due to warp data dependency for over 10% of the cycles (*BFS*, *HIS*, *SAD*, *SGE*, *TPA*). All five have less than 0.66 warp occupancy due to GPU resource constraints (e.g., registers, shared memory, thread contexts). It is also worth mentioning that having a large number of warps is not always enough to prevent stalls when there are too many outstanding long-latency operations. E.g., *SPM* achieves the maximum number of warps that the GPU permits but still experience substantial PLP stalls, since LD/ST unit stalls due to memory congestion. We also observed that even if afforded unlimited TLP and memory bandwidth, the average stall rate in the scheduler is still 17.26%, due to oversubscription of a certain type of function units by some of the benchmarks. For example, kernels such as *LBM*, *BFS*, and *SPM* suffer a stall rate of over 50%, due to significant utilization imbalance among different function units.

Figure 8.5 shows the average utilization of all three types of function units – ALU, SFU, and MEM, as well as MEM stalls. Surprisingly, for certain workloads that do

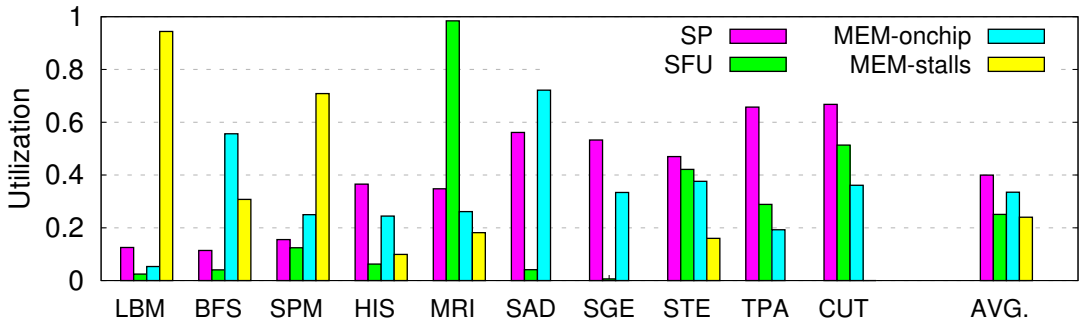

Figure 8.5: Average utilization of SP, SFU, and MEM in the baseline architecture.

not experience significant TLP and PLP stalls (e.g., *CUT*, *TPA*), utilization of different SIMT function units is far from the maximum. Average utilization of ALU, SFU, and MEM are only 66.7%, 39.9%, and 28.8%, respectively, for the above 2 kernels. This is mainly caused by two factors. 1) Poor PLP: The instruction mix in a kernel favors one type of instruction and leaves other function units underutilized. In Figure 8.5, the SP pipeline utilization dominates in 5 kernels, while utilization in SFU and MEM dominate in 1 and 4 kernels, respectively. 2) The GPU scheduler only issues one instruction per cycle and is thus incapable of keeping all the SIMT pipelines fully utilized. For the kernels on the left side of Figure 8.5, performance is primarily limited by scheduler throughput, while on the right side, one particular type of function unit becomes the key constraint on system performance.

While scheduler constraints can be removed by increasing the instruction dispatch and writeback throughput, this microarchitectural change would only be advisable if it results in commensurate performance improvement. We observe that in the baseline architecture, doubling and tripling instruction dispatch/writeback throughput only improves performance by an average of 0.85% and 0.97%, respectively, and incurs an extra 1.79% and 2.70% overhead, respectively, in area. Thus, the limited utilization is not only due to limited dispatch/writeback throughput but is primarily due to the fact that **any single kernel does not contain an appropriately diverse mix of instructions to fully utilize available function units**. The motivational results

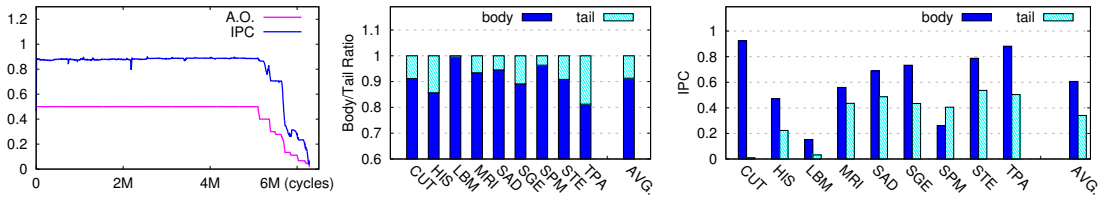

Figure 8.6: The tail effect results in reduced achieved occupancy and IPC for single kernel execution and inter-core multitasking.

above do suggest, however, that **an appropriate mix of kernels may provide adequate instructions to more fully utilize available execution resources**. We will later show that co-scheduling kernels with complementary resource requirements on the same GPU core can significantly improve utilization and performance.

8.4.2 Investigating Memory Stalls

Unlike in ALUs, where structural hazards are only caused by unavailable execution pipelines, structural hazards in the memory can happen at different resources across the multiple levels of the memory hierarchy. First, inside GPU cores, LD/ST units are connected to four different caches. When too many cache misses occur, the memory unit can stall for hundreds of cycles due to a cache reservation failure, unavailability of MSHRs, or a full miss queue. Due to the nature of in-order issue in existing SIMT architectures, once a stall occurs, the entire memory unit issue queue is stalled, preventing any new memory instructions from issuing until the stall is resolved. This is unfortunate, considering that for the benchmarks we studied, stalls account for over 40% of the overall memory pipeline utilization (see Figure 8.5).

Since memory stalls can result from a number of different sources and since different kernels utilize memory system resources differently, we observe an opportunity to reduce memory stalls and improve memory bandwidth efficiency by co-scheduling kernels that are complementary in their utilization of different memory system resources, more details in subsection 8.5.2

8.4.3 Mitigating the Tail Effect

In addition to improved TLP and PLP, another potential benefit of ICMT is mitigation of the tail effect encountered when most of a kernel's CTAs have finished executing and the kernel experiences reduced TLP until the remaining CTAs finish. The left sub-figure of Figure 8.6 illustrates the tail effect for the kernel *TPA*, showing how occupancy and IPC are degraded during the tail end (18.8%) of the kernel's execution. The middle sub-figure of Figure 8.6 shows the percentage of execution time kernels from the Parboil benchmark suite spend in the tail portion of execution. On average the tail accounts for 8.7% of the execution time. The right sub-figure of Figure 8.6 compares the IPC of each kernel during the body and tail portions of its execution. On average, IPC during the tail portion is 44.3% lower than IPC during the body portion. For one kernel (SPM), IPC is higher during the tail because the IPC of the kernel is significantly higher at the end of execution. Over the entire execution of these kernels, the tail effect results in 3.9% performance reduction, on average.

Conventional single kernel execution and inter-core multitasking cannot avoid the tail effect. ICMT, on the other hand, has the potential to mitigate the tail effect, since CTAs from one kernel can fill in to maintain higher occupancy while another kernel experiences its tail. ICMT cannot completely eliminate the tail effect, but it can significantly reduce the impact of the tail effect, as we will demonstrate in section 8.7.

8.4.4 Potential Benefits of Intra-core Multitasking

To summarize, the results in Figures 8.4 and 8.5 for a mix of 10 kernels show that a current GPU core only utilizes 35.55% of its massive throughput potential due to inadequate TLP, scheduler constraints, and insufficient PLP. Ideally, with increased scheduling throughput, intra-core multitasking can address all of these bottlenecks and significantly improve average aggregate throughput. Also, unlike existing approaches [58], since it considers both TLP and PLP stalls, intra-core multitasking has the potential to ameliorate performance bottlenecks without causing another bottleneck to arise.

8.5 Architectural Design Space Exploration

In this section, we explore architectural modifications that may improve the performance of a GPU that supports intra-core multitasking. We primarily focus our exploration on architectural resources that may become bottlenecks in a multi-kernel execution environment.

8.5.1 Instruction Dispatch and Scheduling Bandwidth

One motivation for ICMT is that kernels with complementary resource usage can be co-scheduled on the same GPU core to increase utilization of varied execution resources and subsequently enhance throughput. When complementary kernels are co-scheduled, the dispatch unit in the baseline architecture (Figure 6.1), which only issues one instruction per cycle, may impose a bottleneck to exploiting the additional PLP exposed by ICMT. Consequently, we explore the impact of doubling the fetch, decode, and dispatch bandwidth (all processing logic before issue queue, including doubling the number of I-cache ports), while keeping the SIMT pipelines unchanged. To match the increased dispatch bandwidth, we also increase the scheduling bandwidth such that up to one instruction of each type (SP, SFU, MEM) can be dispatched per cycle to its dedicated pipeline, provided there is sufficient TLP and the corresponding issue queue is not full. I.e., instead of picking one active warp to issue per cycle in the warp scheduler, we tag active warps based on the type of their next instruction (SP, SFU, MEM), and the scheduler picks up to one instruction of each type using an existing scheduling mechanism, like GTO.

8.5.2 Prioritized Memory Issue Queue

As discussed in Section 8.4.2, the baseline architecture has only one issue queue to the memory pipeline. When stalls occur due to limited memory bandwidth, the entire LD/ST unit is stalled until the congestion is resolved, potentially for hundreds of cycles. This could limit the benefits of ICMT, since a stalled LD/ST unit can stall both kernels, even if one of the kernels does not require the congested memory resource. E.g., a memory-intensive kernel co-scheduled with an ALU-intensive kernel could congest available memory bandwidth, forcing both kernels to suffer long stalls in the LD/ST

unit.

Kernels often exhibit diverse memory bandwidth requirements and are affected by different types of memory stalls. Therefore, we explore using a prioritized memory issue queue (PMIQ) that allows kernels with complementary memory resource requirements to interleave memory instructions and avoid stalls when a resource required by only one kernel is congested. For instance, if one kernel is stalled by the memory due to insufficient MSHRs, we can still issue from threads of another kernel that is accessing shared memory.

Prioritization function of PMIQ: In the PMIQ, memory instructions are tagged with their kernel number, and one kernel is designated as having priority. Every cycle, the arbiter picks up to one memory instruction from the kernel with priority and updates the priority designation. The kernel priority toggles in the next cycle if: 1) the high priority kernel suffers a memory stall, or 2) neither kernel stalls on its latest memory instruction. When a memory stall occurs due to one kernel, memory accesses from the other kernel can proceed if they do not require the same type of resource causing the stall. Note that this arbitration mechanism maintains fairness among kernels when neither stalls. If stalls occur, the arbiter favors the un-stalled kernel until the stall is resolved.

Figure 8.7 shows how the microarchitectural modifications discussed in Sections 8.5.1 and 8.5.2 fit into the baseline architecture.

8.5.3 Hardware Overhead

We use McPAT 0.8 [73] integrated with gpgpu-sim 3.20 and GPUWattch [74] to estimate the area overhead of different architectural design points. Based on the area breakdown, shown in Table 8.1, the overhead of incorporating the architectural modifications described in Sections 8.5.1 and 8.5.2 (2X instruction fetch unit, L1 instruction cache ports, and warp scheduler) is $0.7576mm^2$ per core (at $40nm$) or $11.364mm^2$ for the entire processor. This represents a 1.79% area increase for GTX480.

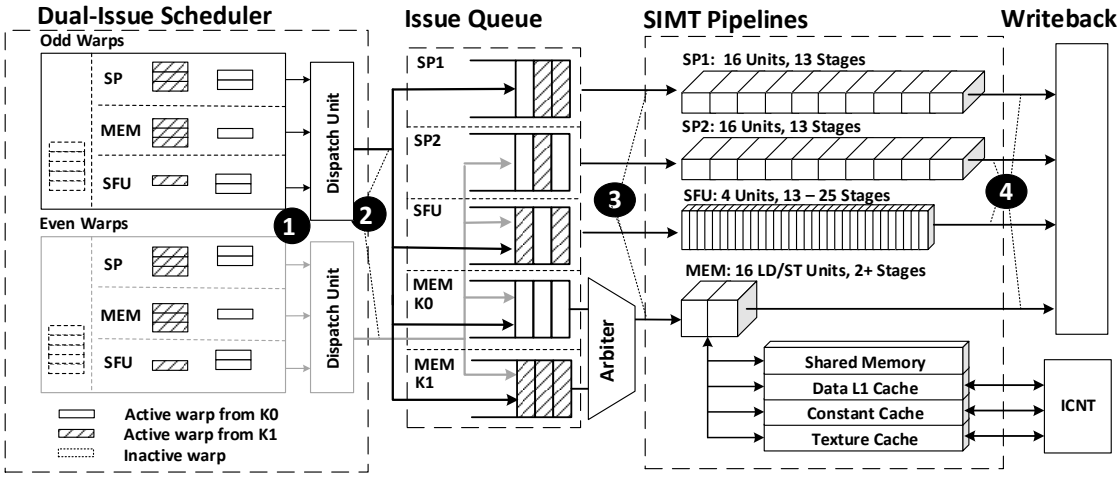

Figure 8.7: ICMT Architecture with increased frontend bandwidth and PMIQ.

Table 8.1: Die area breakdown for Fermi GTX 480, 40nm.

Processor		**Core**	
Module	Area (mm^2)	Module	Area (mm^2)
Total Cores (16)	636.207	Instruction Fetch Unit	0.72424
L2 Cache	9.88038	–L1 Instruction Cache	0.366372
NoCs	0.08929	Load Store Unit (L1 Caches)	2.45388
MCs	28.9458	–Shared Memory	0.6635
Total	675.123	-Execution Unit	35.2924
		-Register Files	2.3079
		–Warp Scheduler	0.01517

8.6 Methodology

We evaluate 9 kernels using GPGPU-Sim 3.2.0 modified to support ICMT. The modified simulator has the capability to co-schedule any two kernels with a CTA allocation that fits within the GPU's resource limitations. Out of the 36 possible kernel pairs that are possible from pairwise combinations of 9 kernels, we simulate all pairs for each scheduling mechanism. One kernel in Parboil benchmark is ruled out to support ICMT, *BFS* only has one CTA, ICMT with *BFS* appears similar to inter-core multitasking. All initial CTA partitions allocate threads evenly from each kernel. Each kernel pair is simulated until both kernels finishes at least once. A kernel is reissued immediately if

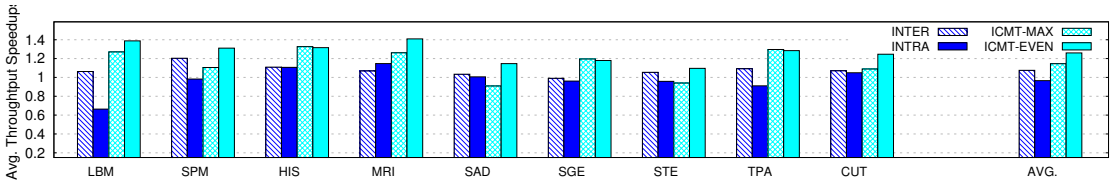

Figure 8.8: Average throughput speedup (G-Mean) of co-scheduled kernels with different scheduling mechanisms.

it finishes early, as if there are multiple invocations from the kernel. This models a real application environment. The simulation period is long enough to capture the entire execution of all kernels, including tail effects.

8.6.1 Scheduling Mechanisms

We simulate all kernel pairs with the following scheduling mechanisms. For the single-kernel baseline we use for comparison against multi-kernel configurations, each of the kernels is executed individually and the performance of each kernel is weighted by half to determine the combined performance.

Unless noted otherwise, all scheduling mechanisms use the greedy-then-oldest (GTO) policy [55]. GTO performs poorly when co-scheduling multiple kernels, especially when warps from different kernels have different lengths. GTO favors kernels with longer warps, as warps that last longer are seen as "older". Therefore, in order to maintain fairness among co-scheduled kernels, we propose a **two-level GTO** scheduling policy. In any given cycle, one kernel has priority over the other kernel, and the scheduler swaps the priorities of the two kernels every cycle. Among warps within the same kernel, instructions are selected according to GTO.

When co-scheduling kernels on the same GPU core, we must implement a policy for selecting the ratio of CTAs to allocate from each kernel. In this chapter, we use heuristics for CTA allocation, as described below. Conventionally, a goal for CTA allocation is to maximize occupancy to provide increased TLP. Therefore, we evaluate one allocation strategy that maximizes the number of warps that can be allocated to each GPU core (ICMT-MAX). We also evaluate an allocation strategy that attempts to balance the number of warps from each kernel while still remaining within a threshold (6 warps)

of the maximum occupancy (ICMT-EVEN). For this strategy, we choose the allocation that is closest to even allocation between kernels and has occupancy within 6 warps of maximum for a given kernel pair. In section 8.7 we test how our heuristics perform with respect to the optimal CTA allocation. However, we leave optimal CTA partitioning for co-scheduled kernels as a topic for future work.

- **INTER**: Inter-core multitasking technique described in [63] – We assign one kernel on 7 cores and the other on the remaining 8 cores.
- **INTRA**: Intra-core multitasking technique described in [66] – Kernels are allocated evenly per core and scheduled with a two-level GTO scheduling mechanism.
- **ICMT-MAX**: Default intra-core multitasking on our proposed architecture including prioritized memory issue queue (PMIQ) described in section 8.5 – Kernels are allocated to maximize occupancy and scheduled with a two-level GTO scheduling mechanism.
- **ICMT-EVEN**: Same as ICMT-MAX, except kernels are allocated evenly per core.

8.7 Experimental Results

8.7.1 Performance of ICMT

Figure 8.8 presents the G-Mean of the speedup achieved with different co-scheduling approaches, relative to single kernel execution. Each bar shows the average speedup of one kernel co-scheduled in pairs with all other kernels. ICMT-EVEN performs best, achieving a 26.01% average speedup, with only one pair suffering a slowdown of merely 0.21%. Note that this average speedup even includes co-scheduling all memory-intensive kernels together, as well as co-scheduling kernels that suffer from the same source of contention. So, results confirm that ICMT does not degrade performance even when only poor co-scheduling choices are available, and in most cases, improves performance considerably. The results also confirm that higher system throughput is possible if co-scheduling pairs are selected more intelligently, such that co-scheduled kernels have complementary resource utilization. INTER and ICMT-MAX achieve average speedups

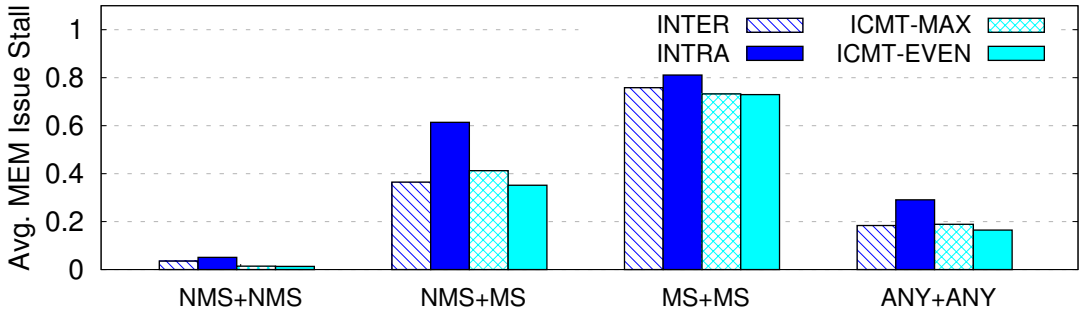

Figure 8.9: Average issue stall due to memory contention with different scheduling mechanisms.

of 7.05% and 14.6%, respectively, while INTRA results in a 3.38% performance loss due to the added contention it introduces. This indicates that performance improvement stems from architectural modifications to support ICMT, like PMIQ, and is enhanced by prudent co-scheduling decisions, like selecting a good CTA partition between co-scheduled kernels.

As noted above, co-scheduling kernels can actually reduce performance if the co-scheduling approach is not designed to account for the resource limitations and potential performance bottlenecks of the architecture. We evaluate how various co-scheduling approaches fare with respect to two major potential system bottlenecks – memory pipeline stalls due to contention in the memory system and TLP stalls due to limited parallelism. Figure 8.9 shows issue stalls resulting from memory contention for different types of kernel pairings. For these pairings, we group kernels into two sets – workloads with significant memory stalls (MEM-S) and those with few memory stalls (Non-MEM-S), based on the issue stall rate breakdown described in subsection 8.4.1. The figure shows the average issue stall rate of co-scheduling all possible pairs between two groups. The results confirm that an approach like INTRA, which does not account for system bottlenecks results in greater memory contention. ICMT-EVEN, on the other hand, exhibits the lowest stall rate.

Figure 8.10 shows issue stalls caused by limited parallelism. ICMT-MAX minimizes stalls due to limited TLP, since it chooses the CTA partition with maximum occupancy.

ICMT-EVEN and INTRA achieve similar stall rates, due to enhanced TLP provided by co-scheduling multiple kernels on a GPU core. INTER achieves a similar stall rate (12.9%) to single kernel execution (14.9%), since both techniques only execute one kernel per core, and INTER primarily has benefit when a kernel is not able to fill the GPU cores with CTAs.

Tail Effect Mitigation

As described in subsection 8.4.3, ICMT can mitigate the tail effect encountered when a kernel experiences waning parallelism as it runs out of CTAs to issue by supplementing with CTAs from a co-scheduled kernel. Figure 8.11 characterizes the impact of ICMT on the tail effect. The left sub-figure shows achieved occupancy and IPC for two co-scheduled kernels, demonstrating that as the occupancy of one kernel decreases, the other kernel covers the gap, maintaining high throughput.

The middle sub-figure in subsection 8.4.3 compares the percentage of execution time spent in the tail segment for different kernels running alone (Solo) and in ICMT. ICMT results are averaged for a kernel co-scheduled in all possible pairs. On average, ICMT reduces the tail by 23%. In one case (HIS), the tail is slightly longer for ICMT than Solo. This is possible if the tails of two kernels coincide; however, this case also mitigates the tail effect since the kernel tails overlap.

The right sub-figure of subsection 8.4.3 compares kernels running alone and in all possible ICMT pairs in terms of IPC degradation introduced by the tail effect. The metric used is $tail\ IPC/body\ IPC$, so less IPC degradation during the tail results in a higher metric value. On average, ICMT reduces IPC degradation by 38%. There is one case (SPM) that shows opposite results. This is because the IPC of SPM spikes at the end of the kernel.

8.7.2 Optimizing Instruction Dispatch and Scheduling Throughput

As discussed in Section 8.4.1, the instruction dispatch and scheduling unit can potentially become the primary performance bottleneck once inadequate TLP, PLP, and memory stalls are reduced by intra-core multitasking. Figure 8.12 illustrates how performance changes when front-end throughput is varied from 1× to 2× and 3× in different

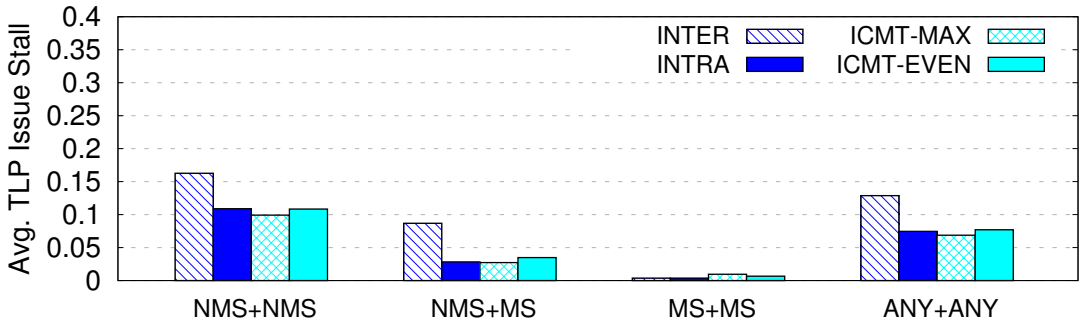

Figure 8.10: Average issue stall due to limited TLP with different scheduling mechanisms.

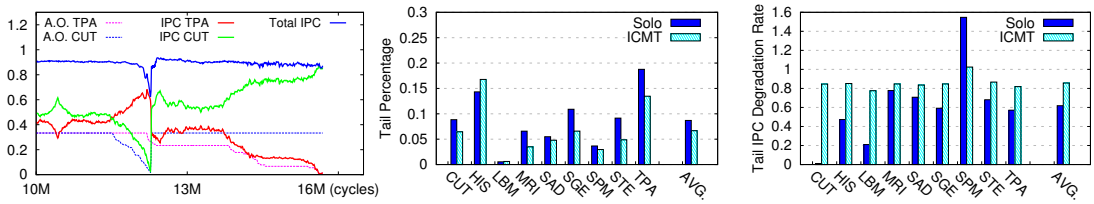

Figure 8.11: ICMT can mitigate the tail effect, resulting in sustained occupancy and higher throughput.

co-scheduling approaches. The figure shows speedup relative to single kernel GTO with single issue per cycle. ICMT achieves 28.1% and 24.4% performance speedup, on average, for 2x and 3x frontend throughput, respectively. Surprisingly, 3x front-end throughput does not improve speedup at all and is even slower than the 26.01% speedup achieved by single-issue ICMT. This is because GTO is not good at maintaining fairness between multiple kernels. A simple CTA partition here is less efficient once more warps can be issued at the same time, and contention increases for system resources making even CTA partitioning far from optimal. With greater front-end throughput, the performance improvement of ICMT can be enhanced further by more intelligent co-scheduling. Complementary workload pairs can achieve 39.2% speedup with 2X throughput, and 36.1% speedup with 1X throughput. Based on the analysis above, we find the optimal design point to be 2X front-end throughput, which incurs a 1.79% area overhead.

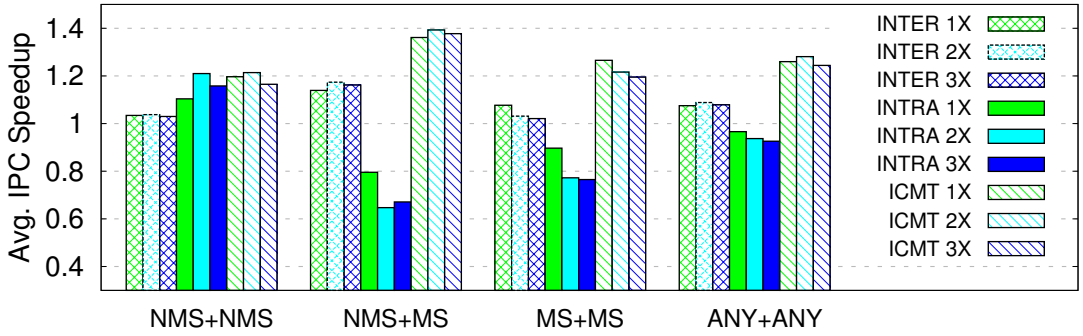

Figure 8.12: IPC speedup by increasing front-end throughput with different scheduling mechanisms.

Limitations of ICMT

We observe that two types of workloads tend to not benefit from ICMT. 1) Workloads that suffer excessive memory stalls that are not alleviated by issuing fewer CTAs – ICMT can benefit from alleviated memory stalls since fewer CTAs are assigned from each kernel. However, if one kernel is extremely memory intensive, the kernel can still cause too much memory contention when co-scheduled with another kernel. 2) Workloads with very low occupancy – If a kernel has low occupancy due to oversubscription of system resources, it can also limit the resource utilization of other kernels sharing the same SM.

More Intelligent CTA Partitioning

In addition to the increased resource contention imposed by previous co-scheduling approaches, we also observe the detriment of a naive CTA partition between co-scheduled kernels, and by contrast, the importance of finding a "good" CTA partitioning strategy. Figure 8.13 shows IPC and utilization for co-scheduling of the kernel pair <SPM, STE> with various CTA partitions. The right sub-figure shows that without multi-tasking, SPM is characterized as memory bound and STE is computation bound. STE alone ($< 8, 0 >$) has high SP utilization; SPM alone (<0,8>) has low MEM utilization but suffers from long off-chip memory stalls. Mixtures of SPM and STE complement one

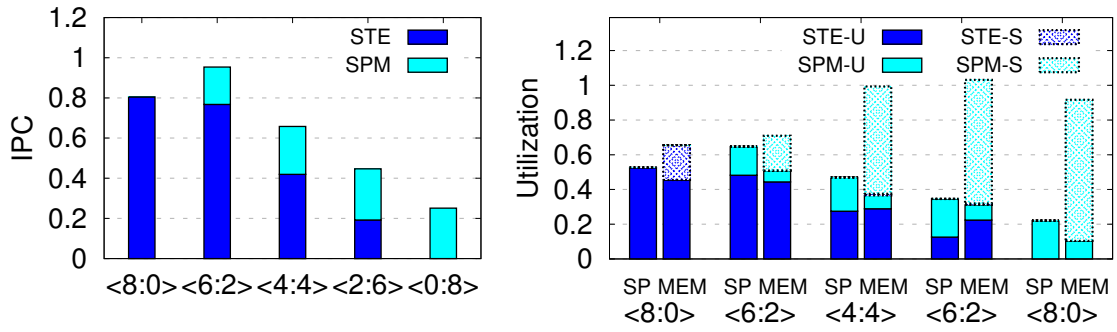

Figure 8.13: Breakdown of IPC and utilization under intra-core multitasking of <STE, SPM> with various CTA partitions. "S-" indicates memory stalls and "U-" indicates effective utilization.

another in the memory pipeline. Fewer SPM threads help to alleviate off-chip memory contention while STE helps hide long latencies via on-chip memory accesses. The right sub-figure demonstrates that some intelligence is needed in determining the CTA partition of co-scheduled kernels that maximizes performance. A naive <4,4> partition has significantly lower IPC than a <6,2> partition. With an optimal partition between co-scheduled kernels, we can achieve substantially higher utilization of execution resources. While we recognize the potential benefits of selecting an optimal CTA partition, we leave optimal CTA partitioning in ICMT as future work, relying for now on our heuristic, which demonstrates good performance.

8.8 Conclusions

Modern GPUs seldom reach their massive throughput potentials due to inadequate TLP and oversubscription of system resources. Co-scheduling of different kernels within or on different GPU cores has been proposed as a means of improving utilization and throughput. However, previous co-scheduling approaches had limited impact on throughput because of their tendency to increase contention for limited resources, as well as their naive approach to co-scheduling. In this chapter, we proposed a full, detailed solution for intra-core multitasking (ICMT), including architectural support and a contention-aware approach to co-scheduling that improves TLP and PLP in a balanced fashion.

We demonstrated 28.07% average performance benefits for ICMT with only 1.79% area overhead, compared to conventional single kernel execution.

Chapter 9

Performance modeling for intra-core multitasking on GPUs

In the previous chapter, we proposed ICMT, a full, detailed solution for intra-core multitasking for GPGPUs, including architectural support and a contention-aware approach to co-scheduling that improves TLP and PLP in a balanced fashion. We also observed the detriment of a naive CTA partition between co-scheduled kernels, and by contrast, the importance of finding a "good" CTA partitioning strategy. However, making a desirable static scheduling decisions including which applications to be combined and the exact thread partition among those applications is challenging. Dynamic interactions between the co-scheduled kernels when contending for resources inside SMs must be taking into consideration.

In this chapter, we propose a computationally-efficient static analytical prediction model that determines thread partitions in fine-grained spatial multitasking to achieve optimal GPU throughput. A key feature of our technique is a novel *fine-grained* kernel mixing algorithm that determines a pairing of kernels that provides optimal or near-optimal throughput enhancement. Then we propose a new warp scheduling algorithm for mixing applications that further improves performance by avoiding local starvation in SIMT pipelines.

The remainder of the chapter is organized as follows. section 9.1 provides background

and motivation. section 9.2 describes the framework of our proposed architecture enhancements and our overall approach. section 9.3 describes the analytical performance model used to make optimal kernel partitions. Static analysis requirements of the model are explained in section 9.5. section 9.6 proposes an inter-warp scheduling policy that avoids local execution pipeline starvation while mixing kernels. Sections 9.7, 9.8, and 9.9 present our evaluation methodology, results, and related work. Finally, section 9.10 summarizes and concludes the chapter.

9.1 Background and Motivation

9.1.1 Terminology

We define some basic terms that will be used throughout this chapter.

Issue interval, denoted as C_p, represents the average number of cycles it takes for one instruction being issued to pipeline p. *Pipeline latency*, denoted as $latency_p$, is the the average number of cycles of one instruction executing in the pipeline p before write back stage. Note $latency_p$ is much longer than C_p, and both C_p and $latency_p$ vary across different SIMT pipelines and are determined by the instruction mix of the kernel. Thus, C_p and $latency_p$ are considered kernel parameters that can be obtained through profiling.

Pipeline utilization, denoted as U, is represented by the probability of the pipeline executing without stalls throughout the execution. As all the cores of the same pipeline have the same utilization due to SIMT paradigm, the utilization of pipeline p can be expressed as $U_p = IPC_p \times C_p$, where IPC_p is the IPC corresponding to pipeline p. Note by definition, the maximum utilization, denoted as U_{max}, is 1 as the pipeline is occupied 100%. Considering system utilization of a SM as the sum of the utilization of SP, MEM and SFU units, thus the system utilization, denoted as U_{system}, can be derived as follows,

$$U_{system} = \sum_{p \in \{SP,MEM,SFU\}} U_p \times \#\text{ of cores in pipeline } p. \tag{9.1}$$

In this chapter, we use U_{system} as the throughput metric to evaluate our kernel mixing technique.

9.1.2 Key Performance Bottlenecks in SMs

To optimize the system utilization in Equation 9.1, we explore the existing constraints in current architecture that limits the shader performance or IPC. As discussed in previous chatper, there are three key performance bottlenecks that can potentially determine IPC:

- **Parallelism constraints.** Every cycle, the scheduler can only issue an instruction if there is at least one active warp. Poor parallelism leads to substantial stalls in the pipelines.
- **Dispatch unit constraints.** Every cycle, the dispatch unit can only issue limited instructions to dedicated SIMT pipelines, thus IPC is no large than dispatch unit throughput.
- **Pipeline constraints.** For each individual SIMT pipeline, the pipeline utilization can never exceed 100%.

In any cycle t, one of three constraints becomes the critical bottleneck that determines the system performance $IPC(t)$. Therefore, given $N_{active}(t)$ is the number of active warp in cycle t and $U_{max\_dispatch}$ denotes the maximum dispatch unit throughput, the above three constraints of the IPC can be formulated as follows,

$$\begin{cases} IPC(t) \leq N_{active}(t), \\ IPC(t) \leq U_{max\_dispatch}, \\ IPC_p(t) \times C_p \leq U_{max}, \end{cases} \tag{9.2}$$

where $p \in \{SP, MEM, SFU\}$. Assuming the behavior of the kernel is relatively stable over the whole execution, only one bottleneck in Equation 9.2 is dominant and thus can be used as the *critical* constaints to determine the shader performance, the rest of the constraints are thus considered non-critical.

Note within SMs multitasking technique can boost the system throughput as it improves both parallelism and pipeline constraints to some extent. However, those two constraints often affect each other, overly improving the critical constraints can sometimes make the non-critical constraints become critical. Therefore, it is challenging to find the optimal scheduling decision that can balance all the constraints.

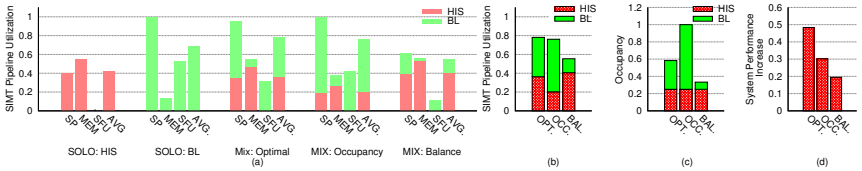

Figure 9.1: An example of multitasking within SMs w/ (HIS, BL) pair. (a)SIMT pipeline utilization of two kernels running alone and within SMs multitasking w/ 3 different kernel partitions. (b) Avg. pipeline utilization w/ different kernel partitions (c) Occupancy breakdown w/ different kernel partitions (d) Avg. system performance improvement w/ different kernel partitions .

9.1.3 Motivational Example

To illustrate how scheduling decision affects the system performance by changing parallelism and pipeline constraints, we simulate two kernels in a kernel mixing scheme under different thread partitions. We pick two kernels with extremely different characteristics. *BlackScholes* (BL) is a compute-bound kernel with good parallelism while *Histogram* (HIS) is a memory-bound kernel that is significantly underutilized due to poor parallelism. Two intuitive thread partition policies are picked here, with the aim to improve parallelism and pipeline constraints of the shader.

- *"occupancy"* (OCC) tries to improve parallelism constraints by picking the thread partition that maximizes the occupancy of the shader.
- *"balance"* (BAL) aims to balance the ALU and MEM utilization. Given the steady-state pipeline utilization partition unknown, it assumes the pipeline utilization in steady state is proportional to the number of threads initially assigned from each kernel.

For comparison, *"optimal"* (OPT) is the thread partition with the highest system throughput improvement, where system throughput improvement is compared with kernel running individually under the same workload.

Figure 9.1(a) shows the pipeline utilization breakdown in 5 scenarios: 2 kernel (HIS and BL) running alone, and kernel pair (HIS, BL) mixing under 3 different partitions. As indicated in Figure 9.1(a), OCC has very good parallelism, but low utilization in

MEM pipeline limits the system performance. While poor parallelism from BAL partition leads to substantial performance loss from a shader with balanced SIMT pipelines. Figure 9.1(b, c, d) illustrate the comparison of OCC, BAL and OPT partitions in pipeline utilization, occupancy and system performance improvement respectively when mixing (HIS, BL). There is at least 20% system performance increase when multitasking within SMs under different thread partitions, as shown in Figure 9.1(d). However, there is still an significant gap between the two initiative partition policies and OPT. Such underutilization is mainly due to overly improving one constraints, causing other constraints become critical. Therefore, to fully take advantage the within SMs multitasking technique, we need to address both pipeline and parallelism constraints together. As indicated in Equation 9.2, it requires the prediction of N_{active} and pipeline utilization of each kernel in steady state. As shown in Figure 9.1(b) and (c), initial thread partition ratio is not sufficient enough to indicate the steady-state performance/ utilization partition. Therefore, to have a within SMs multitasking technique that can always permit optimal or near-optimal scheduling decisions, we need to develop an analytical performance model that can predict kernel IPC in steady state given any thread partition.

9.2 System Framework

9.2.1 Fine-grained Multi-tasking within SMs

The top level framework for the proposed partitioning technique is presented in Figure 9.2(a), showing changes that are made on top of the current GPU structure. When a new kernel is assigned to the GPU, we place it in the *kernel pool*. The *kernel management unit* determines the optimal pair of kernels from the kernel pool that can run concurrently, as well as the detailed grid allocation among the kernels that optimizes the system throughput. This determination is made by gathering certain kernel-dependent information during *static program analysis*. All possible kernel mixes, along with their characterized profiles, are fed into the *performance model*, and the optimal kernel mix and grid partitions are determined. This solution is sent to kernel distributor, which allocates the kernel grids from multiple kernels to each SM, as if they are from the same kernel.

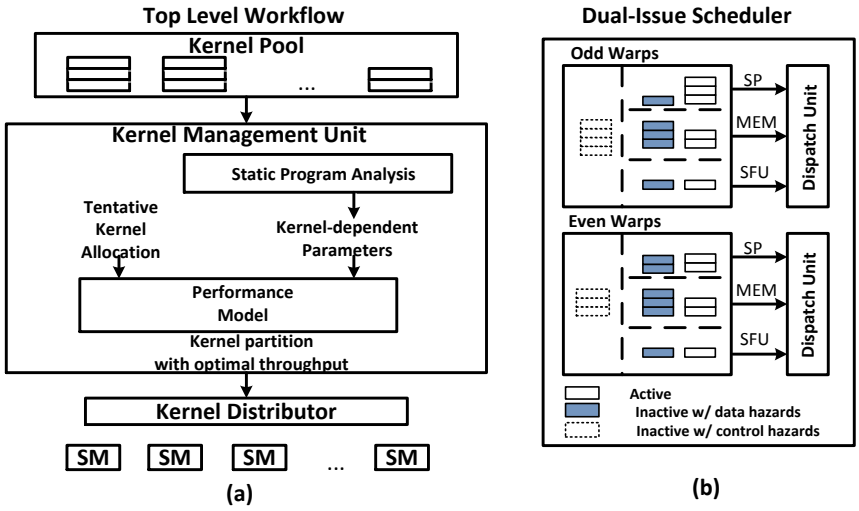

Figure 9.2: (a) Overall System Framework of Fine-grained Kernel Mixing and Grid Partitioning Technique, (b) Hardware Modification on Dual-Issue Scheduler

9.3 Analytical Performance Model

We consider the situation where two kernels, $k1$ and $k2$, are mixed together. We use the subscript $k1$ or $k2$ to refer to the value of a parameter in the corresponding kernel. The subscript p is used to refer to a specific pipeline. For example, $IPC_{k1,p}$refers to the IPC of pipeline p with respect to instructions from kernel $k1$. Clearly, $IPC_{k1} = \sum_{p \in \{SP,MEM,SFU\}} IPC_{k1,p}$.

9.3.1 Problem Formulation and Assumptions

With the scheduler modification, the performance prediction problem can be formulated as follows. Let N_k be the number of warps initially assigned per SM from kernel k. Predict the steady-state IPC_k of each kernel during concurrent execution.

We make the following assumptions:

1. For each kernel, with a fixed scheduling policy, IPC is considered stable throughout the kernel's execution. In the case of multiple non-repeating phases with distinct patterns and instruction classification behavior, non-repeating phases should be modeled separately.

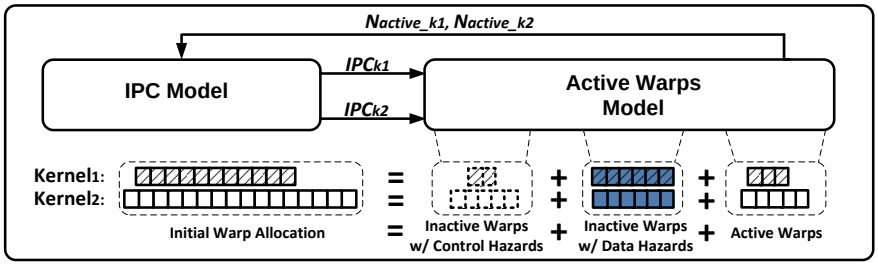

Figure 9.3: Detailed Mean Value Based Performance Model

2. In the steady state, the number of warps issued to each SIMT pipeline in the time interval $(0, T]$ is a Poisson process with an arrival rate of $IPC_p \times T$, where $p \in \{SP, MEM, SFU\}$. If the SM has multiple kernels running concurrently, IPC of each kernel also remains stable.

As a result, a mean value based prediction model is good enough to provide decisions that result in optimal or near-optimal throughput for mixed kernel partitions. Given that parallelism and pipeline constraints are the only two performance bottlenecks, Equation 9.2 can be written as follows.

$$\begin{cases} \sum_{p\in\{SP,MEM,SFU\}} IPC_{k,p} \leq N_{active,k}, \\ \sum_{k\in\{k1,k2\}} IPC_{k,p} \times C_{k,p} \leq U_{max}, \end{cases} \tag{9.3}$$

Note that $N_{active,k,p}$ is the number of active warps from kernel k with next instruction to be issued in pipeline p right before the warp scheduler issues the next instruction. We refer to the total number of active warps as $N_{active,k} = \sum_{p\in\{SP,MEM,SFU\}} N_{active,k,p}$.

$N_{active,k}$ reflects how aggressively one kernel can utilize SIMT pipelines during contention. Therefore, as shown in Equation 9.3, if we can find the $N_{active,k}$ in each pipeline, we can derive IPC_k. On the other hand, $N_{active,k}$ can be derived by $N_{inact,k}$ subtracted from N_k, and $N_{inactvie,k}$ can also be derived from IPC_k, as IPC_k has a direct impact on how many warps become inactive due to data hazards.

9.3.2 Mean Value Based Performance Model

The goal of the performance model is to predict the steady-state IPC_k and $N_{active,k}$. Figure 9.3 illustrates the performance model for a given kernel pair with detailed warp status distribution in steady state. The figure shows a warp status distribution during

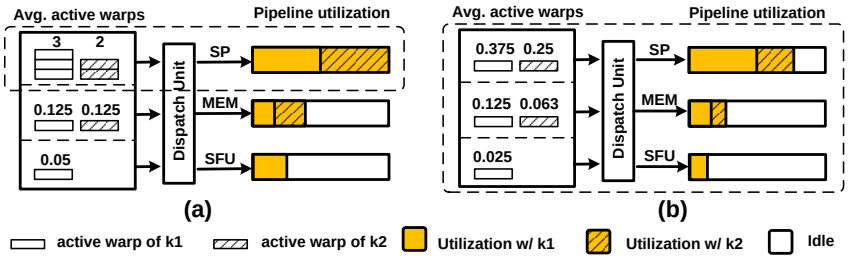

Figure 9.4: Determining throughput for mixed kernels: (a) Pipeline constrained scenario; (b)Parallelism constrained scenario.

execution from an individual kernel's perspective. The kernel consists of inactive warps with control hazards, inactive warps with data hazards, and the rest are active warps. In our presentation below, we

This model integrates two sub-models:

- The IPC model, described in subsection 9.3.3, estimates the steady state IPC for the two kernels as a function of the number of active warps of each kernel.
- The Active Warps model, described in subsection 9.3.4, derives $N_{active,k1}$ and $N_{active,k2}$ as a function of IPC_k.

The two sub-models together form a simple nonlinear equation that does not admit a closed-form solution but can be solved iteratively in a few iterations using a standard root-finding method.

9.3.3 IPC Model

We now develop an analytical model that calculates IPC_{k1}, IPC_{k2} from the number of active warps of each kernel. To explain how performance is determined with multiple active warps in the scheduler, we consider two scenarios, shown in Figure 9.4. For both scenarios, the figure shows two kernels being co-issued to an SM by a single dispatch unit[1] . The numbers within the scheduler box indicate the average number of active warps in the steady state for each of the three pipelines. We use $P_{k,p}$ to denote the

[1] Recall that an SM has two dispatch units, each of which can feed an SP pipeline (exclusively) and the MEM and SFU pipelines (on a shared basis). The figure shows these three pipelines (SP, SFU, MEM) being fed by a single dispatch unit.

probability that an instruction will be issued to pipeline p from kernel k. This quantity reflects the degree of balance in the way kernel k utilizes the available pipelines, and can be obtained by profiling the kernel and its instruction mix. Individual pipeline utilization is shown to the right of the dispatch unit, with a solid bar representing utilization of kernel $k1$, a hashed bar representing utilization of kernel $k2$, and an empty bar representing idleness in the pipeline.

Determining Shader IPC

When multiple kernels are running concurrently in an SM, as shown in Equation 9.3, the mixed kernels are in one of two scenarios.

Parallelism constrained scenario: When there are not enough warps to keep any of the pipelines fully utilized, the SM suffers extra stalls, as illustrated in Figure 9.4(b). Many factors can lead to insufficient active warps, including low occupancy due to large resource requirements such as registers, shared memory, or grid size, or frequent thread block synchronizations that lead to invalid warps. In general, we say the SM is parallelism constrained, as the SM cannot provide sufficient active warps to keep at least one of the pipelines fully utilized. In the steady state, an equilibrium is reached where a warp is issued as soon as it turns active. Hence, for $\forall p \in \{SP, MEM, SFU\}$, we have:

$$IPC_{k,p} = N_{active,k,p}, \forall k \in \{k1, k2\}. \tag{9.4}$$

From the definition of $P_{k,p}$, it can be assumed that $N_{active,k,p} = N_{active,k} \times P_{k,p}$, where $N_{active,k}$ is the number of active warps in kernel k. Considering that none of the pipelines are fully utilized, the following condition must be satisfied for $\forall p \in \{SP, MEM, SFU\}$:

$$N_{active,k1} \times P_{k1,p} \times C_{k1,p} + N_{active,k2} \times P_{k2,p} \times C_{k2,p} < 1 \tag{9.5}$$

Pipeline constrained scenario: In this scenario, the SM has sufficient active warps ready to issue from the scheduler; however, one of the pipelines is fully utilized and becomes the performance bottleneck of the SM. Figure 9.4(a) illustrates a fully utilized SP pipeline.

$$IPC_{k1,p} \times C_{k1,p} + IPC_{k2,p} \times C_{k2,p} = U_{max} = 1 \tag{9.6}$$

When the pipeline p is fully utilized, not all the active warps can be issued immediately. Thus $IPC_{k,p}$ should be smaller than $N_{active,k,p}$, combining with Equation 9.6, the left side of Equation 9.5 must be greater than one when the shader is in the pipeline constrained scenario. Therefore, Equation 9.5 can be used as the boundary between the two scenarios.

When pipeline constrained, the warp scheduling policy determines how the pipeline utilization breaks down among two kernels in steady state. In this chapter, we assume the scheduler uses a loosely round-robin scheduling policy. As each active warp is equally likely to be issued in a round-robin warp scheduler, it is reasonable to assume that when pipeline p is fully utilized, the IPC of kernel k with respect to pipeline p ($IPC_{k,p}$) is proportional to the number of active warps of the kernels ($N_{active,k,p}$). Furthermore, as only p is fully utilized, Equation 9.4 also works for the rest of the pipelines. Therefore, we have:

$$\frac{IPC_{k1,p}}{IPC_{k2,p}} = \frac{N_{active,k1,p}}{N_{active,k2,p}} = \frac{N_{active,k1} - IPC_{k1} \times (1 - P_{k1,p})}{N_{active,k2} - IPC_{k2} \times (1 - P_{k2,p})} \tag{9.7}$$

From the assumption that kernel behavior is stable over its execution, a kernel that reaches the steady state must either be limited by parallelism constraints or pipeline constraints; one of the constraints is always dominant over the other when determining the performance of the shader. This is also verified through our experiments: each benchmark suffers primarily from either single pipeline congestion or from insufficient warps, with the effect of other factors such as branch divergence being less than 5% for GPU applications.

Therefore, with Equation 9.5 serving as the boundary condition of the two scenarios, from Equation 9.4, 9.6 and 9.7, the performance of individual kernel IPC_k can be derived as a *piecewise linear* function of $N_{active,k}$.

9.3.4 Active Warps Model

As shown in Figure 9.3, the Active Warps model estimates the number of inactive warps in the steady state based on IPC_k and certain kernel-dependent information. $N_{inact\_control,k}$ and $N_{inact\_data,k}$ of kernel k reflect the impacts of control hazards and data hazards during execution, both of which are determined by the behavior of its own instruction sequence of the kernel, and thus are independent from the other kernels

running simultaneously. Therefore, we can calculate $N_{active,k}$ considering only its own IPC_k and the kernel characterization parameters. Note with N_k warps initially assigned to each SM from kernel k, $N_{active,k}$ can be calculated as follows.

$$N_{active,k} = N_k - N_{inact\_control,k}(IPC_k) - N_{inact\_data,k}(IPC_k). \tag{9.8}$$

The following two sections will consider control hazards and data hazards, respectively.

Control Hazards

Inactive warps due to control hazards are mainly caused by branch prediction and grid synchronization, both of which are kernel-dependent characteristics under a round-robin scheduler. In the steady state, the fraction of inactive warps with control hazards typically remains stable. Thus, for a kernel k, $N_{inact\_control,k}$ can be written as $N_k \times P_{inact\_control,k}$, where $P_{inact\_control,k}$ is the probability of a warp turning inactive due to control hazards and N_k is the total number of warps assigned from the kernel. This probability is obtained through profiling.

Data Hazards

Data hazards occur when instructions which exhibit data dependence modify data in different stages of the pipelines. As an in-order SIMT processor, CUDA prevents data hazards by making a warp inactive when there is a read-after-write (RAW) or write-after-write (WAW) dependence between the next instruction of the warp and previous instructions of the warp which still reside in the pipelines. The warp returns to the active pool and resumes when there is no longer such data-dependencies in the pipelines. Given the scheduler modification with three individual dispatch units proposed in subsection 8.5.1, we further break down active warps into three categories based on the pipelines next instruction will be issued to (SP, SFU, MEM). For each pipeline, the behavior of warps that turn inactive due to data hazards can be modeled as an independent queuing system.

Figure 9.5 shows the schematic of the queuing system. We say that an inactive warp arrives to the system when the warp turns inactive due to a data hazard. Each inactive warp remains in the system until the data hazard is resolved and the warp

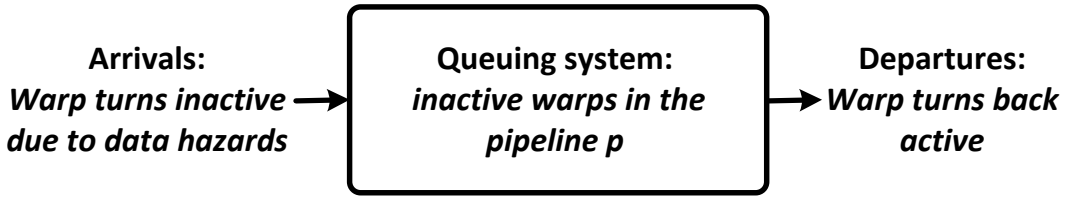

Figure 9.5: Flow of inactive warp through a queuing system

becomes active. According to Little's Law [75], in the steady state, the average number of items in a queuing system equals the average rate at which items arrive multiplied by the average time that an item spends in the system. Considering each pipeline as an independent queuing system,

$$L = \lambda W \tag{9.9}$$

where L is the average number of inactive warps in the queuing system, W is the average waiting time before a warp returns to the active pool, and λ is the average number of inactive warps arriving per cycle.

To solve the problem, we define the following two key concepts. Considering pipeline p of kernel k as the queuing system, *data hazards probability*, denoted as $P_{inact\_data,k,p}$, is the probability of one instruction of kernel k turning the warp inactive due to a data hazard in pipeline p. *Data hazard inactive time*, denoted as $T_{inact\_data,k,p}$, represents the average waiting time before the warp in kernel k becomes active again in pipeline p. Therefore, we have $\lambda = IPC_{k,p} \times P_{inact\_data,k,p}$ and W as $T_{inact\_data,k,p}$. From Equation 9.9, $N_{inact\_data,k,p}$ can be calculated as:

$$\begin{aligned} &N_{inact\_data,k,p} = \\ &IPC_{k,p} \times P_{inact\_data,k,p}(IPC_{k,p}) \times T_{inact\_data,k,p}(IPC_{k,p}). \end{aligned} \tag{9.10}$$

As denoted in Equation 9.10, both $P_{inact\_data,k,p}$ and $T_{inact\_data,k,p}$ are functions of $IPC_{k,p}$.

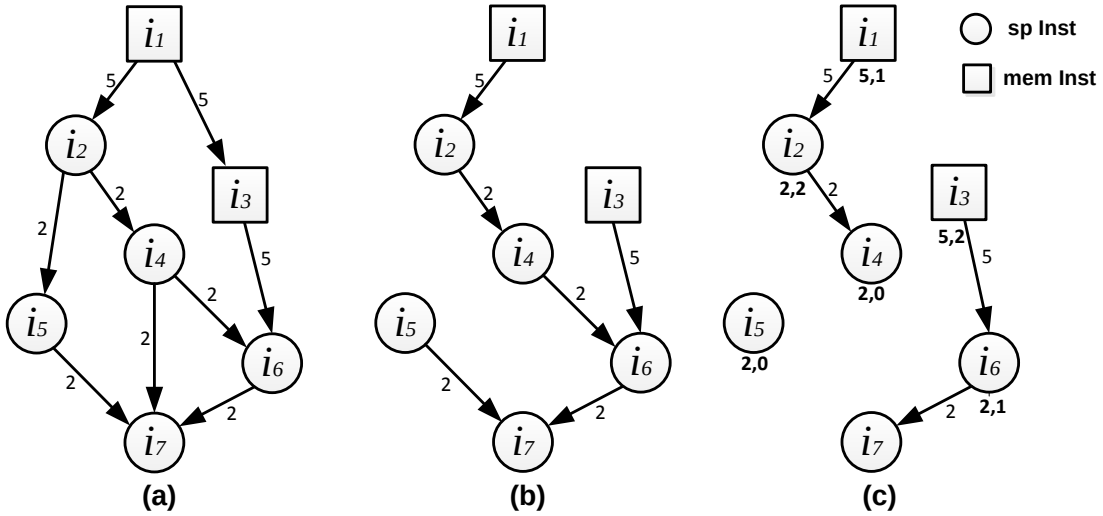

Figure 9.6: DAG Data Dependency Example: (a) Original DAG Graph; (b) DAG Graph After Forward Trace; (c) Critical Data Dependency DAG Graph

9.4 Expressing $E[P_{inactive\_data}(IPC)]$, $E[W(IPC)]$

In this section, we show how the two functions on the right-hand side of Equation 9.10, $P_{inactive\_data,k,p}$ and $W_{k,p}$ can be written as functions of $IPC_{k,p}$. For notational simplicity, we will drop the k subscript in this section. We begin by showing how data dependency is profiled in each kernel, and then show how this information is used to build the required models.

9.4.1 Profiling Data Dependency

For local data dependency analysis [76], based on run-time profiling where all loops are unrolled, instructions and their data dependencies are often represented using a directed acyclic graph (DAG) [77]. A graph *node* represents an instruction with a index number indicating instruction execution order. Each node has a type (SP, SFU, MEM) and a corresponding execution latency. A graph *edge* from node i to node j denoted as (i, j) represents a data dependency between node i and j with a weight equal to the execution latency of node i. This weight is the minimal number of cycles that must

elapse between the issue of i and the issue of j. For example, Figure 9.6(a) shows a DAG of 7 instructions. There are two types of instructions with different latencies. Nodes i_1 and i_3 have a latency of 5 and the remaining instructions have a latency of 2.

Not all the edges in a DAG will cause data hazards due to the nature of in-order nature of CUDA. As shown in Figure 9.6(a), ($i1$,i_3) will never cause a data hazard as transitivity implies that meeting (i_1, i_2) and the in-order precedence between i_2 and i_3 imply that (i_1, i_3) will always be met. Thus it is possible to refine the DAG with each edge indicating critical data dependency information that resulting a stall.

We define an edge (i, j) as *critical edge* if the warp turns inactive at node j due to data dependency between (i, j) when node i still resides in the pipelines and turns active right after node i exits the pipelines. A two-step algorithm is introduced to find all the critical edges of a DAG as follows, given a data dependency DAG,

1. *Forward trace*: for each edge (i, j), keep the edge if j is the first node in execution order that has a edge with i, otherwise, remove it.

2. *Backward trace*: for each remaining edge (i, j), keep the edge if node i is the last instruction that exits the pipelines before node j.

Figure 9.6(a) and Figure 9.6(b) are the DAG after forward trace and backward trace respectively.

The pruning operations due to forward trace are exact since all edges emanating from a node have equal weight. The implementation of the backward trace depends on the ability to identify the last instruction that exits the pipeline. We use an approximate heuristic here, using the highest numbered instruction for this purpose, unless one or more of the preceding instructions is a memory instruction, in which case we use the highest numbered memory instruction. Note that this approximation is acceptable since we are trying to build a performance predictor where some degree of inaccuracy is acceptable.

A critical edge not only determines if an instruction will turn the warp inactive, but also indicates how long this inactive period can be. Given a critical edge (i, j) we know that node i will cause the warp turn inactive if node i still resides in the pipeline when node j is about to be issued. The warp turns active again after node i exits the pipeline. This inactive period determined by the execution latency of node i and number

of instructions from the same warp between node i and j. Moreover, the latter has a impact on the inactive probability for a instruction: the more instructions between i and j, the better chance that i will finishes execution before j is about to be issued.

After the forward and backward traces, each node has at most one predecessor and at most one successor. For a node i_j, let us refer to its successor node, if any, as i_k. We define the *critical data dependency distance* for node i_j, denoted CD3(i_j), as the number of instructions between i_j and i_k, i.e., $|k - j|$. If i_j has no successor, its CD3 value is defined to be zero.

Applying this definition to Figure 9.6(c), CD3(i_1) = CD3(i_6) = 1, CD3(i_2) = 2, CD3(i_3) = 3, and CD3(i_4) = CD3(i_5) = CD3(i_7) = 0.

Calculating $P_{inactive\_data}$

Intuitively, CD3 captures the most critical data dependency associated with an instruction. If x consecutive instructions of the same warp reside within the pipelines, thus all the nodes with CD3 less than x will result in an inactive warp. The CD3 for all instructions can be obtained through profiling, and a *CD3 histogram* is used to represent the distribution of CD3 of the entire kernel. Note that different types of instructions usually have diverse data dependency characteristics hence are considered separately.

Figure 9.7(a) shows the CD3 histogram of SP and MEM instructions of kernel AES. The x-axis is the CD3 value and the y-axis is the probability of occurence, i.e., the fraction of total instructions that have this CD3 value, in AES. In other words, $hist_{p,x}$ represents the probability of an instruction from pipeline $p \in \{SP, SFU, MEM\}$ having a CD3 of x. For example, the first bar in the figure, i.e., $hist_{sp,1}$ is the probability that an instruction associated with SP has a CD3 value of 1, i.e., that the warp turns inactive immediately after a SP instruction being issued.

Therefore, if x is the total number of instructions issued from the same warp since $Inst$, before $Inst$ exits the last stage (write back stage) of the pipeline p, the probability of inactivity of instruction $Inst$ can be expressed as:

$$P_{inactive\_data,p}(x) = \sum_{j=1}^{x} hist_{p,j} \tag{9.11}$$

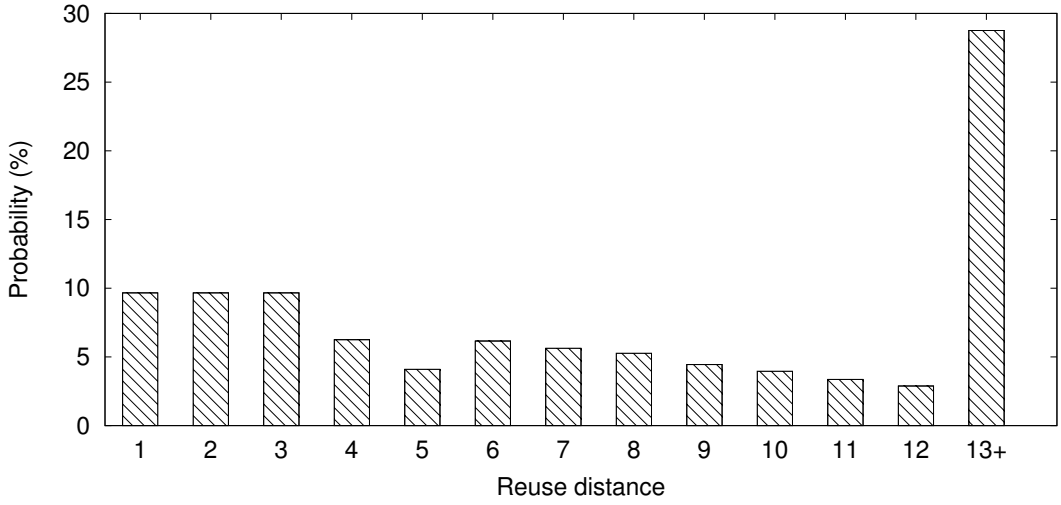

Figure 9.7: (a) CD3 histogram of AES; (b) $P_{inactive\_data}(x)$ of AES

This is simply the cumulative probability of CD3 being no larger than x. Next, we must find the expected value of the LHS of this expression, based on the distribution of x.

Figure 9.7(b) shows computed $P_{inactive\_data}(x)$ of SP and MEM instructions for kernel AES. Considering that the behavior of issuing instructions to pipeline p can be described as a Poisson process with an arrival rate of IPC_p, given N warps available in the scheduler with round robin scheduling, each warp is equally likely to be issued. The arrival rate of instructions to pipeline i from the same warp is IPC_p/N. If x is the number of instructions issued from the same warp to pipeline p, and the latency of the pipeline is $latency_p$ cycles, then x is modeled as a discrete stochastic variable x that has a Poisson distribution with parameter λ that is the product of the arrival rate and the time interval, i.e.,

$$\lambda = \frac{IPC_p \times latency_p}{N}. \tag{9.12}$$

The probability mass function of x is given by

$$f(k;\lambda) = Pr(x = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \tag{9.13}$$

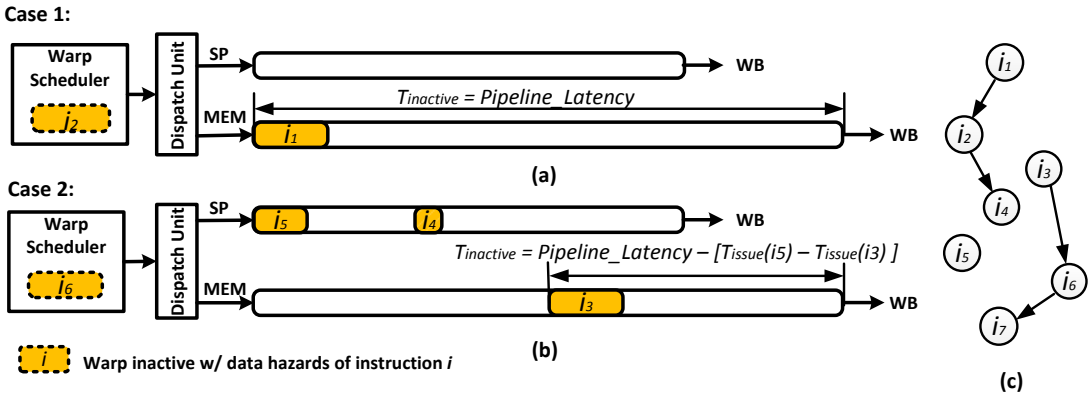

Figure 9.8: Calculateing $T_{inactive\_data}$

Therefore, the mean of $P_{inactive\_data,p}(x)$ can be derived as the following kernel-dependent nonlinear function of IPC_p:

$$E[P_{inactive\_data,p}(IPC_p)] = \sum_{i=1}^{N} f(i; \frac{IPC_p \times latency_p}{N}) \times P_{inactive\_data,p}(i). \quad (9.14)$$

From Equation 9.14, we know the more frequently the scheduler issues instructions from the same warp, more likely it is that a stall occurs. However, the issue rate of the same warp not only has an impact on how often one warp is inactive, but also affects how long the inactive period lasts.

Calculating W

When a stall occurs, the warp stays inactive until data hazard is resolved. To explain how data dependency affects W, we illustrate two scenarios in Figure 9.8. There are only two pipelines shown in the example and the the numbers inside the pipeline stages indicate instruction index number from the same warp. The right DAG in Figure 9.8 shows the CD3 graph of the kernel sample.

For case 1 in Figure 9.8(a), instruction i_1 has a CD3 value of 1. Thus the warp turns inactive right after i_1 is issued to the pipeline and will turn back active when i_1 exits the pipeline. As a result, W here equals to the execution latency of i_1.

For case 2 in Figure 9.8(b), there are three instructions, (i_3, i_4 and i_5) from the targeting warp executing in the pipeline. Due to the data dependency between i_3 and i_6, the warp turn inactive right after i_5 being issued, since i_6 may not be issued until i_3 exits the pipeline and the data dependency regarding i_6 is resolved. Therefore, W here equals to the time interval between i_5 being issued and i_3 exiting the pipeline. If $T(i)$ is the issue time of instruction i, we have $W = latency - (T(i_5) - T(i_3))$.

For both cases, if an instruction with a CD3 value of i causes the warp stall, W can be summarized as *latency* minus the issue time interval of $i-1$ instructions. If there are x instructions that currently reside in the pipelines, the average issue interval between each instruction from the same warp is considered as $\frac{latency}{x+1}$.

According to the CD3 histogram, considering x as the number of instructions from the same warp that reside in the pipelines, the probability of a warp turning inactive due to an instruction with CD3 value of i ($i \leq x$), denoted as $P[CD3 = i|x]$, is given by

$$P[CD3 = i|x] = \frac{hist_{p,i}}{\sum_{j=1}^{x} hist_{p,j}} = \frac{hist_{p,i}}{P_{inactive\_data,p(x)}} \tag{9.15}$$

Therefore, the average inactive waiting time of pipeline p, W_p can be expressed as a function of x,

$$\begin{aligned} W_p(x) &= latency_p - \sum_{i=1}^{x} P[CD3 = i|x] \times \left(\frac{latency_p}{x+1}\right)(i-1) \\ &= latency_p \left(1 - \frac{\sum_{i=1}^{x} hist_{p,i} \times \frac{i-1}{x+1}}{P_{inactive\_data,p}(x)}\right), \end{aligned} \tag{9.16}$$

where $latency_p$ is the average execution latency of pipeline p, and is considered a kernel-dependent constant.

Similar to Equation 9.14, we can write the mean of the LHS of Equation 9.16 as a function of IPC_p as follows,

$$E[W_p(IPC_p)] == \sum_{i=1}^{N} f(i; \frac{IPC_p \times latency_p}{N}) \times W_p(i). \tag{9.17}$$

Combining Equations 9.10, 9.12, 9.14, and 9.17, N_{active} of each kernel in stable state can be expressed as a nonlinear function of IPC_p .

9.4.2 Numerical Solution of the Complete Model

From section 9.3.3, IPC is a piecewise linear function of N_{active}. Combining the IPC model, which provides the left hand side of Equation 9.8, and the Active Warps model, which provides the right-hand side, we numerically solve this nonlinear equation N_{active}. Both N_{active} and N_{inact} are monotonic increasing functions of IPC by definition, and therefore, only one positive solution exists. Since the solution for IPC is unique, we can solve the performance model using a standard root-finding algorithm, such as the bisection method [78] or the false position method [79]. In practice, this is a simple numerical computation that typically converges in four or five iterations.

9.4.3 Limitations of the Analytical Model

Our analysis does not consider the impact that kernel mixing has on cache misses and memory access latency. The tendency of the model is to balance memory-bound and computation pipeline utilization. There is little chance that our model decides to pair two memory-bound kernels together, which would create extra memory contention. Branch divergence is not handled explicitly in our model; however, our model is aware of $P_{inact\_control}$, and mixing kernels effectively alleviates the impact of insufficient warps caused by frequent branch divergence.

9.5 Static Program Analysis

All the parameters that fed to performance model can be obtained either from an automated characterization or by running PTX or assembly level code analysis[]. In this chapter, we choose hardware based automated characterization aiming for fast online response.

Once a new kernel comes to the pool, before mixing it with any other kernels, we assign the kernel alone for a sampling period, while during the execution, hardware performance counters on the SM are triggered to measure the following 5 behaviors: number of instructions issued to each pipeline, number of cycles that each pipeline is occupied, number of invalid warps in warp scheduler in each cycle, execution latency and the input /output registers of each instruction.

Note that the first three can be achieved in existing hardware performance counters by performance sampling. The last two require modifications to existing performance counter. For our purposes, we aim to determine the average execution latency and data dependency information of the kernel, which requires only targeting be behavior of one warp instead of the whole SM. This can be implemented by adding a module to compare warp id prior the sampling each cycle. Assuming the kernel have uniform behavior among the warps, then we can obtain such parameters through some simple calculation. Once the sampling process is completed, we can switch back to normal execution, and the migration overheard is negligible given fast context switch capability of GPU.

9.6 Warp Scheduling against Pipeline Starvation

When an instruction is dispatched in MEM pipeline, the memory controller will submit the corresponding memory transactions to interconnection network. Once all transactions are submitted, the MEM pipeline will start dispatch the next instruction permitting there is any. However, if the memory instruction is non-coalesced, it might potentially take hundreds of cycles until all memory transactions are fully submitted. Ultimately, if it takes too long that there will be no active warp left for the rest of the pipelines as all active warps are stuck in MEM pipeline, we call this pipeline starvation. New warp scheduling policy can improve pipeline starvation by setting low issue priority to non-coalesced memory instructions. Heuristically, we adjust the priority so that the number of memory transactions from kernel k is proportional to $N_{active,k,MEM}$.

9.7 Experimental Methodology

We have modeled our proposed architectural enhancement using a cycle-accurate GPU simulator, GPGPU-Sim [52]. The evaluation benchmarks are selected from the CUDA SDK [59], Rodinia [80], and GPGPU-Sim benchmark suites [81]. We include results on 18 benchmarks with a wide variety of behaviors – compute-bound vs. memory-bound; sufficient warps vs. insufficient warps due to resource constraints; barrier synchronization vs. barrier-free; and branch divergence vs. branch divergence free. For benchmarks

that contain multiple kernels, we only evaluate the first kernel.

9.7.1 Performance and Throughput Metrics

Our fined grained kernel scheduling and partitioning mechanism is based on two metrics, the SM IPC and the *throughput ratio*.

We use speedup in execution time under the same workloads to represent the throughput improvement. The **throughput ratio** is the number of cycles when kernels run alone, divided by the number of cycles when they run concurrently under the same workload. To understand the metric, consider the case where two kernels are assigned together, where the workloads, in terms of number of instructions, are denoted as L_k, $k \in \{k1, k2\}$. If one workload is significantly more than the other, we should deliberately assign more grids from this kernel in order to further take advantage of kernel mixing. This will change the objective function that we try to optimize and is dependent on the workload ratio.

In this chapter, we do not consider the problem of predicting the number of instructions of each kernel. In order to focus on the effect of improvement in throughput, we assume all the workloads are running indefinitely or throughout the duration of the evaluation period. Under this assumption, the throughput ratio can be described as follows. If we consider kernels running concurrently with a certain partition, long after the SM reaches steady state, after cycle T, then

$$\text{Throughput Ratio} = \frac{\sum_{k\in\{k1,k2\}} L_k(T)/T}{IPC_{solo,k}}, \tag{9.18}$$

where $IPC_{solo,i}$ is the SM IPC of kernel k when running alone on the SM.

9.7.2 Simulation Framework

Our simulator is modified from GPGPU-Sim v3.0.1 and is configured to model a GPU similar to NVIDIA's GTX480. The warp scheduling module is configured as a dual-scheduler, and we increase the throughput to support up to 3 instructions per cycle as mentioned in subsection 8.5.1. The dual-schedulers operate alternately, leaving only one scheduler active each cycle. GPGPU-Sim configures the SM frequency to half to compensate 16 execution units with a warp size of 32. We keep the original clock but

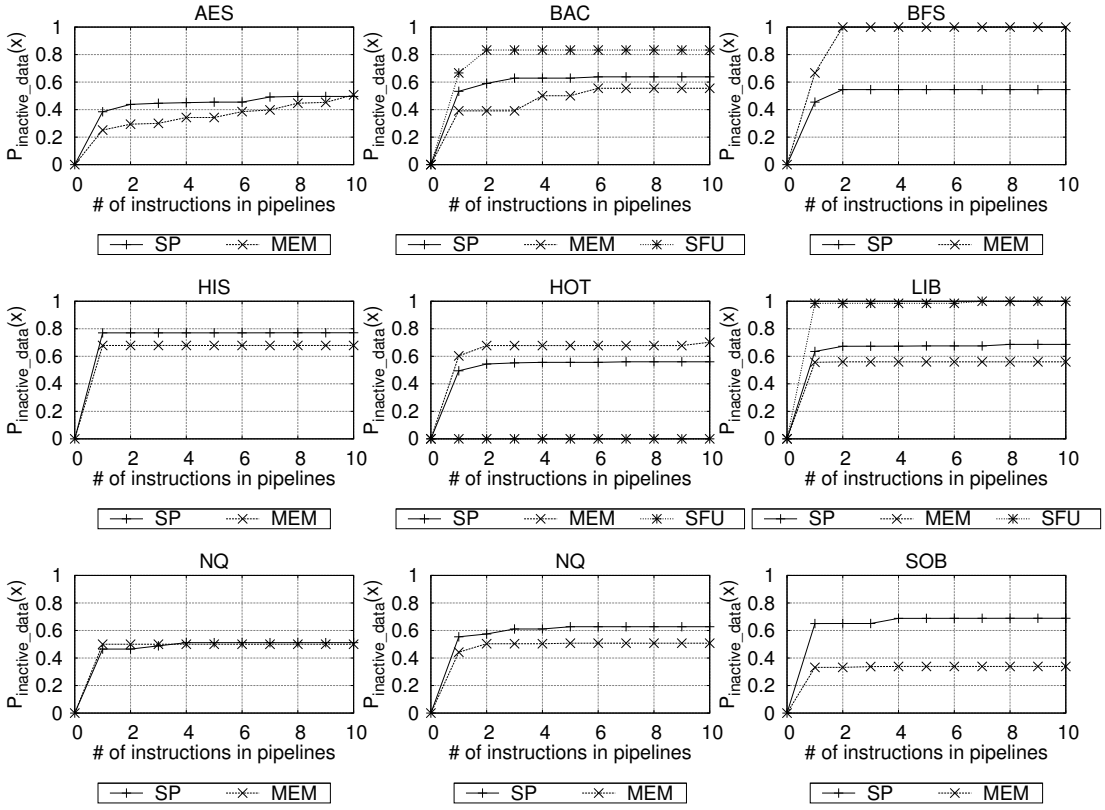

Figure 9.9: $P_{inactive\_data}(x)$ of the Benchmarks

modify the pipeline stages, throughput and latency based on the type of the instructions. The numbers come from the measurement on real hardware by [50]. As stated earlier, for simplicity, only two kernels are allowed to issued concurrently in our experiments. The simulator has the capability to co-schedule any two kernels with a specified grid partition within the resource limit. 147 (out of the 153 kernel mix pairs that are possible from pairwise combinations of 18 kernels) are simulated with at least 5 different grid partitions on each pair. The remaining 6 kernel pairs cannot be co-issued due to resource limitations. Each kernel pair with given grid partition is simulated for 200,000 cycles, and the kernel is reissued immediately if it finishes early. However, it is rare for a kernel to finish early: most of the kernels last longer than the simulation time.

The maximum thread blocks and warps allowed per SM is configured as 16 and 48.

Table 6.2 shows the major configuration parameters of GPGPU-Sim.

9.8 Results

9.8.1 Benchmark Characteristics

Automated characterization is described in section 9.5 by profiling certain hardware performance counters. We access the same information in GPGPU-Sim as if they are sampled from the performance counters, the sampling period being set to less than 2s for each kernel. Figure 9.9 illustrates the $P_{inactive\_data}$ of three pipelines corresponding to the number of instructions reside in the pipelines from the same warp. For some kernels, only two pipelines are shown as there is no SFU instructions from the kernels. As shown in Figure 9.9, for most of the kernels, different pipelines display diverse data dependencies. This further proves the correlation between the data dependency pattern and instruction type, which matches our assumption. It is worth pointing out that the rising rate in Figure 9.9 indicates the likelihood of a kernel suffering from resource constraints. If a curve increases rapidly and this type of instruction is heavily used by the kernel, e.g., the SP pipeline of SOB, this kernel is more likely to suffer from resource constraints.

9.8.2 Throughput Prediction in Mixed Kernel Scenario

Figure 9.10 shows the prediction accuracy of the model under kernel mixing. The y-axis demonstrates the throughput improvement of kernel BL when it is mixed with each of the other 17 kernels. Each pair picks the grid allocation from the performance model that provides the highest throughput improvement. Except MUM, the predictions of all the other benchmarks match well with the simulation results with an average of 5.7% error. The figure also shows an average throughput improvement of 23.4% when other kernels are mixed with BL.

For a specific pair of mixed kernels, AES and BL, Figure 9.11 shows the predicted vs. measured throughput improvement for different grid partitions. The x-axis indicates the number of grids issued from AES and BL respectively. All possible grid partitions ratios are evaluated to find the optimal throughput. Overall, there is significant throughput

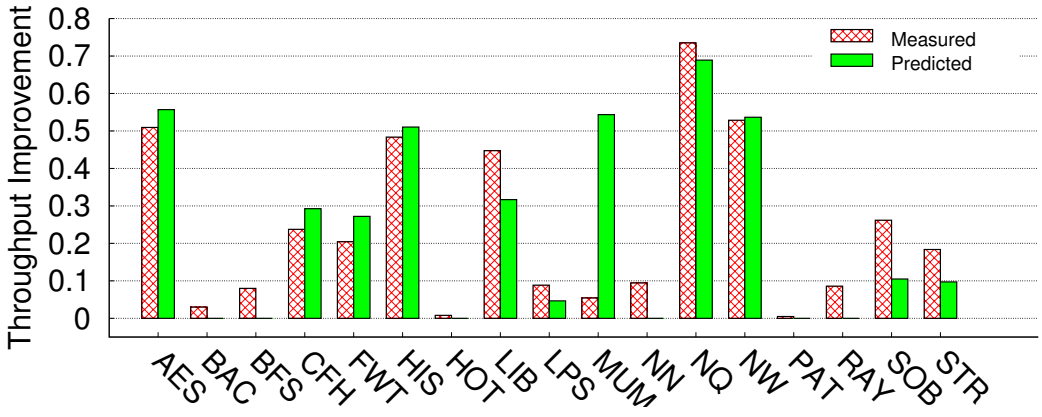

Figure 9.10: Throughput Improvement of BL Mixing with Other Benchmarks

improvement as AES is pipeline-constrained in MEM while BL is pipeline-constrained in SP. The mix of AES and BL allows a more balanced pipeline utilization between MEM and SP, and hence sees a significant throughput boost.

The prediction model failed to find the optimal grid allocation decision, corresponding to the optimal partition is (2,4) with 50.99% throughput improvement. However, the choice that the prediction model makes, (2,5), has a similar (46.15%) increase, corresponding to the second best partition. In fact, over evaluations of all 147 kernel pairs, our prediction model successfully predicts the optimal grid partition in 75 pairs, and for 95 pairs it is very close to the optimal partition (within 5% in terms of throughput improvement) while only 21 pairs go beyond 10%. Therefore, the performance model can accurately capture how throughput changes over a wide range of grid partitions, providing optimal or near optimal grid partition of each kernel mixes from a throughput perspective.

9.8.3 Execution Lane Starvation and Inter-warp Scheduling Results

Figure 9.13 shows the effect of inter-warp scheduling discussed in section 9.6 for pipeline starvation. MUM is the only one of the 18 kernels that we evaluated with $C_{mum,MEM}$ over 40, which is 10 times larger than the rest of the kernels. For each kernel mix, we heuristically set the issue priority in the MEM pipeline, as 0.1 to MUM and 0.9 to

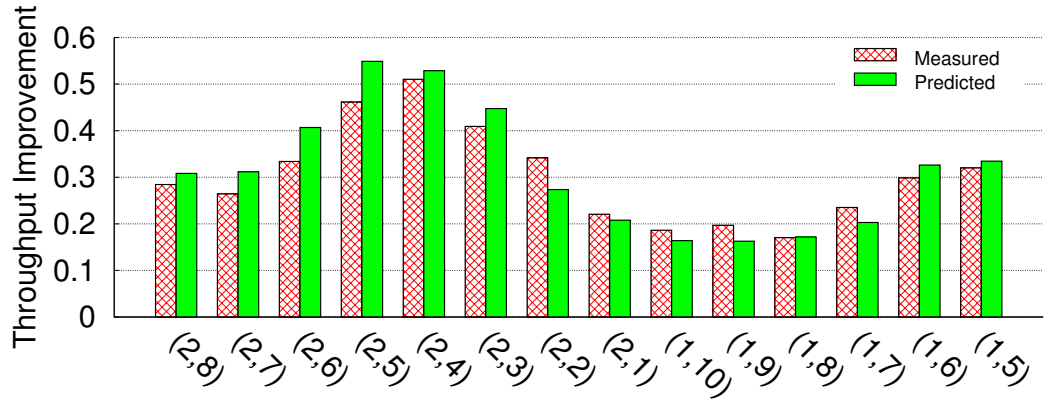

Figure 9.11: Throughput Improvement Different Kernel Partitions in (AES,BL) Pair

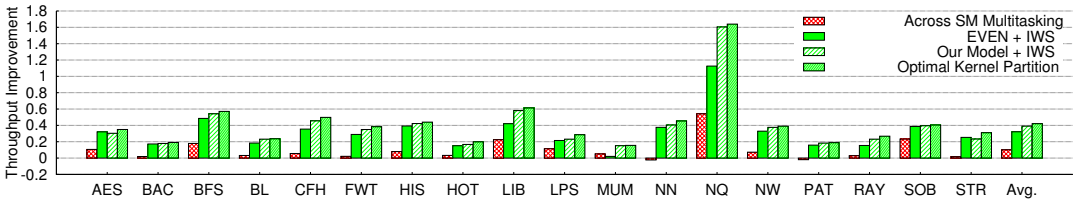

Figure 9.12: Average Throughput Improvement of Benchmarks

the other kernel. As a result, in every cycle in the scheduler, if there are $N_{active,MEM}$ from both MUM and the other kernel, there is 10% chance the scheduler will issue a instruction of kernel MUM to pipeline MEM. With inter-warp scheduling, the average throughput is improved from 26.14% loss to 15.13% increase.

9.8.4 Average Throughput Improvement

To our knowledge, there is no other technique that provides throughput-wise optimal grid partition. For comparison purposes, we considered and evaluated a heuristic partition technique that we call EVEN. EVEN aims to maximize the number of threads can issued on the SM, which potentially can alleviate the impact of resource constraints. In addition, EVEN tries to allocate threads evenly between two kernels that are mixed

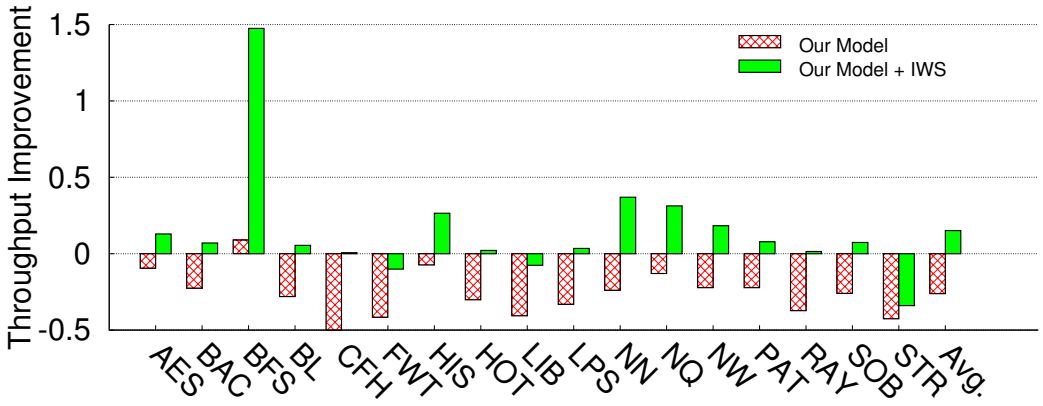

Figure 9.13: Effects of Inter-Warp Scheduling of MUM with Other Benchmarks

together, which in general, will improve the balance of the utilization in pipelines. Figure 9.12 presents the effects of our model and the EVEN model on finding the optimal grid partition. To sum up, mixing multiple kernels on the SM can achieve an average of 42.12% throughput improvement and our mean value based prediction model can accurately predict the near optimal kernel partition, averaging 39.15% in throughput improvement, better than the results of EVEN.

9.9 Related Work

9.9.1 Simultaneous Multitasking for GPGPUs

The GPU spatial multitasking technique proposed by Adriaens *et al.* [63] alleviates system bottlenecks and improves TLP by partitioning GPU cores among multiple applications with each core executing in the normal single-kernel fashion. This strategy does not address underutilization (e.g., low PLP) within GPU cores and still applies homogeneous simultaneous multithreading per core. Gregg *et al.* [67] and Guevara *et al.* [68] first demonstrate the throughput potential of intra-core kernel co-scheduling on real hardware via off-line kernel merging in software. Such software-based approaches are not applicable to all workloads and suffer high overhead. Pai *et al.* [64] implement concurrent kernel execution on the real hardware, by merging two instruction trace of

the kernels running alone. Due to in-order-issue feature of GPUs, merging two instruction trace serializes two kernels with pre-determined instruction order. Such merged trace cannot accurately reflect how two kernels interact given different CTA partitions. Lee *et al.* citelee14hpca also illustrate the benefit of simultaneous multitasking within SMs, but the detailed hardware implementation and CTA partition unclear. Our work is mainly focused on SIMT efficiency due to scheduler, resource and pipeline constraints, and is thus orthogonal to prior work and can be integrated with above approaches to further improve SIMT efficiency.

9.9.2 Performance Modeling of GPU

Our performance model is strongly related with our fine-grained kernel partition technique. However, there is no analytical model that can predict how two kernels reach an equilibrium, and how the individual kernel performance is impacted by the other kernel when sharing the SM pipelines. The closest work is by Hong *et al.*[82], who proposes an analytical performance model with memory bandwidth and thread-level parallelism awareness in the single kernel scenario. However, their work assumes a uniform data dependency, as a warp always stalls before the previous issued instruction of the warp exits the pipeline, which introduces significant inaccuracy under changing pipeline and resource constraints. Furthermore, their work cannot predict the performance breakdown in steady state and therefore cannot provide accurate kernel mix decision for our fine-grained kernel partition technique.

9.10 Conclusion

We have presented an approach for fine-grained kernel mixing, based on a new analytical performance model. The approach is demonstrated to provide large improvements in the throughput over existing methods as well as an intuitive kernel mixing heuristic. Its performance is further enhanced using our inter-warp scheduling algorithm, and the combined result of these methods show that our mean value based prediction model can accurately predict the near optimal kernel partition: it achieves an average of 39.15% in throughput improvement, compared to the optimal 42.12% throughput improvement possible via our exhausted kernel mixing test.

Chapter 10

Conclusions

In this dissertation, we have presented a comprehensive set of modeling and scheduling techniques for design-time validation and run-time monitoring and optimization for high performance computing systems such as CMPs and GPUs.

We have designed and evaluated a shared cache aware performance model named CAMP for CMPs in a multi-programmed environment. CAMP is capable of accurately and quickly predicting the effective cache sizes of cache-sharing processes on a CMP machine using last-level cache access related information. Thanks to the hardware performance counters that are built into most modern high-performance computers, CAMP does not require modifications to applications, operating system, or the underlying hardware. We also describe an automated way of gathering process-dependent information for using CAMP online. CAMP has been validated on multiple CMP machines with different architectures. The average performance prediction error is 3.38% across 36 different process combinations on a quad-core server and 1.57% across 55 different process combinations on a dual-core workstation, respectively.

We presented a system-level power model for processor power estimation during run-time in a multi-programmed CMP environment, account for core-wise time sharing and chip-wise cache contention. Similar to CAMP, the power model makes use of hardware performance counters, thus requiring no changes to the underlying hardware or software. We validated the power model on a dual-core workstation and a four-core server. Experimental results indicate the average error is 3.17% for the dual-core workstation across 60 different process-to-core mappings and 3.16% for the four-core

server across 37 different process-to-core mappings, respectively. We also explain how to integrate CAMP with the power model for power estimation during assignment. We validated the combined model on the four-core server. The average error is 2.38% across 83 different process-to-core mappings.

Both CAMP and the system-wide power model indicates the last-level cache miss rate is a good indicator of energy saving opportunities. Therefore, we proposed an off-chip memory access-aware runtime DVFS control technique for performance-constrained energy minimization problem. We first proposed an oracle algorithm to determine the best case energy savings achievable under a performance constraint, assuming a priori knowledge about application behavior. We then proposed a practical on-line predictive DVFS algorithm that is capable of generating close-to-optimal results without requiring a priori knowledge of application behavior. Both algorithms have been evaluated on a real system. When compared with the most advanced related work (F-DVFS), P-DVFS leads to energy consumptions within 1.83% of the optimal oracle solutions on average with a maximum deviation of 4.83%, whereas the F-DVFS results in energy consumptions within 9.80% of the optimal oracle solution on average with a maximum deviation of 29.86%. In addition, P-DVFS also reduces power consumption by 9.93% on average and up to 25.64% compared to F-DVFS.

Moving to the GPU side, after a thorough analysis on per-warp CPI breakdown, we laid out all the key factors that govern GPU throughput from a single warp perspective. In order to improve GPU throughput, we need to improve the degree of parallelism, reduce structural and data hazards, and improve stalls due to barrier and functions done.

We proposed and evaluated GTLS-TAWS, a new two-level priority scheduling scheme, which ranks CTAs based on the number of warps suffering stalls due to barrier and function done, then prioritize warps within CTAs in a greedy then least scheduled fashion. By keeping warps within the same CTA at similar pace, while different CTAs at different progress, GTLS-TAWS can effectively improve stalls due to barrier, function done, and structural hazards. Compared with baseline GTO scheduling policy, GTLS-TAWS reduces CPI due to barrier and function done by 47.15%, and achieves an average IPC speedup of 4.92%.

We proposed ICMT, a full, detailed solution for intra-core multitasking for GPGPUs, including architectural support and a contention-aware co-scheduling approach that improves TLP and PLP in a balanced fashion. We demonstrated 28.07% average performance benefits for ICMT with only 1.79% area overhead, compared to conventional single kernel execution.

Finally, to coupled with intra-core multitasking on GPGPUs, we proposed a new contention-aware analytical performance model for GPUs. The approach is demonstrated to provide large improvements in the throughput over existing methods as well as an intuitive kernel mixing heuristic. Its performance is further enhanced using our inter-warp scheduling algorithm, and the combined result of these methods show that our mean value based prediction model can accurately predict the near optimal kernel partition: it achieves an average of 39.15% in throughput improvement, compared to the optimal 42.12% throughput improvement possible via exhausted kernel mixing test.

References

[1] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Queue*, 8(1), January 2010.

[2] W Kim, M Gupta, G Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. *Proc. Int. Symp. High-Performance Computer Architecture*, January 2008.

[3] C Isci, A Buyuktosunoglu, C Chen, P Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. *Proc. Int. Symp. Microarchitecture*, pages 347 – 358, December 2006.

[4] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. Int. Symp. Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[5] S Zhuravlev, S Blagodurov, and A Fedorova. Addressing shared resource contention in multicore processors via scheduling. *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, March 2010.

[6] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, second edition, 2003.

[7] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software–Practice and Experience*, pages 705–736, 1996.

[8] Hiroshi Sasaki, Yoshimichi Ikeda, Masaaki Kondo, and Hiroshi Nakamura. An intra-task DVFS technique based on statistical analysis of hardware events. In *Proc. Int. Conf. Computing frontiers*, May 2007.

[9] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *proc. int. conf. parallel architectures and compilation techniques*, pages 25–38, September 2007.

[10] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *proc. int. conf. parallel architectures and compilation techniques*, pages 13–22, September 2006.

[11] T. Qiming, P. F. Sweeney, and E. Duesterwald. Understanding the cost of thread migration for multi-threaded Java applications running on a multicore platform. In *proc. int. conf. performance analysis of systems and software*, pages 123–132, April 2009.

[12] Xi E. Chen and Tor M. Aamodt. A first-order fine-grained multithreaded throughput model. In *proc. int. symp. high-performance computer architecture*, pages 329–340, March 2009.

[13] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *proc. int. symp. high-performance computer architecture*, pages 340–351, February 2005.

[14] Ravi Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *proc. annual international conference on supercomputing*, pages 257–266, June 2004.

[15] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *proc. int. conf. parallel architectures and compilation techniques*, pages 111–122, September 2004.

[16] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *proc. int. symp. high-performance computer architecture*, pages 117–128, February 2002.

[17] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *proc. int. symp. microarchitecture*, December 2006.

[18] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *proc. annual international conference on supercomputing*, pages 1–12, June 2001.

[19] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *proc. int. conf. parallel architectures and compilation techniques*, pages 221–231, October 1999.

[20] Tipp Moseley, Joshua L. Kihm, Daniel A. Connors, and Dirk Grunwald. Methods for modeling resource contention on simultaneous multithreading processors. In *proc. int. conf. computer design*, pages 373–380, October 2005.

[21] Joshua L. Kihm and Daniel A. Connors. Implementation of fine-grained cache monitoring for improved SMT scheduling. In *proc. int. conf. computer design*, pages 326–331, October 2004.

[22] Dinero IV trace-driven uniprocessor cache simulator. `http://www.cs.wisc.edu/~markhill/DineroIV`.

[23] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *proc. int. conf. parallel architectures and compilation techniques*, pages 339–352, September 2007.

[24] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *proc. int. conf. architectural support for programming languages and operating systems*, pages 121–132, March 2009.

[25] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-term workload phases: Duration predictions and applications to DVFS. *IEEE Micro*, (25):39–51, October 2005.

[26] Intel 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/products/processor/manuals/`.

[27] Aj Kleinosowski, John Flynn, Nancy Meares, and David J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Proc. Int. Wkshp. Workload Characterization*, pages 83–100, September 2000.

[28] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *proc. int. symp. computer architecture*, pages 83–94, June 2000.

[29] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *proc. int. symp. microarchitecture*, pages 78–88, December 2006.

[30] Gilberto Contreras and Margaret Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. In *proc. int. symp. low power electronics & design*, pages 221–226, August 2005.

[31] K. Singh, M. Bhadhauria, and S.A. McKee. Real time power estimation and thread scheduling via performance counters. *acm sigarch computer architecture news*, pages 46–55, May 2008.

[32] PAPI 3.6.2. `http://icl.cs.utk.edu/papi/`.

[33] Jui-Ming Chang and Massoud Pedram. Energy minimization using multiple supply voltages. *ieee trans. computer-aided design of integrated circuits and systems*, (4):436–443, December 1997.

[34] Yumin Zhang, Xiaobo S. Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *proc. design automation conf.*, pages 183–188, June 2002.

[35] G. Varatkar and R. Marculescu. Communication-aware task scheduling and voltage selection for total systems energy minimization. In *proc. int. conf. computer-aided design*, pages 510–517, November 2003.

[36] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *proc. int. symp. microarchitecture*, pages 359–370, November 2003.

[37] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *proc. int. symp. microarchitecture*, November 2005.

[38] Yongpan Liu, Huazhong Yang, R. P. Dick, H. Wang, and Li Shang. Thermal vs energy optimization for DVFS-enabled processors in embedded systems. In *proc. int. symp. quality of electronic design*, pages 204–209, January 2007.

[39] Kihwan Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. In *ieee trans. computer-aided design of integrated circuits and systems*, pages 18–28, December 2004.

[40] A. R. Chandrakasan and R. W. Brodersen. *Low Power Digital CMOS Design.* Kluwer Academic Publishers, MA, 1995.

[41] C. Poirier, R. McGowen, C. Bostak, and S. Naffziger. Power and temperature control on a 90 nm Itanium–family processor. In *proc. int. solid-state circuits conf.*, pages 304–305, February 2005.

[42] Prabhakant Sinha. The multiple-choice knapsack problem. *Operations Research*, 27(3), 1979.

[43] lpsolve 5.5. `http://lpsolve.sourceforge.net/5.5/`.

[44] David Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European J. of Operational Research*, pages 394–410, 1995.

[45] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, pages 28–35, July 2000.

[46] The Green500 List - June 2015.

[47] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. Int. Symp. Computer Architecture*, pages 451–460, 2010.

[48] E Lindholm, J Nickolls, S Oberman, and J Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Proc. Int. Symp. Microarchitecture*, 28:39–55, 2008.

[49] NVIDIA. The CUDA compiler driver NVCC.

[50] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. Int. Conf. Performance Analysis of Systems and Software*, pages 235 –246, March 2010.

[51] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. The Parboil technical report.

[52] GPGPU-Sim.

[53] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi.

[54] Shin-Ying Lee and Carole-Jean Wu. Caws: Criticality-aware warp scheduling for gpgpu workloads. PACT '14, 2014.

[55] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *Proc. Int. Symp. Microarchitecture*, pages 72–83, 2012.

[56] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU

performance. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 395–406, 2013.

[57] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proc. Int. Symp. Computer Architecture*, pages 332–343, 2013.

[58] O. Kayiran, A. Jog, M.T. Kandemir, and C.R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, pages 157–166, 2013.

[59] NVIDIA Corporation. NVIDIA CUDA SDK 4.0.

[60] A. Munshi. The OpenCL specification. 2011.

[61] NVIDIA. CUDA C programming guild.

[62] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110.

[63] J.T. Adriaens, K. Compton, Nam Sung Kim, and M.J. Schulte. The case for GPGPU spatial multitasking. In *Proc. Int. Symp. High-Performance Computer Architecture*, February 2012.

[64] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, 2013.

[65] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *Proc. Int. Symp. High-Performance Computer Architecture*, pages 260–271, Feb 2014.

[66] M. Awatramani, J. Zambreno, and D. Rover. Increasing gpu throughput using kernel interleaved thread block scheduling. In *Proc. Int. Conf. Computer Design*, pages 503–506, Oct 2013.

[67] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX.

[68] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling task parallelism in the cuda scheduler.

[69] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. Int. Symp. Computer Architecture*, pages 392–403, 1995.

[70] S.J. Eggers, J.S. Emer, H.M. Leby, J.L. Lo, R.L. Stamm, and D.M. Tullsen. Simultaneous multithreading: a platform for next-generation processors. volume 17, pages 12–19, 1997.

[71] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15:322–354, 1997.

[72] P. Michaud. Demystifying multicore throughput metrics. *Computer Architecture Letters*, 12(2):63–66, July 2013.

[73] Sheng Li, Jung-Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480, Dec 2009.

[74] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, 2013.

[75] John D. C. Little and Stephen C. Graves. *Little's Law*. Springer US, 2008.

[76] P. G. Emma and E. S. Davidson. Characterization of branch and data dependencies on programs for evaluating pipeline performance. *IEEE Trans. Comput.*, 36(7):859–875, July 1987.

[77] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., 1997.

[78] B. Bradie. *A Friendly Introduction to Numerical Analysis.* Pearson Prentice Hall, 2006.

[79] Michael T. Heath. *Scientific Computing: An Introductory Survey.* McGraw-Hill Higher Education, 2nd edition, 1996.

[80] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proc. Int. Symp. Workload Characterization*, pages 1–11, 2010.

[81] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. Int. Conf. Performance Analysis of Systems and Software*, 2009.

[82] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. Int. Symp. Computer Architecture*, 2009.