# Performance, Power, and Thermal Modeling and Optimization for High-Performance Computer Systems

by

Xi Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

# ACKNOWLEDGEMENTS

I would like to express my gratitude towards all the people who have helped and inspired me during my Ph.D. study.

First, I am heartily thankful to my advisor, Professor Robert Dick, for his guidance, encouragement, and support during my research and study. We closely collaborated on all the work presented in the body of this dissertation. His perpetual energy and enthusiasm in research was always contagious and motivational for me, especially during the tough times in my Ph.D. pursuit. Despite his tight schedule, Professor Dick is always accessible and willing to provide his insights and share his wisdom when I am stuck at research. Without his patience and suggestions, this thesis would not have been completed.

I am sincerely grateful to Professor Li Shang, from the University of Colorado at Boulder, for his insightful comments and suggestions on the dynamic thermal analysis work described in Chapter 5, and hardware-based cache compression work described in Chapter 6. I especially would like to thank him for taking the time to verify the correctness of the proof in Chapter 5 and discuss its applications with me.

I am deeply thankful to my dissertation committee, Professor Thomas F. Wenisch, Professor Satish Narayanasamy, and Professor Heath Hofmann, for spending their time reviewing my dissertation and providing stimulating suggestions to improve my work. In particular, I am grateful to Professor Wenisch and Professor Narayanasamy for their comments on the performance maximization work described in Chapter 8, and Professor Hofmann for generously spending his time discussing the stability issues in the thermal analysis work in Chapter 5.

I would like to thank Dr. Christopher Sadler and Dr. Xiaobo Fan for their guidance and suggestions during my internship, from which I realized how modeling and optimization techniques can be applied in real-world scenarios. Such experience broadened my perspective on the practical aspects in the industry and directly motivated the power-constrained performance maximization technique described in Chapter 8. In addition, their invaluable insights helped me to define the problem in Chapter 8.

I was fortunate to have the opportunity to collaborate and disucss research with many brilliant researchers, including Lan Bai, David Bild, Lide Zhang, Yun Xiang, Xuejing He, Yue Liu, Phil Knag, Chi Xu, Sausan Yazji, Lei Yang, Alex Shye, Haris Lekatsas, Professor Peter Scheuermann, and Professor Zhuoqing Morley Mao. I would also like to extend my gratitude to my friends at Northwestern University and University of Michigan for their help and support, which makes my life joyful and exciting.

Finally, I am forever indebted to my parents for their constant love and support throughout my life. They deserve far more credit than I can ever give them.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure**

# LIST OF TABLES

# ABSTRACT

This dissertation presents several models for performance, power, and thermal estimations in high-performance computer systems. In addition, it also describes a hardware-oriented cache compression algorithm, a software-based online dynamic voltage and frequency scaling (DVFS) algorithm, and a software-based performance maximization technique in a power-constrained CMP environment, all of which are motivated by the observations obtained when developing the aforementioned models.

After summarizing the impact of architectural evolutions on various aspects of computer modeling, we present three models that estimate the performance, power, and temperature in such systems. The first model, CAMP, is a fast and accurate cache aware performance model for chip multiprocessors (CMPs) that estimates the performance degradation due to cache contention of processes running on cache-sharing cores. We then propose a system-level power model in a multi-programmed CMP environment that accounts for cache contention and explain how to integrate the two models for power estimation during process assignment, helpful for power-aware assignment. We also describe an IC thermal model and analyze the performance and accuracies of a variety of time-domain dynamic thermal analysis techniques that build upon the aforementioned thermal model, which motivates our new thermal analysis technique that significantly improves performance while maintaining similar accuracy.

When developing the performance model and the power model, we realized that memory hierarchy is of critical importance to system performance and energy consumption. This observation inspires the design and implementation of a high-performance microprocessor cache compression algorithm to expand effective on-chip last-level cache size and improve cache performance. It also leads to a predictive dynamic voltage and frequency control (DVFS) algorithm that takes advantage of the performance model and the power model for on-line minimization of energy consumption under a performance constraint without requiring a priori knowledge of an application's behavior. Finally, we propose PerfMax, a performance optimization technique that considers both process assignment and local power state control in a power-constraint environment for multi-chip CMPs with chip-wide DVFS based on accurate performance and power models.

CHAPTER 1

# Introduction

High-performance computer systems are commonly seen. Typical high-performance computer systems include stationary desktop computers, workstations, and servers. Unlike battery-powered handheld devices and smartphones, high-performance computer systems are equipped with significant processing power and abundant on-chip and off-chip memory, thus making them the ideal platform for processing resource (e.g., CPU and memory) intensive tasks. In addition, they usually have less stringent power and thermal constraint than embedded systems due to external power supplies and better cooling conditions.

## 1.1 Modeling High-Performance Computers

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 guardbanding 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. With the on-going move from uniprocessors to chip-multiprocessors (CMPs), we still want to use the models as we are integrating more and more cores on chip. 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 trade-offs 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. Runtime 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 CMP architecture 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 Computers

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 computers (see Chapter 6, 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 IMPORTANT SYSTEM METRICS

For high-performance systems, performance has always been a critical design metric because most consumers are willing to pay extra money for improved performance. Early work on performance modeling includes queueing network theory based analytical models [24, 27] and trace-driven simulation models [88]. Such models are general-purpose performance models that aim at modeling the entire computer. However, with new computer architectures including superscalar processors and multi-threading processors becoming popular in the 1990s and early 2000s, general-purpose performance models were no longer feasible due to the complicated interactions among different system components. Hence, new component-specific performance models have been developed [112, 86]. However, as continued technology scaling leads to higher and higher power density, industry has shifted their focus from high-performance single-core chips to CMP chips to sustain continued performance improvement. This major architectural change once again calls for new performance models.

Historically, power consumption only matters to battery-powered portable systems such as laptops and cellphones. The exponential increase in power consumption due to process scaling brings critical challenges to various aspects of high-performance microprocessor design. Although the current transition to CMP systems temporarily alleviates the growth rate of power consumption, it is likely that power will remain as a first-class design constraint due to non-stopping desire for higher performance. This implies the need for a fast and accurate power model to justify power-related design decisions. In addition, resources such as last-level cache are usually shared among multiple cores on the same chip in a CMP processor, implying new complications to power modeling techniques. Without a properly designed power model for CMP systems, many severe power-related problems will remain unresolved.

Temperature has a huge impact on IC performance and power. Increased IC temperature can lead to increased gate delays, increased subthreshold leakage power consumption, accelerated wear, and even functional failures. Thermal issues have been largely ignored in the past. However, As power density has increased exponentially with each new technology generation, managing on-chip temperatures have become a major obstacle to continued scaling of VLSI systems. Researchers have shown a growing interest in addressing thermal problems at the architecture level and the system level. Therefore, a configurable thermal simulator that can quickly and accurately generate both steady-state and dynamic chip thermal profiles is highly desirable.

## 1.4 DISSERTATION OVERVIEW

The rest of this dissertation is organized as follows. In Chapter 2, we first give an overview of the parameters that influence performance, power, and temperature. We then describe the performance, power, and thermal implications of architectural evolutions, especially the shift from single-core to CMP systems.

Chapter 3–Chapter 5 present our models for performance, power, and thermal estimation for high-performance computers. Chapter 3 describes a shared cache aware performance model for CMPs [94]. 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 [99]. Chapter 5 points out the problems with various time-domain dynamic thermal analysis algorithms during IC thermal simulation. We then present a novel dynamic thermal analysis algorithm [95]. For each of these models, we present experimental data to demonstrate that these models are fast and accurate.

Chapter 6–Chapter 8 describe our optimization techniques motivated by the aforementioned models. Chapter 6 presents a microprocessor cache compression algorithm to increase the effective capacity of the last-level on-chip cache with little hardware overhead [98, 97]. Chapter 7 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 [96]. Chapter 8 presents a power-constrained performance maximization technique that optimizes performance across the boundary of process assignment and local power state control for multi-chip CMPs with chip-wide DVFS. All three techniques are inspired by the findings obtained during the modeling process. We summarizes the contributions of the work presented in this dissertation in Chapter 9.

# CHAPTER 2

# Background

Performance, power, and thermal 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 model, power model, and thermal models 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 this dissertation.

The rest of this chapter is organized as follows. Section 2.1 gives an overview of the fundamental parameters that influences performance, power, and temperature, many of which must be accounted for in the models to generate accurate estimations. Section 2.2 briefly summarizes the impact of architectural evolutions, especially the changes in memory hierarchy, on performance and power modeling. Section 2.3 introduces thermal analysis and new requirements due to constantly evolving computer architectures.

## 2.1 Overview

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 [83]. 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 [53, 87] and clock modulation [31].

The chip thermal profile, i.e., the spatial temperature distribution on a chip, is governed by the heat equation. In order to prevent the chip from overheating, there exists an efficient heat transfer path from the chip to the ambient environment. A typical heat transfer path includes the active layer, the bulk silicon, a heat spreader, the thermal interface material, a heat sink in a forced-air ambient environment, and the packaging material. Intuitively, the chip temperature depends on the geometry and thermal properties such as thermal conductance and heat capacitance of the components on the heat transfer path. In addition, the chip power profile, i.e., the spatial power distribution on a chip, also has a significant impact on the chip thermal profile.

## 2.2 Performance and Power Modeling

Performance modeling and power modeling in high-performance computers have received significant research attention in the past. However, most prior work focus on modeling single-core (SC) systems or a specific system component [20, 58, 86, 112]. With the increase in VLSI integration density and on-chip power density, CMP architecture is becoming more and more popular in high-performance computers. Such transition necessitates the redesign of performance and power models due to the presence of new architectural features such as cache sharing. This is one of the fundamental problems that motivate the techniques presented in the following chapters of the dissertation, in particular the cache aware performance model and the system-level power model. In this section, we first give an overview of the memory hierarchy and explain how it affects system performance and power. We then describe the impact of computer architecture evolution, especially the memory hierarchy evolution, on performance and power modeling. Finally, we introduce hardware performance counters and its role in performance and power modeling.

### 2.2.1 Memory Hierarchy

The concept of memory hierarchy takes advantage of the principle of locality in computer systems: (1) temporal locality: recently accessed data will tend to be accessed again soon and (2) spatial locality: data adjacent to recently accessed data will tend to be accessed soon. A typical memory hierarchy consists of register files, caches, main memory, and I/O devices such as disks. Different levels in the memory hierarchy have different speeds and sizes, with register files being the fastest and smallest and I/O devices being the largest and slowest.

The memory hierarchy has a great impact on system performance and power. To explain this more clearly, we trace the events that occur during the execution of an instruction. When the new instruction is fetched, the CPU first determine whether the current instruction is a load instruction or a store instruction. If not, the CPU will serve the instruction by reading from the registers, executing it, and writing the results back into the registers. Otherwise, the CPU uses the memory management unit (MMU) to translate the virtual memory address to a physical address, which is then used to access the data in the cache. Data are fetched from cache to CPU immediately on a cache hit; on a cache miss, the requested memory contents must be read from the main memory, which may in turn lead to a page fault if the desired memory page does not reside in the main memory. Therefore, each level of the memory hierarchy can influence the number of CPU cycles to retire an instruction and hence, the system performance. In addition, the system power is inversely proportional to the amount of time spent on each instruction. Therefore, factors such as last-level cache miss rate also affect the system power. Since the memory hierarchy changes dramatically as the system architecture evolves from SC to CMP, it is necessary to examine the features of each architecture and their performance and power implications.

### 2.2.2 Impact of Computer Architecture Evolution on Modeling

The SC architecture is the dominant system architecture in early 2000s. As illustrated in Figure 2.1, an SC processor consists of a single CPU core running at a high clock frequency. In a single-issue, single-threading SC processor, only one hardware thread is active at any point of time. Therefore, processes running on a single-threaded SC system interact with each other through time multiplexing the resources such as CPU and memory, known as the

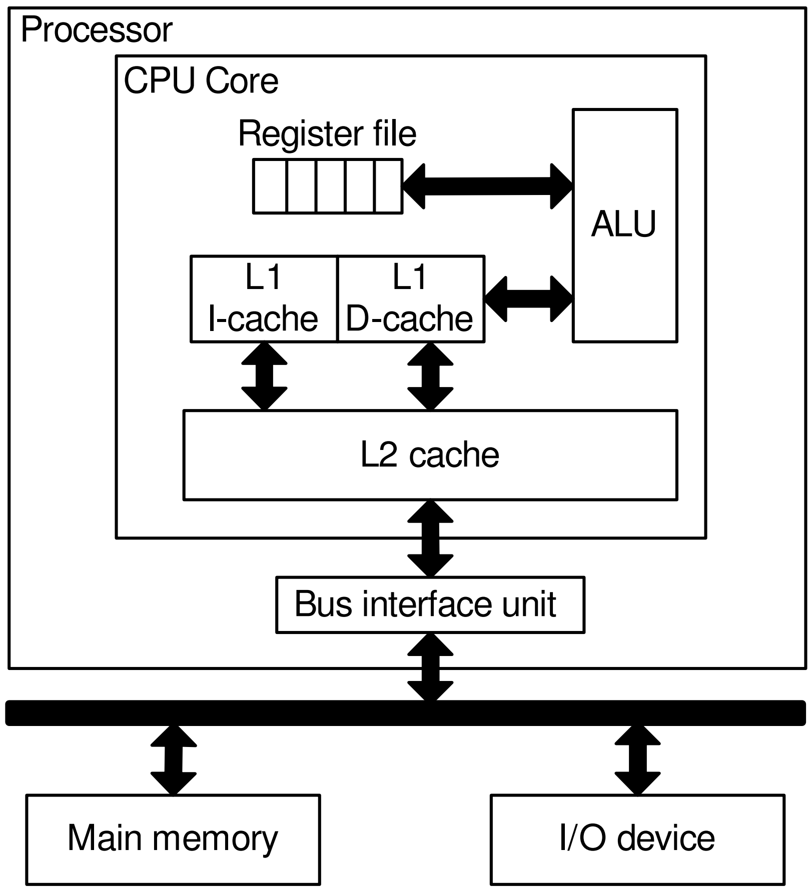


Figure 2.1: Basic structure of a single-core processor. The size of each level of the memory hierarchy does not reflect its size or capacity in a real processor.

*time sharing* problem. For example, a process can affect other processes' performance by either taking over the CPU, thus temporarily suspending other processes, or evicting cache lines or memory pages that belong to other processes, thereby causing more cache misses or page faults. However, such interactions are indirect since all resources are exclusively used by the current running process. This indicates the corresponding performance model and the power model should focus on the time sharing problem.

In order to boost SC system performance, researchers have proposed multiple-issue processors including superscalar processors and very long instruction word (VLIW) processors to exploit instruction-level parallelism and multi-threading processors including simultane-

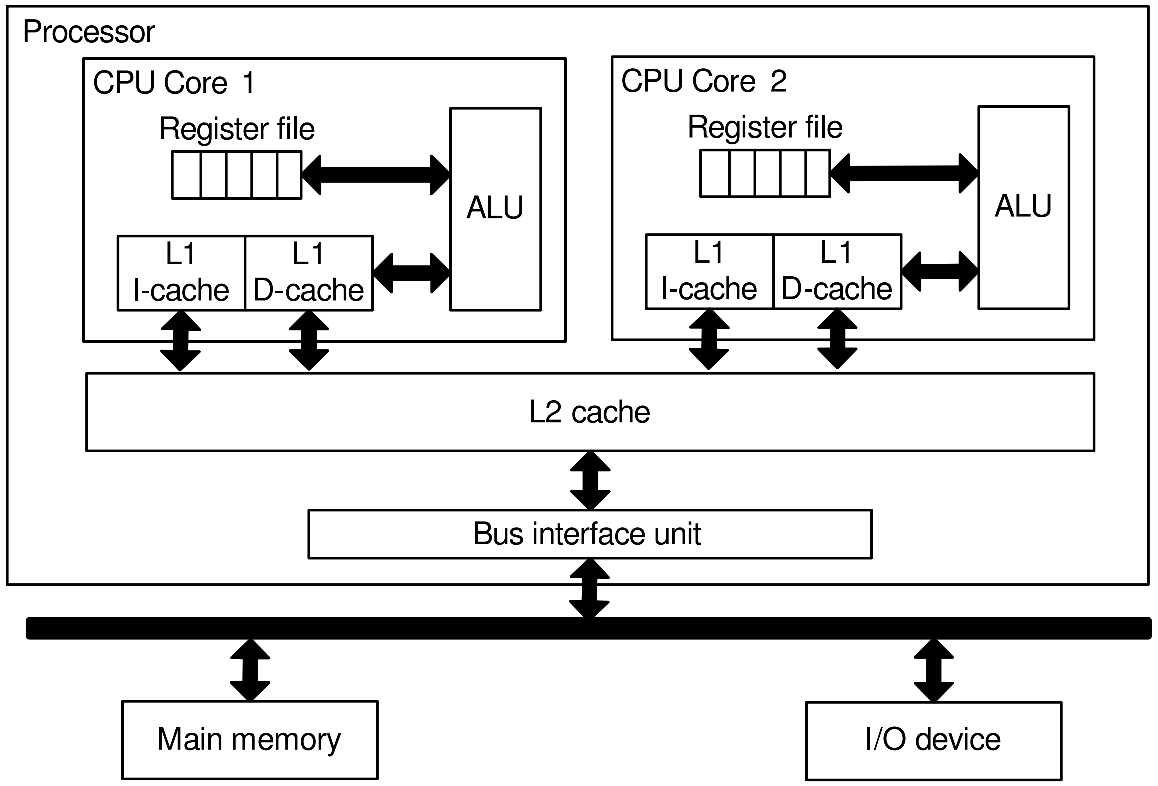


Figure 2.2: Basic structure of a two-core CMP. The size of each level of the memory hierarchy does not reflect its size or capacity in a real processor.

ous multithreading (SMT) processors to exploit thread-level parallelism. While multiple-issue processors can be modeled similarly to single-issue processors, multi-threading processors, especially SMT-enabled processors can be modeled similarly to CMPs, as multiple hardware threads may contend for the same resource simultaneously. Finally, researchers have also proposed multiprocessor systems in which a set of processing unit, each comprising a CPU and a private memory, cooperate by communicating through a bus or over an on-chip network. However, since no resource contention exists within each processing unit, performance models for such systems are inapplicable to shared cache based CMPs, which is the focus of our dissertation.

Figure 2.2 illustrates a basic architecture of a two-core CMP. Although only two cores are shown in Figure 2.2, there can be more than two cores per chip in real CMPs. Each

CMP processor is composed of two or more independent CPU cores, thereby allowing more parallelism than SC architecture. Each CPU core has its own private L1 caches, with the last-level cache (L2 cache in this case) 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 SC systems. It is worth mentioning that there is no concensus regarding the last-level cache structure; some CMPs have private last-level caches [16], i.e., each core has its own L2 cache. Shared cache and private cache have their own advantages and disadvantages [18, 82, 117]. This dissertation focuses on shared cache based CMP systems because it is more commonly found in modern high-performance commercial CMPs [49, 50] and (2) compared to on-chip caches, CMP systems usually have abundant off-chip memory. This makes the memory sharing problem a second order effect on system performance and power compared to cache effects, indicating private cache based systems may be modeled similarly to SC systems. We also note that there exist CMPs that consists of multiple chips, each of which in turn contains multiple CPU cores [52]. We focus on single-chip systems for a similar reason: the memory sharing problem posed by multi-chip systems is a second order effect compared to cache contention problem.

### 2.2.3 Hardware Performance Counters

Hardware performance counters (HPCs) are a set of special-purpose registers that are built into most of the modern high-performance computers to store the counts of various hardware events such as last-level cache misses and branch mispredictions. The number of available HPCs in a processor is quite limited (usually no more than 8) [3]. In addition, the set of hardware events that can be monitored by the HPCs can vary widely depending on the processor architecture. Nonetheless, HPCs provide a better alternative to software profilers for performance and power modeling because they provide detailed information about low-level hardware events related to system components such as integer unit, float point unit, and cache with very low performance overhead. In addition, it does not require changes to the underlying software or hardware, making it an ideal tool for building non-intrusive models, as shown in Chapter 3, Chapter 4, and Chapter 7.

## 2.3 Thermal Analysis

Thermal analysis for integrated circuits (ICs) is the process of building the IC thermal model and simulating the thermal conduction from an IC's power sources through cooling packages to the ambient environment. Thermal analysis can be separated into two subproblems: steady-state thermal analysis and dynamic thermal analysis. Steady-state thermal analysis determines the thermal profile (i.e., a temperature at each physical position) as time proceeds to infinity resulting from a power profile (i.e., a power consumption at each physical position). Dynamic thermal analysis determines the thermal profile as a function of time resulting from time-varying power profiles.

Thermal analysis has a long history. In the past, most of the prior work address thermal issues during cooling and packaging design using worst-case analysis. Although there exists a number of industrial tools widely used by packaging designers [38, 17, 32], it usually takes minutes or even hours to conduct each simulation with these tools because they are not designed for IC synthesis. It is not until very recently that researchers start to propose thermal analysis techniques that are suitable for IC designers. Traditionally, the dynamic thermal analysis problem can be solved using either frequency-domain or time-domain techniques. Time-domain methods rely on numerical integration. Therefore, with increasing time scale, their running times increase while their accuracies decrease, making them most suitable for short-time-scale thermal analysis. On the other hand, frequency-domain methods approximate the chip thermal profile using analytical solutions to avoid the problems of numerical integration. However, the one-time computational cost to derive the analytical solutions is very high. In addition, they ignore high-frequency components, thereby introducing errors to short-time-scale simulations. Hence, they are most suitable for long-time-scale thermal analysis.

With process scaling and increasing device density, thermal issues are becoming a major concern due to increased IC power densities and competing requirements of higher density, higher performance, and lower cost. It is envisioned that such thermal problems must not be addressed not only during packaging, but also during chip design and synthesis. However, as the design complexity of high-performance chips continues to grow, the thermal implications of various design decisions and optimization techniques are not intuitive. Nonetheless, understanding such thermal impacts is essential because transistor delay and leakage power are dependent on temperature. Therefore, an accurate thermal analysis algorithm is highly desirable. In addition, the impacts of changed IC thermal profiles on performance, power consumption, and reliability need to be analyzed repeatedly during the iterative IC design process. This calls for a fast thermal analysis technique to enable numerous detailed simulation runs of a IC thermal model consisting of a large number of thermal elements in the inner loop of a IC synthesis flow. Developing a fast and accurate thermal analysis algorithm is challenging. This is especially true for high-performance chips because they have complex architecture for performance and power purposes, which leads to complicated thermal interactions among different chip components and thus a longer simulation time.

In this dissertation, we focus on dynamic thermal analysis. Although more computationally intensive, dynamic thermal analysis is necessary when the power profile varies before the thermal profile converges and to detect transient violation of thermal constraints. In addition, we focus on time-domain techniques due to short simulation runs normally seen in IC synthesis. In particular, we experimented with Hotspot 4.0 [93] and ISAC [115] for dynamic thermal analysis on power profiles produced by behavioral synthesis algorithms during architectural optimization for high-performance ICs. To our surprise, we noticed that the step sizes of the time-domain methods used by Hotspot 4.0 converge to a constant value after a few iterations regardless of the initial power profile, thermal profile, and error threshold. On the other hand, the performance overhead of ISAC is quite high (it sometimes took quite more than a minute to conduct a 2 ms thermal simulation). The strange behavior of Hotspot 4.0 and the inefficiency of ISAC motivate us to explore the properties of various time-domain dynamic thermal analysis algorithms, which leads to a new temporally-adaptive technique that achieves significant speedup compared to Hotspot and ISAC, as shown in Chapter 5. It is worth mentioning that the proposed dynamic thermal analysis algorithm can handle both SC processors and CMP processors. In addition, it supports both two-dimensional (2D) and three-dimensional (3D) chips.

CHAPTER 3

# Performance Modeling

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 CMPs. CAMP estimates the performance degradation due to cache contention of processes running on CMPs. We have developed a fast and automated way to obtain process-dependent characteristics, such as reuse distance histograms, as input parameters to CAMP 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%. This work was done in collaboration with other researchers. In particular, Chi Xu was the leader on designing and evaluating the performance model. The author of this dissertation was responsible for automating the offline parameter extraction process and evaluating the performance model on one of the evaluation platforms.

This chapter gives a brief background introduction of the shared cache aware performance model for CMPs (CAMP), and focuses on the automated profiling process to characterize process memory access behavior to permit later prediction of cache contention and the potential sources of error during the automated profiling stage. Interested readers may refer to a previous publication [94] for a detailed discussion of the cache aware performance model. The rest of this chapter is organized as follows. Section 3.1 motivates the problem and summarizes our contributions. Section 3.2 describes related techniques. Section 3.3 provides a high-level overview of the assumptions and CAMP model itself. Section 3.4 provides an automated way to characterize process memory access behavior to permit later prediction of cache contention. It also discusses the potential sources of error during the automated profiling stage. Section 3.5 describes the experimental set-up and the workloads we evaluated and briefly summarizes the experimental 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 [37, 48, 79]. However, the problem of predicting the impact of cache sharing on application performance during process assignment has been considered by only a few researchers [29, 25]. 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 CMPs (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. All the measurements are performed on real processors.

## 3.2 Related Work

Past work [59, 36] has considered modeling transient performance penalties as a result of miss events such as branch mispredictions, instruction cache misses, and data cache misses for superscalar processors. However, our work intends to predict the performance impact of tentative assignment decisions on cache contention level in a CMP environment. Numerous researchers [57, 61, 101, 82] has considered the problem of adjusting cache partitioning during run time 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 [100, 39] 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.

Researchers have considered addressing the performance prediction problem using offline simulation [2] or modifications to the existing hardware or operating system [120]. For example, Suh et al. [101] proposed to add a hardware counter to each cache way and use them to determine the reuse distance histogram. Our goal in this work is runtime prediction of the performance of processes concurrently running on a shared-cache CMP, without requiring prior characterization.

There also exist numerous analytical performance models for cache contention aware performance prediction. Tam et al. [103] 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. However, They do not explain how to use miss rate curves to predict instruction throughput for processes sharing cache space. Their approach relies on performance counters peculiar to the POWER5 architecture. Chandra et al. [25] proposed using the reuse distances and/or circular sequence profiles for each thread to predict inter-thread cache contention. Chen et al. [29] 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. However, both techniques require input information that cannot be obtained without expensive simulation.

## 3.3 CAMP: Shared Cache Aware Performance Model for CMPs

In this section, we first formulate the performance modeling problem. We then give details on how to derive the non-linear equilibrium equations for effective cache size prediction.

### 3.3.1 Problem Formulation and Assumptions

The problem of performance prediction in the presence of cache contention can be formulated as follows: given $N$ processes assigned to cores sharing the same $N$-way set-associative last-level cache, predict the steady-state cache size occupied by each process during concurrent execution. Solving this problem is helpful for process assignment and migration in a CMP environment. However, accurate prediction of process performance is challenging due to the exponential number of possible process-to-core mappings.

In this paper, we consider a $N$-core processor with an L2 cache being the last-level on-chip cache. In the rest of paper, we refer to L2 cache simply as “cache” whenever this does not introduce ambiguity. We assume no hardware prefetching. Hardware prefetching complicates the model by predictively fetching cache lines based on access patterns. The model might therefore be inaccurate for systems using prefetching. However, we argue that the default prefetching mechanism is of limited value to the platforms and benchmarks we evaluated. For the 10 benchmarks tested on our target platforms, the average improvement was $3.25\%$, and only *equake* benefitted significantly. Note that our conclusion applies only to the particular computer systems and benchmarks for which data were gathered. For systems or benchmarks where prefetching plays a crucial role and thus should not be disabled, our performance model may lead to less accurate results. We also make the following assumptions: (1) the cache uses an LRU replacement policy and (2) processes are single-phased. In the case of multiple non-repeating phases with distinct memory access patterns, non-repeating phases should be modeled separately. Although these two assumptions simplify model design and explanation, we will later experimentally evaluate the proposed models when many of the assumptions are violated.

When multiple processes share a cache, contention may occur. We define the number of ways occupied by process $i$ in a set, denoted as $S_i$, as the *effective cache size* associated with process $i$. Therefore,

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

where $N$ is the total number of processes sharing the cache.

We define the *reuse distance*, $R_i$, of cache line $i$ as the number of distinct cache lines within the same set accessed between two consecutive accesses to line $i$. A *reuse distance histogram* represents the distribution of cache line reuse distances for an entire shared cache. For process $i$ with an effective cache size of $S$, all accesses to the cache lines with a reuse distance larger than $S$ result in cache misses. Hence, the probability of a cache access resulting in a miss for a process with an effective cache size of $S_i$, i.e., the misses per access (MPA), can be expressed as follows.

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

We also experimentally determined that SPI, number of seconds per instruction, can be expressed as a linear function of MPA,

$$\text{SPI} = \alpha \cdot \text{MPA} + \beta, \tag{3.3}$$

where $\alpha$ and $\beta$ are parameters that can be obtained during offline characterization. This observation is re-affirmed by Choi et al. [30].

### 3.3.2 Estimating Effective Size After $n$ Accesses

In this section, we use the reuse distance histogram of a process to derive its effective cache size. To simplify explanation, we will for the moment assume that the cache is initially empty. This assumption will later be relaxed. Given that $P_{i,n}$ is the probability of having an effective cache size of $i$ after $n$ consecutive cache accesses, the following recursive equation can be derived:

$$P_{i,n} = P_{i,n-1} \cdot (1 - \text{MPA}(i)) + P_{i-1,n-1} \cdot \text{MPA}(i-1), 1 < i \leq n. \tag{3.4}$$

This can be explained as follows. The fact that $n$ cache accesses result in an effective cache size of $i$ can only be the result of the following two scenarios: (1) the first $n-1$ cache accesses lead to an effective cache size of $i$ and the $n$th access results in a cache hit. The probability of this scenario, $P(A)$, is thus $P_{i,n-1} \cdot (1 - \text{MPA}(i))$; (2) the first $n-1$ cache accesses lead to an effective cache size of $i-1$ and the $n$th access causes a cache miss. The probability of this scenario, $P(B)$, is thus $P_{i-1,n-1} \cdot \text{MPA}(i-1)$. Since $P_{i,n} = P(A) + P(B)$, we can derive Equation 3.4. Note that $P_{1,1} = 1$ because the first cache access will always occupy a cache line. Assuming the process reaches its steady state after $n$ accesses, let $G(n)$ be a process' effective cache size, we have

$$G(n) = \sum_{i=1}^{n} (P_{i,n} \times i) \tag{3.5}$$

### 3.3.3 Equilibrium Condition

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 preserved in the cache. 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 $[t - T,\, t]$, which indicates Equation 3.4 and Equation 3.5 hold. Hence, the effective cache size of process $i$, denoted as $S_i$, can be written as $G_i(\text{APS}_i \cdot T)$. Conversely, $\text{APS}_i$ can be expressed as $G_i^{-1}(S_i)/T$. From Equation 3.3, we can derive an expression for $\text{APS}_i$:

$$\text{APS}_i = G_i^{-1}(S_i)/T = \text{API}_i/(\alpha_i \text{MPA}_i(S_i) + \beta_i). \tag{3.6}$$

Note that Equation 3.6 holds for any process $i$, where $i = 1, 2, \cdots, k$, given that $k$ is the total number of processes. Therefore, we have

$$\frac{G_1^{-1}(S_1)}{G_i^{-1}(S_i)} - \frac{\text{API}_1 \cdot (\alpha_i \text{MPA}_i(S_i) + \beta_i)}{\text{API}_i \cdot (\alpha_1 \text{MPA}_1(S_1) + \beta_1)} = 0, \forall_{i=1}^{N} \tag{3.7}$$

where $G^{-1}(S_i)$ and $\text{MPA}(S_i)$ are application-dependent nonlinear functions of $S_i$. Combined with Equation 3.1, we have $N$ equations that are independent of each other. Newton–Raphson iteration can therefore be used to solve for each $S_i, 1 \leq i \leq N$.

## 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

In this section, we describe how we characterize the inputs for our model using build-in hardware performance counters (HPC). We propose a new way to approximately capture the reuse distance histogram without using any offline simulation or additional hardware, instead, we calculate it by intentionally assigning a co-running process with multiple phases of different memory access behavior, based on the characteristics of the process we assign and the information collected from performance counter, we can calculate reuse distance histogram using the model above.

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 [3]. 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 $l$th 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 $l$th run, and considering the $l$th and the $l+1$st runs, we have

$$\begin{aligned} MPA_{B,l+1} &= \int_{S_{B,l+1}}^{\infty} \text{hist}_B(x)dx \text{ and} \\ MPA_{B,l} &= \int_{S_{B,l}}^{\infty} \text{hist}_B(x)dx. \end{aligned} \tag{3.8}$$

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

$$\text{hist}_B(S_{B,l}) \approx MPA_{B,l+1} - MPA_{B,l}. \tag{3.9}$$

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

**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: $S[Set \cdot Step \cdot \text{k}]$ 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
```

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 Algorithm 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 Algorithm 1, two consecutive reads are *Step* elements apart to ensure an 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 $\text{MPA}_{stress}$ 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 $l$th run, we set $k$ to $l$ for the stressmark in Algorithm 1. Since API is fixed for a process with the same input data, given that $API_l$ is the process's API in the $l$th run, the average API of the process can be estimated as

$$\text{API} = \frac{\sum_{l=1}^{A} API_l}{A}. \tag{3.10}$$

Similarly, we can get $A$ pairs of a process's 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 $l$th run, the $\alpha$ and $\beta$ in Equation 3.3 can be determined using linear regression, i.e.,

$$\alpha = \frac{A \cdot (\sum_{l=1}^{A} MPA_l \cdot SPI_l) - (\sum_{l=1}^{A} MPA_l)(\sum_{l=1}^{A} SPI_l)}{A \cdot (\sum_{l=1}^{A} {MPA_l}^2) - (\sum_{l=1}^{A} MPA_l)^2} \tag{3.11}$$

$$\text{and } \beta = \frac{(\sum_{l=1}^{A} SPI_l) - \alpha \cdot (\sum_{l=1}^{A} MPA_l)}{A}. \tag{3.12}$$

We note that most programs have repeating phases with periods ranging from 200 ms to 2,000 ms [53]. Numerous work exists 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, $\alpha$, $\beta$, and the reuse distance histogram can be estimated using Equations 3.9–3.12. 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 cache, which in turn can be translated to SPI values using Equations 3.2 and 3.3. 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 3.7), permitting convergence.

## 3.5 Evaluation Methodology and Results

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 processor has a 3 MB, 12-way set associative on-chip L2 cache with a line size of 64 B. 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 Section 3.4.2. 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. Experimental results indicate that CAMP has an average of 1.57% performance estimation error over all 10 benchmarks. For a more detailed description of experimental results and comparison, please refer to [94].

## 3.6 Conclusion

Cache contention among processes running on different CMP cores heavily influences performance. We have described CAMP, a predictive model that uses non-linear equilibrium equations for fast and accurate estimation of system performance degradation due to cache contention. An efficient off-line method is proposed to automate the profiling and performance prediction process with little performance overhead compared to expensive simulation. 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%.

**CHAPTER 4**

# Power Modeling

In the previous chapter, we described CAMP, a shared cache aware performance model for CMPs. By taking advantage of hardware performance counters built into most modern high-performance computers, we can automate the process of gathering 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 or software infrastructure. CAMP takes these inputs as parameters in non-linear equilibrium equations to model the steady-state effective cache sizes of different processes in a least-recently-used (LRU) or pseudo-LRU last-level cache, which are then solved using Newton-Raphson method to obtain performance estimations for each cache-sharing process. CAMP has been validated on real processors. In addition, we demonstrate the generality of CAMP by profiling processes on one CMP and using the resulting models for accurate performance estimations on two other CMPs with different cache sizes.

However, performance is not the only critical design metric to high-performance computers. With the continuously increasing system integration density and performance requirements, power consumption is becoming a major challenge for computer architects and system designers; some high-performance servers are now consuming several hundred watts [51]. High power consumption leads to increased cooling and packaging cost, reduced system reliability, and increased energy consumption and expensive electricity bills. Power modeling is an essential first step in design optimization. Accurate power models allow the architects to explore new power-efficient architectures. In addition, they are the critical building block for optimization techniques such as power-aware scheduling and assignment. Therefore, it is desirable to have a fast and accurate power model for high-performance processors.

In this chapter, we propose a fast, fully-automated technique to accurately estimate the core power consumption and the processor power consumption during runtime in a multi-programmed CMP environment with no modifications to the underlying hardware and software platform. The system-level power model takes advantage of hardware performance counters, existing on most modern processors, to infer the relationship between hardware events and processor power consumption. More specifically, the power model is derived through multi-variable linear regression using event rates recorded in 5 different hardware performance counters, implicitly accounting for time sharing and cache contention in a CMP environment. We validated the power model using SPEC CPU2000 benchmarks on a dual-core processor and a 4-core server to demonstrate the generality of the proposed modeling technique. Finally, we explain how to integrate CAMP and the power model to estimate processor power for any tentative assignment without run-time information. The combined model is validated on the 4-core server using SPEC CPU2000 benchmarks. The average estimation error is 2.38%, with a maximum error of 6.29%. Therefore, the combined model is effective in estimating processor power during assignment, useful for power-aware assignment.

The rest of the chapter is organized as follows. Section 4.1 discusses the implications of power consumption and motivates the power modeling problem for high-performance computers. Section 4.2 summarizes the past work on power modeling. Section 4.3 describes the power model construction process and elaborates on how the power model handles core-wise time sharing and chip-wise cache contention. Section 4.4 describes how to combine the proposed performance and power models to quickly and accurately estimate the processor power consumption for any process-to-core mapping during assignment. Section 4.5 describes the experimental setup for model validation and presents the validation results for the power model and the combined model. Section 4.6 concludes this chapter.

## 4.1 Introduction and Motivation

Power consumption is important due to its impact on energy cost, reliability, and thermal issues. This is especially for high-performance computers, given that modern server power consumption may reach several hundred watts. High dynamic power consumption indicates high current density, leading to accelerated wear and permanent faults due to process such as electromigration. Both dynamic power and static power including the leakage power produce heat, thereby increasing processor temperature and potentially degrading performance. Finally, high power consumption implies high energy cost, resulting in expensive electricity bills.

Although monitoring and controlling power consumption is critically important, power measurement devices are usually not available in off-the-shelf modern processors, thus making the power models necessary for monitoring and optimization purposes. However, 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) introduces the cache contention problem (see Chapter 3), impacting the power consumption and further complicating the modeling problem. Accurately modeling the performance and power consumption in a multi-programmed CMP environment is necessary for design-time architectural optimization and run-time dynamic resource management [22, 54].

Power modeling in a multi-programmed CMP 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 before processes are assigned. To the best of our knowledge, no existing 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 CAMP and the power model; and (4) our models are general enough to accommodate heterogeneous tasks and processors. Both the power model and the combined model have been validated on different machines with different architectures and nominal power consumptions. Note that although constructing the CAMP model (and thus the combined 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 is suitable for high-performance computing systems.

## 4.2 Related Work

Past work on power modeling can be roughly decomposed into two categories: simulation based power models and runtime power models. A number of researchers has proposed simulation-based power models for power analysis. Brooks et al. proposed a simulation framework named Wattch for architectural power estimation and optimizations [22]. Benini et al. proposed power macro-models for major system components including processor core, register file, instruction caches and data caches [19]. Their power models have been integrated into a system-level simulation framework for power analysis in VLIW-based embedded systems. However, such models impose significant performance overhead and are therefore inappropriate to use during runtime. In addition, the accuracy of power models is dependent on the accuracy of the underlying hardware model, thus making them platform-dependent. Other researchers have proposed performance counter based power models for on-line power estimation. Isci et al. proposed a performance-counter-based, per-unit power model for processor power estimation [54]. However, their model construction requires the die layout to identify each physical components, which is not always feasible. In addition, it targets at the power consumption of a single application; it is not straightforward to extend them for power estimation in a multi-programmed CMP environment considering CPU time multiplexing and cache contention. Economou et al. proposed a full-system power model that takes the utilization ratios of various system components as input to estimate the total system power for server environments [33]. Rivoire surveyed a number of high-level full-system power models for a variety of workloads and machine configurations [85]. However, their model uses only as many counters as can be collected at one time, therefore unnecessarily reducing estimation accuracy. Compared to our technique, their technique suffers from relatively large errors (an average error of 13% for SPECINT with a maximum error larger than 21%). Furthermore, it cannot be determined whether their model can accurately estimate power in a multi-programmed CMP environment based on their experimental results. Singh et al. proposed a performance counter based power model in a multi-programmed CMP environment [90]. This work is the closest 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 4.5, it can handle time sharing among processes and applies to CMP systems with different architectures without any changes to the model construction process.

Researchers have also proposed various techniques during assignment for optimization. Xie and Hung proposed a temperature-aware task allocation and scheduling algorithm for MPSoC embedded systems [114]. Hanumaiah et al. proposed a throughput-optimal task allocation technique under thermal constraints for CMPs [43]. However, to the best of our knowledge, we are the first to propose a solution for processor power consumption estimation during assignment.

## 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 CMP 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 $P_{\text{idle}}$ 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.

It is inaccurate to monitor all HPC-detected hardware events and associate each of them with the components generating the event due to two reasons: (1) the number of HPCs that can be monitored simultaneously is usually limited to 2 to 4 [3] due to limited number of counter registers per core. Although it is possible to rotate the event sets monitored assuming the program behavior is consistent, the assumption is unlikely to hold if it takes quite a few rotations to finish gathering all HPC values needed to generate a power estimation. In addition, event set rotation introduces additional performance overhead. (2) A lot of system components may not have a big impact on the total core power. Choosing these irrelevant features will reduce the accuracy of the power model and increase computational complexity. Therefore, it is desired to use only a limited number of system components for model construction.

In order to determine the most significant contributors among all the HPC-detected events, we use 8 SPEC CPU2000 benchmarks consisting of 5 integer programs and 3 floating point programs [1] as our training data. For each benchmark, we run one instance on each core. Note that we start these instances at the same time and gather the HPC values and power consumption for one core, assuming every core has the same power consumption and HPC values for different instances of the same program. However, it is impractical to exhaustively evaluate all combinations of HPC-detected events because there are more than 130 events available. Therefore, we associate each event with a system component, resulting in a total of 10 components. For each component, we pick an event that is most related to the access frequency of the component to represent its power consumption. Since we can only monitor 2 HPCs at the same time on the test system, we then run each benchmark 5 times to gather its power consumption and the event rates (number of events per second) of the 10 events with a sampling frequency of 10 ms. Note that the HPC values and the power consumption are measured on a real system (see Section 4.5 for detailed experimental setup). Finally, we combine the sampling data of 8 benchmarks and calculate the

[1] We used the 8 benchmarks that compiled on the test system using gcc 4.1

Pearson's correlation coefficient between each event rate and the power consumption. The top 5 events with the highest correlation coefficients are L1RPS, L2RPS, L2MPS, BRPS, and FPPS, which represent the number of L1 data cache references per second, the number of L2 cache references per second, the number of L2 cache misses per second, the number of branch instructions retired per second, and the number of floating point instructions retired per second, respectively. We ruled out components such as L1 instruction cache and ITLB because the power consumption is insensitive to the changes in their access frequency. Note that these events are supported by HPCs on most existing systems, which indicates our power model is widely applicable.

Although we have identified the 5 events that are most related to the core power, it still remains undetermined 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 4.5) 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 $P_{\text{core}}$ can be expressed as

$$P_{\text{core}} = P_{\text{idle}} + c_1 \cdot \text{L1RPS} + c_2 \cdot \text{L2RPS} + c_3 \cdot \text{L2MPS} + c_4 \cdot \text{BRPS} + c_5 \cdot \text{FPPS}, \qquad (4.1)$$

where $P_{\text{idle}}$ 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. More specifically, when a context switch from process A to process B occurs, components such as L1 caches and TLB are flushed, impacting the performance and power consumption of process B. 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. Due to the uncertainty in the timeslice length of a process, which is dynamically calculated based upon its priority and interactivity [89], 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 note that this formula needs to be adjusted if one or more processes have longer timeslice on average than other processes due to differences in priority or interactivity.

We now define the *processor power consumption* as the sum of all core power consumptions in a multi-programmed CMP 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 4.5). As indicated in Section 4.5, the amount of power increase resulting from adding a process to a nearby idle core differs greatly depending on the degree of cache contention. Therefore, the proposed power model can handle the CMP 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. In other words, if $S$ is the cartesian product of $S_i$, $i = 1, 2, ..., N$, we assume each element in $S$ is equally likely to happen. The power consumption of an idle core is $P_{idle}$. The total power consumption is the sum of each core's power consumption.

## 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 $P_{\text{process}}$ into two parts:

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

Here, $P_{\text{idle}}$ 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 L2MPR represents

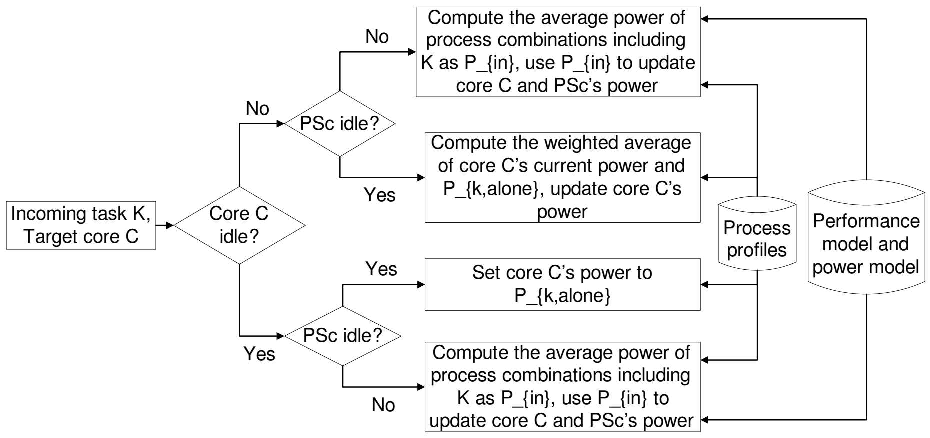

Incoming task K, Target core C
Core C idle?
No
PSc idle?
No
Compute the average power of process combinations including K as P_{in}, use P_{in} to update core C and PSc's power
Yes
Compute the weighted average of core C's current power and P_{k,alone}, update core C's power
Yes
PSc idle?
Yes
Set core C's power to P_{k,alone}
No
Compute the average power of process combinations including K as P_{in}, use P_{in} to update core C and PSc's power
Process profiles
Performance model and power model

Figure 4.1: Algorithm for power estimation for process assignment.

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 Chapter 3. Hence, if we record the instruction-related event rates during profiling for each process and use performance model in Chapter 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 Chapter 3 and Section 4.3. We also assume for each process $i$, the profiling vector $\text{PF}_i$, i.e., ($P_{i,\text{alone}}$, $\text{L1RPI}_i$, $\text{L2RPI}_i$, $\text{BRPI}_i$, $\text{FPPI}_i$) 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 $S_{\text{ex}}$, the average power consumption, denoted by $P_{\text{ex}}$, is the sum of current power consumptions of core $C$ and cores in PS$_C$. On the other hand, if we use $S_{\text{in}}$ to represent the set of process combinations that include process $K$, for each item $I$ in $S_{\text{in}}$, 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 $P_{\text{in}}$ to denote the average power consumptions for all combinations in $S_{\text{in}}$. 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 $P_{\text{rest}}$ 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 power model and the combined model.

### 4.5.1 Experimental Setup

We use PAPI 3.6.2 [6] to sample the HPCs. The sampling period is 30 ms. Our test-suite 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. [103] 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.

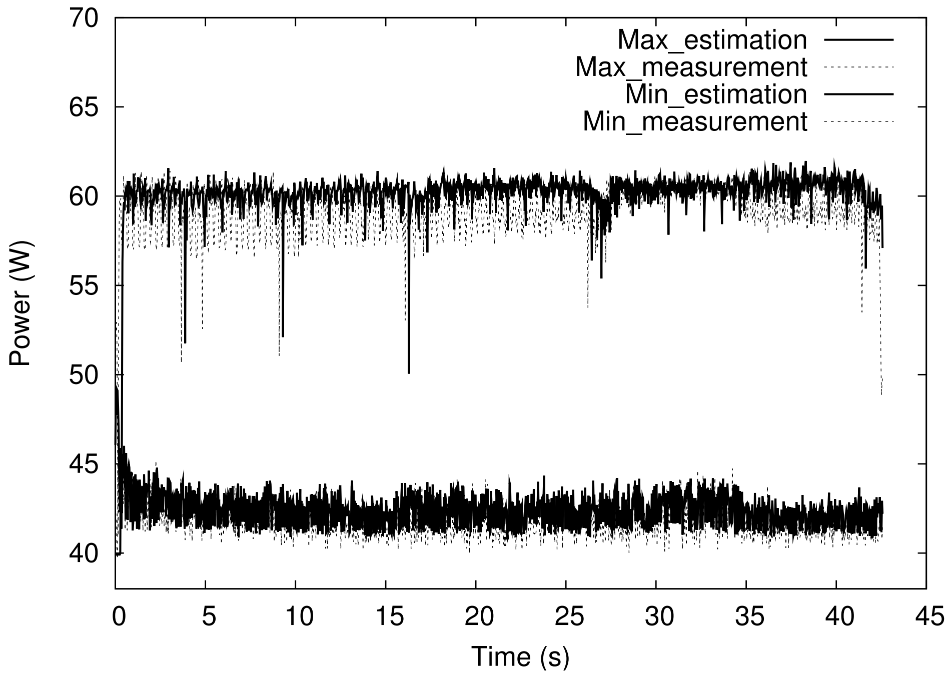


Figure 4.2: Power model validation on 4-core server. Estimations and measurements are shown for the maximum power and minimum power cases.

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 |

### 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) an Intel Core2 Quad core Q6600 processor with two chips (and two cores per chip) and 8 MB 16-way set-associative L2 cache in total, which runs Linux 2.6.28 (denoted as "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 Section 4.3.1. We then validate the power model 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

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 |

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 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

Table 4.3: Performance Model Validation

| Benchmark | | gzip | vpr | mcf | bzip2 | twolf | art | equake | ammp | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|
| MPA | E (%) | 0.16 | 2.54 | 1.33 | 2.97 | 1.91 | 1.33 | 0.42 | 3.48 | 1.76 |
| | >5% (%) | 0 | 0 | 0 | 25 | 0 | 0 | 0 | 12.5 | 4.69 |
| SPI | E (%) | 0.58 | 5.58 | 2.15 | 2.06 | 4.54 | 5.51 | 1.89 | 3.80 | 3.38 |
| | >5% (%) | 0 | 50 | 0 | 0 | 37.5 | 50 | 12.5 | 25 | 21.9 |

Table 4.4: 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 |

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

In order to validate the combined performance and power model for average power estimation during assignment on the 4-core server, we first built the CAMP model as explained in Chapter 3. More specifically, we first obtained the feature vectors of all 8 benchmarks using the stressmark. CAMP then takes the feature vector of each process for performance prediction. We measured all 36 possible pairwise combinations of 8 benchmarks: each benchmark is paired with every other benchmark (including another instance of itself) and assigned to two cache-sharing cores. The measured performance data are then compared to those predicted by CAMP.

Table 4.3 presents the average prediction error in MPA and SPI for each benchmark when it runs simultaneously with each of the 8 benchmarks. Row 2 shows the average absolute estimation error in MPA based on CAMP. Row 4 indicates the average relative estimation error in SPI. Row 3 and 5 present the percentage of test cases with an estimation error larger than 5% among all 8 test cases for each benchmark. The last column corresponds to the average result of all 8 benchmarks, i.e., 36 testcases. As indicated in Table 4.3, our technique has an average of 3.38% SPI estimation error across all 8 benchmarks with only 21.9% of the cases having an estimation error greater than 5%.

Given that both the CAMP model and the power model are applicable to the 4-core server, 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.

As indicated in Table 4.4, 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 power consumption in a multi-programmed CMP environment is challenging but essential for optimizing process assignment and migration. This chapter describes an on-line power modeling framework that rapidly and accurately estimates the power consumption of particular process-to-core mappings. This process requires no changes to existing operating system or hardware. The power model has been validated on multiple CMP machines with distinct architectures and nominal power consumptions. We conclude that the proposed framework is effective for power estimation during both process assignment and execution.

CHAPTER 5

# Thermal Modeling and Thermal Analysis

Temperature has a strong influence on integrated circuit (IC) performance, power consumption, and reliability. With continued technology scaling and increasing performance demand, thermal issues are becoming a major challenge during high-performance microprocessor design, thereby making thermal analysis an indispensable step. However, accurate thermal analysis can impose high computation costs during the IC design process. We analyze the performance and accuracies of a variety of time-domain dynamic thermal analysis techniques and use our findings to propose a new analysis technique that improves performance by 38–138$\times$ relative to popular methods such as the fourth-order globally adaptive Runge-Kutta method while maintaining accuracy. More precisely, we prove that the step sizes of step doubling based globally adaptive fourth-order Runge-Kutta method and Runge-Kutta-Fehlberg methods always converge to a constant value regardless of the initial power profile, thermal profile, and error threshold during dynamic thermal analysis. Thus, these widely-used techniques are unable to adapt to the requirements of individual problems, resulting in poor performance. We also determine the effect of using a number of temperature update functions and step size adaptation methods for dynamic thermal analysis, and identify the most promising approach considered. Based on these observations, we propose FATA, a temporally-adaptive technique for fast and accurate dynamic thermal analysis.

The rest of this chapter is organized as follows. Section 5.1 motivates the thermal analysis problem and summarizes our contributions. Section 5.2 presents the IC thermal model that characterizes the heat transfer among IC, package, heatsink, and the ambient environment. We also formulate the dynamic thermal analysis problem in the matrix-vector form. Section 5.3 explores the properties of various adaptive time-domain dynamic analysis algorithms and proves that popular numerical methods such as globally adaptive fourth-order Runge-Kutta method suffer from step size convergence problem, making them inappropriate for thermal analysis. Section 5.4 gives an overview of the proposed technique named FATA and describes the major components in FATA that enable fast and accurate simulation. Section 5.5 explains the experimental set-up, evaluates the accuracy and performance of FATA, and compares it with existing thermal analysis techniques. We also discuss the impact of various temperature update functions and step size adaptation methods on accuracy and performance. Section 5.6 concludes this chapter.

## 5.1 Introduction and Past Work

Temperature has a strong influence on integrated circuit (IC) performance, power consumption, and reliability. High temperature leads to high charge-carrier concentration, which in turn results in high leakage power consumption. In addition, it affects system performance by decreasing charge-carrier mobility, thus degrading performance, and decreasing transistor threshold voltage, thus increasing performance. Finally, it significantly increases the probability of electromigration failure rate and causes reliability problems.

Therefore, accurate and fast thermal analysis that models and analyzes microprocessor temperature is essential for performance, power, and reliability. As mentioned in Section 2.3, thermal analysis can be separated into steady-state thermal analysis and dynamic thermal analysis. In this chapter, we focus on dynamic thermal analysis because it allows us to capture dynamic temperature changes rather than steady-state thermal profiles.

IC dynamic thermal analysis models the thermal conduction from an IC's power sources through cooling packages to the ambient environment, usually using partial differential equations. In order to approximate the solutions to these equations using numerical methods, an IC model is decomposed into a large number of thermal elements such that the thermal variation within an element is negligible. This permits accurate thermal simulation. The resulting large system matrix makes direct solutions such as LU decomposition prohibitive. Traditionally, the dynamic thermal analysis problem can be solved using either frequency-domain or time-domain techniques [67, 93, 115]. Compared with frequency-domain techniques, time-domain techniques are fast and accurate for the shorter simulation runs typically encountered when power profiles frequently change. This chapter focuses on time-domain techniques.

A number of researchers have worked on the time-domain dynamic thermal analysis problem. Skadron et al. developed HotSpot [93], which uses an adaptive fourth-order Runge-Kutta method that dynamically adjusts the step size according to the local error at each time step to solve the finite difference equations. However, it is a synchronous time marching method: all the thermal elements have the same step size at each time point. Recently Yang et al. [115] developed an IC thermal analysis technique called ISAC. This technique adapts to spatial and temporal variation in material properties and power profiles. Although it achieved speedups over the fourth-order Runge-Kutta method in some circumstances, its assumption about the temperatures of the neighbors when solving finite difference equations is inaccurate (see Section 5.4), often reducing performance and/or accuracy. We know of no work that does a thorough analysis of the properties of commonly used time-domain thermal analysis techniques or points out fundamental problems with using popular finite difference techniques, such as adaptive Runge-Kutta methods, to solve the dynamic thermal analysis problem.

This chapter makes the following contributions. First, it proves that the step size will always converge to a constant value for (a) a step doubling based globally adaptive fourth-order Runge-Kutta (GARK4) method and (b) Runge-Kutta-Fehlberg method regardless of the initial power profile, thermal profile, and error threshold during dynamic thermal analysis. This reveals a fundamental and surprising performance limitation when these techniques are used for thermal analysis. Second, we propose FATA, a new fast asynchronous dynamic thermal analysis technique. This technique improves performance by 38–138$\times$ relative to popular methods such as the GARK4 method [93] and by 118.59–222.60$\times$ relative to existing work in asynchronous time-domain thermal analysis [115] with similar accuracy. Third, this article indicates the impact of various combinations of temperature update functions and step adaptation methods on performance and accuracy. Our analysis suggests that combining the trapezoidal method with third-order finite temperature difference based step size adaptation yields the best combination of performance and accuracy.

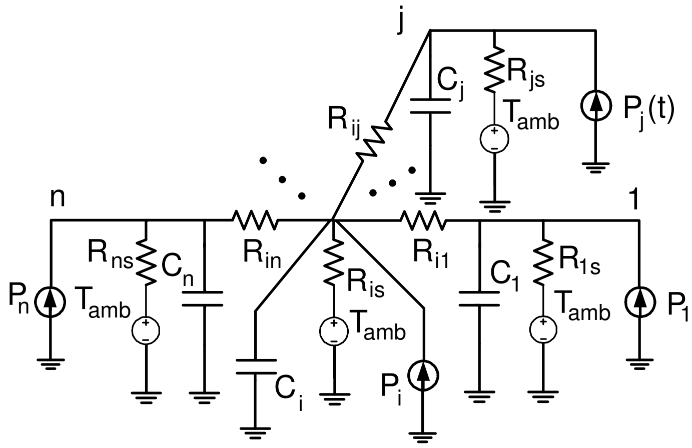


Figure 5.1: Model for a single thermal element.

## 5.2 Thermal Model and Problem Formulation

In this section, we first give details on the IC thermal model that characterizes the heat transfer among IC, package, heatsink, and the ambient environment. We then formulate the dynamic thermal analysis problem in the matrix-vector form.

Heat and electrical conduction are both governed by the diffusion equation and can be similarly modeled; thermal conductance corresponds to electrical conductance, heat capacity corresponds to electrical capacitance, temperature corresponds to voltage, and power dissipation corresponds to electrical current. We model a chip as a regular mesh containing $N$ discretized elements, or a *thermal grid*, with the ambient temperature corresponding to ground. Each element has a ground capacitance and a ground thermal conductance. In the case where no heat dissipation channel exists between a thermal element and the ambient, the corresponding ground thermal conductance is set to zero. Physically adjacent elements are connected with resistors. The power consumption of each thermal element is modeled as a current source with current flowing into the element. Using this model, the thermal grid can be modeled as a linear system.

Figure 5.1 illustrates the model for a given element $i$, which is connected to $n$ neighbors via resistors. Ground represents the ambient temperature. We use $N_i$ to represent element $i$'s neighbors. Given $n \in N_i$, $T_i(t)$ is the temperature of element $i$ at time t, $T_n(t)$ is the temperature of element $i$'s neighbor $n$, $T_{amb}$ is the ambient temperature, $C_i$ is the ground capacitance at element $i$, $P_i(t)$ is the heat source associated with element $i$, $R_{is}$ is the ground thermal resistance at element $i$, and $R_{in}$ is the resistance between element $i$ and its neighbor $n$, Kirchhoff's Current Law can be used to derive the following equation:

$$\left(\sum_{n\in N_i} \frac{T_i(t) - T_n(t)}{R_{in}}\right) + \frac{T_i(t) - T_{amb}}{R_{is}} + C_i \frac{dT_i(t)}{dt} - P_i(t) = 0. \tag{5.1}$$

Equation 5.1 can be simplified as follows:

$$\begin{aligned} \text{Letting } T_i' &= T_i - T_{amb}, \forall i \text{ in the thermal grid and} && (5.2)\\ \alpha_i &= \sum_{n\in N_i} \frac{1}{R_{in}} + \frac{1}{R_{is}}, && (5.3)\\ \frac{dT_i'(t)}{dt} &= \frac{1}{C_i}(\sum_{n\in N_i} \frac{T_n'(t)}{R_{in}} - \alpha_i T_i'(t) + P_i(t)). && (5.4) \end{aligned}$$

For convenience, we will use $T_i(t)$ instead of $T_i'(t)$ to represent the normalized temperature of element $i$, i.e., the difference between element $i$'s temperature and the ambient temperature. Assume the $j$th neighbor of element $i$ is $k_j$, we define $\overrightarrow{g_i}$ as

$$\overrightarrow{g_i} = \left(\underbrace{0,\cdots,0}_{k_1-1}, \frac{1}{R_{i1}}, \underbrace{0,\cdots,0}_{k_2-1}, \frac{1}{R_{i2}}, \cdots, -\alpha, \cdots, \frac{1}{R_{in_i}}, \cdots, 0\right), \tag{5.5}$$

where $\frac{1}{R_{ij}}$ is the $k_j$th entry of vector $\overrightarrow{g_i}$, representing the thermal conductance between $i$ and its $j$th neighbor $k_j$, $-\alpha$ (defined in Equation 5.3) is the $i$th entry of the vector, and all other entries are 0s. Similarly, we use $\overrightarrow{T(t)}$ and $\overrightarrow{T^{(1)}(t)}$ to represent the temperature and first-order temperature derivatives of all $N$ elements in the system at time $t$, i.e.,

$$\begin{aligned} \overrightarrow{T(t)} &= (T_1(t), T_2(t), \cdots, T_N(t))^T \text{ and} \qquad (5.6) \\ \overrightarrow{T^{(1)}(t)} &= (dT_1(t)/dt, dT_2(t)/dt, \cdots, dT_N(t)/dt)^T. \qquad (5.7) \end{aligned}$$

Hence, Equation 5.4 can be written as

$$\frac{dT_i(t)}{dt} = \left(\overrightarrow{g_i} \cdot \overrightarrow{T(t)} + P_i(t)\right)/C_i. \qquad (5.8)$$

Equation 5.4 holds for all elements in the system. If we define system matrix $\mathbf{A}$ (size $N \times N$), thermal capacitance matrix $\mathbf{C}$ (size $N \times N$), and $N$-dimensional power vector $\overrightarrow{P(t)}$ as

$$\begin{aligned} \mathbf{A} &= \left(\overrightarrow{g_1}^T, \overrightarrow{g_2}^T, \cdots, \overrightarrow{g_N}^T\right)^T, \qquad (5.9) \\ \mathbf{C} &= diag\left(C_1, C_2, \cdots, C_N\right), \text{and} \qquad (5.10) \\ \overrightarrow{P(t)} &= (P_1(t), P_2(t), \cdots, P_N(t))^T, \qquad (5.11) \end{aligned}$$

the dynamic thermal analysis problem can be described using the following equation: $\overrightarrow{T^{(1)}(t)} = \mathbf{C}^{-1}\left(\mathbf{A}\overrightarrow{T(t)} + \overrightarrow{P(t)}\right)$. However, it is computationally expensive to directly solve the system equation due to the high order of system matrices $\mathbf{A}$ and $\mathbf{C}$ for a typical thermal model with tens of thousands of discrete elements. A common approach is to divide time into discrete time steps and approximate the solutions using finite temperature difference equations, i.e., using finite difference methods.

## 5.3 Globally Adaptive Runge-Kutta Methods: Are They Really Adaptive?

Traditionally, finite difference equations are solved by synchronous numerical methods. Runge-Kutta methods such as the fourth-order Runge-Kutta method and the Runge-Kutta-Fehlberg (RKF) method are commonly used solve ordinary differential equations [93, 115]. We have determined that, despite their popularity, these step size control methods are not appropriate for thermal analysis.

**Theorem 5.3.1** (Step Size Convergence Property)**.** *Given an IC thermal model with $N$ thermal elements that satisfy Equation 5.4, the step size of (1) a step doubling based globally adaptive 4th-order Runge-Kutta (GARK4) method and (2) RKF method converge to a constant value $c_h$ regardless of initial step size and specified error threshold.*

*Proof.* Since the proofs for both methods are similar, we present only the proof for the synchronous, adaptive GARK4 method used in HotSpot [93]. We assume the power profile, $\overrightarrow{P}$, is constant during a simulation run. Given that $\overrightarrow{T_k}$ is the temperature vector at iteration $k$, $\overrightarrow{T_k^{(1)}}$ is the first-order temperature derivative at iteration $k$, and $h_{k+1}$ is the step size at iteration $k+1$, the temperature vector at iteration $k+1$, i.e., $\overrightarrow{T_{k+1}}$ can be expressed as follows:

$$\begin{aligned}
\overrightarrow{k_1} &= \overrightarrow{T_k^{(1)}} = \mathbf{C}^{-1}\left(\mathbf{A}\overrightarrow{T_k} + \overrightarrow{P}\right), && (5.12)\\
\overrightarrow{k_2} &= \mathbf{C}^{-1}\left[\mathbf{A}\left(\overrightarrow{T_k} + 1/2h_{k+1}\overrightarrow{k_1}\right) + \overrightarrow{P}\right] \\
&= \overrightarrow{k_1} + 1/2h_{k+1}\mathbf{C}^{-1}\mathbf{A}\overrightarrow{k_1}, && (5.13)\\
\overrightarrow{k_3} &= \overrightarrow{k_1} + 1/2h_{k+1}\mathbf{C}^{-1}\mathbf{A}\overrightarrow{k_2}, \text{ and} && (5.14)\\
\overrightarrow{k_4} &= \overrightarrow{k_1} + h_{k+1}\mathbf{C}^{-1}\mathbf{A}\overrightarrow{k_3}, \text{ yielding} && (5.15)\\
\overrightarrow{T_{k+1}} &= \overrightarrow{T_k} + 1/6h_{k+1}\left(\overrightarrow{k_1} + 2\overrightarrow{k_2} + 2\overrightarrow{k_3} + \overrightarrow{k_4}\right). && (5.16)
\end{aligned}$$

Starting from Equations 5.12, 5.13, 5.14, and 5.15, substitute the corresponding terms into Equation 5.16, resulting in

$$\overrightarrow{T_{k+1}} = \overrightarrow{T_k} + \sum_{n=1}^{4}\frac{h_{k+1}^n}{n!}(\mathbf{C}^{-1}\mathbf{A})^{n-1}\overrightarrow{k_1}. \tag{5.17}$$

Next, we present the temperature and step size update functions in the matrix-vector form. Using Equation 5.12, we introduce a few variables to simplify Equation 5.17.

$$\begin{aligned}
\text{Let } \mathbf{B} &= \mathbf{C}^{-1}\mathbf{A} \text{ and } \mathbf{D} = \mathbf{C}^{-1}\overrightarrow{P}. \\
\overrightarrow{T_{k+1}} &= \overrightarrow{T_k} + \left(\mathbf{B}\overrightarrow{T_k} + \mathbf{D}\right)\sum_{n=1}^{4}\frac{h_{k+1}^n}{n!}\mathbf{B}^{n-1} \\
&= \sum_{n=0}^{4}\frac{(h_{k+1}\mathbf{B})^n}{n!}\overrightarrow{T_k} + \sum_{n=1}^{4}\frac{(h_{k+1}^n\mathbf{B}^{n-1})}{n!}\mathbf{D}. && (5.18)
\end{aligned}$$

Given that $f_{\mathbf{B}}(h) = \sum_{n=0}^{4} \frac{(h\mathbf{B})^n}{n!}$, the temperature vector at iteration $k+1$ is

$$\overrightarrow{T_{k+1}} = f_{\mathbf{B}}(h_{k+1})\overrightarrow{T_k} + (f_{\mathbf{B}}(h_{k+1}) - \mathbf{I}_{N\times N})\mathbf{B}^{-1}\mathbf{D}, \tag{5.19}$$

where $\mathbf{I}_{N\times N}$ is a $N \times N$ unit matrix. Given that $\overrightarrow{T_k}$ and $h_k$ are the temperature vector and step size at iteration $k$, step doubling based step size adaptation first determines the absolute difference between the results computed by taking one $h_k$ step and two $\frac{h_k}{2}$ steps, where $\overrightarrow{T_{k+1,s}}$ is the temperature vector at iteration $k+1$ using one step and $\overrightarrow{T_{k+1,d}}$ is the predicted temperature vector at iteration $k+1$ using two steps.

$$\begin{aligned}
\overrightarrow{T_{k+\frac{1}{2}}} &= f_{\mathbf{B}}(\frac{h_k}{2})\overrightarrow{T_k} + (f_{\mathbf{B}}(\frac{h_k}{2}) - \mathbf{I}_{N\times N})\mathbf{B}^{-1}\mathbf{D}, && (5.20)\\
\overrightarrow{T_{k+1,s}} &= f_{\mathbf{B}}(h_k)\overrightarrow{T_k} + (f_{\mathbf{B}}(h_k) - \mathbf{I}_{N\times N})\mathbf{B}^{-1}\mathbf{D}, && (5.21)\\
\overrightarrow{T_{k+1,d}} &= f_{\mathbf{B}}(\frac{h_k}{2})\overrightarrow{T_{k+\frac{1}{2}}} + (f_{\mathbf{B}}(\frac{h_k}{2}) - \mathbf{I}_{N\times N})\mathbf{B}^{-1}\mathbf{D}. && (5.22)
\end{aligned}$$

The step size for the next iteration, $h_{k+1}$, is calculated by dividing the error threshold by that difference, as shown below:

$$h_{k+1} = h_k \times \left(\epsilon / ||\overrightarrow{T_{k+1,d}} - \overrightarrow{T_{k+1,s}}||_\infty\right)^{\frac{1}{5}}, \tag{5.23}$$

where $\epsilon$ is a user-specified error threshold used to control accuracy. Combining Equations 5.20, 5.22, and 5.21 yields the next safe step size:

$$h_{k+1} = h_k \times \left(\epsilon / ||[f_{\mathbf{B}}^2(\frac{h_k}{2}) - f_{\mathbf{B}}(h_k)](\overrightarrow{T_k} + \mathbf{B}^{-1}\mathbf{D})||_\infty\right)^{\frac{1}{5}}. \tag{5.24}$$

Equation 5.19 and Equation 5.24 can be simplified by defining $\overrightarrow{Y_k} = \overrightarrow{T_k} + \mathbf{B}^{-1}\mathbf{D}$, yielding the following temperature and step size update functions:

$$\begin{aligned} h_{k+1} &= h_k \times \left( \epsilon / || [f_{\mathbf{B}}^2(\frac{h_k}{2}) - f_{\mathbf{B}}(h_k)] \overrightarrow{Y_k} ||_\infty \right)^{\frac{1}{5}} && (5.25) \\ \overrightarrow{Y_{k+1}} &= f_{\mathbf{B}}(h_{k+1}) \overrightarrow{Y_k}. && (5.26) \end{aligned}$$

We now prove that the numerical solution of $\overrightarrow{Y}$ converges to a constant vector regardless of the user-specified error threshold $\epsilon$. This will be used to prove step size convergence, i.e., $h_k$ will converge to a constant. Due to the characteristics of RC linear systems, the exact temperature vector (or $\overrightarrow{Y_{true}}$) will become stable as time proceeds to infinity, i.e., $\lim_{k\to\infty} |\overrightarrow{Y_{k+1,true}} - \overrightarrow{Y_{k,true}}| = 0$. Given that step doubling is used to control the step size, we have $|\overrightarrow{Y_k} - \overrightarrow{Y_{k,true}}| = O(h_k^6)$ and $|\overrightarrow{Y_{k+1}} - \overrightarrow{Y_{k+1,true}}| = O(h_{k+1}^6)$, where $\overrightarrow{Y_k}$ and $\overrightarrow{Y_{k+1}}$ are obtained by GARK4 at iterations $k$ and $k+1$ [77]. Therefore, when the thermal profile reaches steady state,

$$\begin{aligned} |\overrightarrow{Y_{k+1}} - \overrightarrow{Y_k}| &= |\overrightarrow{Y_{k+1}} - \overrightarrow{Y_{k+1,true}} + \overrightarrow{Y_{k+1,true}} - \overrightarrow{Y_{k,true}} + \\ &\quad \overrightarrow{Y_{k,true}} - \overrightarrow{Y_k}| \\ &\le |\overrightarrow{Y_{k+1}} - \overrightarrow{Y_{k+1,true}}| + \\ &\quad |\overrightarrow{Y_{k+1,true}} - \overrightarrow{Y_{k,true}}| + |\overrightarrow{Y_{k,true}} - \overrightarrow{Y_k}| \\ &= O(h_k^6) + O(h_{k+1}^6). && (5.27) \end{aligned}$$

As shown later in the section, the steady-state step size is on the order of $10^{-6}$, i.e., the difference between two consecutive $\overrightarrow{Y}$ vectors, is on the order of $10^{-36}$, which is dominated by numerical error. Hence, we assume $\lim_{k\to\infty}\overrightarrow{Y_k} = \overrightarrow{c_Y}$, where $\overrightarrow{c_Y}$ is a constant vector. If we use $c_h$ to represent the step size in steady state, Equation 5.26 gives us

$$\overrightarrow{c_Y} \;=\; \lim_{k\to\infty}\overrightarrow{Y_{k+1}} = f_{\mathbf{B}}(c_h)\lim_{k\to\infty}\overrightarrow{Y_k} = f_{\mathbf{B}}(c_h)\overrightarrow{c_Y}. \tag{5.28}$$

Therefore, $f_{\mathbf{B}}(c_h)$ has an eigenvalue of 1, with $\overrightarrow{c_Y}$ being one of the corresponding eigenvectors. Note that this argument still holds in the presence of a typical numerical error of $10^{-16}$. We omit the numerical analysis here due to space limitations. According to Equation 5.28, $\overrightarrow{c_Y} = f_{\mathbf{B}}(c_h)\overrightarrow{c_Y} = \cdots = \lim_{k\to\infty} f_{\mathbf{B}}^k(c_h)\overrightarrow{c_Y}$. Based on the matrix analysis theory on the relationship between convergence of matrix powers and eigenvalues, given that $\lambda_1(f_{\mathbf{B}}(h)), \lambda_2(f_{\mathbf{B}}(h)), \cdots, \lambda_N(f_{\mathbf{B}}(h))$ are the $N$ eigenvalues of $f_{\mathbf{B}}(h)$, $\lim_{k\to\infty} f_{\mathbf{B}}^k(h)$ exists if and only if $\max_{1\le i\le N}|\lambda_i(f_{\mathbf{B}}(h))| \le 1$, with $|\lambda_i(f_{\mathbf{B}}(h))| = 1$ only if $\lambda_i(f_{\mathbf{B}}(h))$ is not defective and $\lambda_i(f_{\mathbf{B}}(h)) = 1$ [47].

We then determine the relationship between $c_h$ and $\max_{1\le i\le N}|\lambda_i(f_{\mathbf{B}}(h))|$. $\mathbf{A}$ is real and symmetric because $R_{ij} = R_{ji}, \forall 1 \le i, j \le N$. We also notice that each row of matrix $\mathbf{A}$ satisfies

$$\forall i : |a_{ii}| - \sum_{j=1,j\ne i}^{N} |a_{ij}| = \frac{1}{R_{is}} \ge 0. \tag{5.29}$$

Furthermore, there exists at least one positive thermal conductance $\frac{1}{R_{ks}}$ between element $k$ and the ambient such that the heat flow into the chip can be conducted through $R_{ks}$ to the ambient, i.e., $|a_{kk}| > \sum_{j=1,j\ne k}^{N}|a_{kj}|$. Therefore, $\mathbf{A}$ is a Hermitian diagonally dominant matrix with negative diagonal elements and non-positive off-diagonal elements. This indicates $\mathbf{A}$ is invertible and negative semi-definite. Define diagonal matrix $\mathbf{Q}$ to have an $i$th element that is the reciprocal of the square root of the $i$th element of the thermal capacitance matrix $\mathbf{C}$, i.e., $\forall_{1\leq i\leq N}\ :\ q_{ii} = \frac{1}{\sqrt{c_{ii}}}$. Note that $\mathbf{Q}^{-1}\mathbf{C}^{-1} = \mathbf{Q}$. Thus, $\mathbf{Q}^{-1}\mathbf{B}\mathbf{Q} = \mathbf{Q}^{-1}\mathbf{C}^{-1}\mathbf{A}\mathbf{Q} = \mathbf{Q}\mathbf{A}\mathbf{Q} = \mathbf{Q}^T\mathbf{A}\mathbf{Q}$. Since $\mathbf{A}$ is negative semi-definite, $\mathbf{Q}^T\mathbf{A}\mathbf{Q}$ is also negative semi-definite. Given $\mathbf{Q}^T\mathbf{A}\mathbf{Q}$ is also real and symmetric, its eigenvalues are non-defective and non-positive. It has no zero eigenvalues because it is invertible ($\mathbf{A}$ is invertible). Therefore, $\mathbf{Q}^T\mathbf{A}\mathbf{Q}$ is diagonalizable with negative eigenvalues. $\mathbf{B}$ is similar to $\mathbf{Q}^T\mathbf{A}\mathbf{Q}$, $\mathbf{B}$ is also diagonalizable with negative eigenvalues, i.e., there exists an invertible matrix $\mathbf{P}$ such that $\mathbf{B} = \mathbf{P} \times \Lambda(\mathbf{B}) \times \mathbf{P}^{-1}$, where $\Lambda(\mathbf{B})$ is the diagonal matrix consisting of $\mathbf{B}$'s eigenvalues. Thus, we can express $f_{\mathbf{B}}(h)$ as follows: $f_{\mathbf{B}}(h) = \mathbf{P}\sum_{n=0}^{4}\frac{(h\Lambda(\mathbf{B}))^n}{n!}\mathbf{P}^{-1}$. Hence, if $\lambda_i(\mathbf{B})$ is the $i$th eigenvalue of $\mathbf{B}$, the $i$th eigenvalue of $f_{\mathbf{B}}(h)$ is $\lambda_i(f_{\mathbf{B}}(h)) = \sum_{n=0}^{4}\frac{(h\lambda_i(\mathbf{B}))^n}{n!}$. Since function $f(x) = \sum_{n=0}^{4}\frac{x^n}{n!}$ is monotonically decreasing when $x < 0$, we have $f(x) = 1 \Leftrightarrow x = -2.785$ when $x$ is negative. We know (1) 1 is an eigenvalue of $f_{\mathbf{B}}(c_h)$ and (2) $h\lambda_i(\mathbf{B}) < 0, 1 \leq i \leq N$, so the steady-state step size is

$$c_h = 2.785/\max{}_{1\leq i\leq N}|\lambda_i(\mathbf{B})|. \tag{5.30}$$

Note that $\max_{1\leq i\leq N}|\lambda_i(\mathbf{B})|$ can be found numerically, e.g., using von Mises' power method [44]. Thus, we have proven that the step size of the GARK4 method with step doubling based step size adaptation converges to $c_h$ regardless of initial thermal profile and error threshold. This conclusion has been validated using different thermal grid structures in Hotspot 4.0 [93] and ISAC [115]. Note that this argument also holds for step doubling based low-order explicit methods such as the forward Euler, other variants of step doubling adaptation [93], and RKF. For example, HotSpot uses a variant of step doubling method in which auxiliary bounds are imposed on the maximum and minimum safe step size relative to the current step size. However, the constraints have no impact on the steady-state thermal behavior, i.e., Equations 5.28 and 5.30 still hold, leading to the same steady-state step size. A similar proof also applies to RKF. $\square$

### Implications of Step Size Convergence Property

In each iteration, the step size adaptation function calculates a new safe step size based on the current thermal profile activity. Intuition suggests that it will use small steps when the chip temperature is rapidly changing and large step sizes when there is little change in temperature. When the IC thermal profile reaches steady state, the temperature function can be accurately approximated with very large step sizes. Therefore, the steady-state step size will generally be the longest encountered during dynamic thermal analysis. We validated our conclusion using different benchmarks and different grid structures (see Section 5.5.1). We found that the percentage of step sizes exceeding $c_h$ (the step size after temperature convergence) ranges from 0.3% to 0.8%, i.e., the steady-state step size is only rarely exceeded. In summary, the step sizes of the step doubling based GARK4 method converge to $c_h$ long before IC thermal profile reaches steady state, significantly degrading performance.

### Steady-State Step Size Analysis

In this section, we give a rough estimate of the steady-state step size using the thermal resistances and thermal capacitances in the thermal grid based on Equation 5.30. Similar analysis can also be applied to RKF method. Given trace$(\mathbf{B}) = \sum_{i=1}^{N} b_{ii} = \sum_{i=1}^{N} \lambda_i(\mathbf{B}) \leq N \times \max_{1 \leq i \leq N} |\lambda_i(\mathbf{B})|$, we can estimate the spectral radius of $\mathbf{B}$ as $\max_{1 \leq i \leq N} |\lambda_i(\mathbf{B})| \geq$

$\frac{\sum_{i=1}^{N} b_{ii}}{N}$. We define $\frac{1}{\tau_{avg}} = \frac{\sum_{1 \le i,j \le N, i \ne j} \frac{1}{R_{ij}C_{ii}}}{m}$, where $m$ is the number of non-zero off-diagonal entries in $\mathbf{B}$. Since $\sum_{i=1}^{N} b_{ii} = \sum_{1 \le i,j \le N, i \ne j} \frac{1}{R_{ij}C_{ii}} = \frac{m}{\tau_{avg}}$, the estimated steady-state step size $c_{h,s}$ is

$$c_{h,s} = \frac{2.785}{\max_{1 \le i \le N} |\lambda_i(\mathbf{B})|} \le \frac{2.785}{\frac{m}{N\tau_{avg}}} = \frac{2.785N}{m}\tau_{avg}. \tag{5.31}$$

Note that $\tau_{avg}$ is the harmonic mean of the RC constants of different thermal elements. In our validation experiments, $\frac{m}{N} \approx 6$. Therefore, $c_{h,s} \approx 0.464\tau_{avg}$. Furthermore, $\tau_{avg}$ usually underestimates the RC constant of the chip. In our test cases, $\tau_{avg}$ is approximately 10 µs while the thermal RC constant associated with the IC, i.e., $\tau_{IC}$, is on the order of 100 µs [92], i.e., $\tau_{IC} \approx 10\tau_{avg}$. This indicates the stable step size is approximately $\frac{1}{10} \times 0.464\tau_{IC} \approx 0.05\tau_{IC}$, even when the thermal profile is perfectly approximated by the temperature update function, i.e., the thermal profile is stable. Hence, the steady-state step size is severely limited for explicit step-doubling GARK4 methods.

## 5.4 FATA: Fast Asynchronous Time Marching Technique

In this section, we first give an overview of the proposed technique: FATA. We then describe the major components in FATA that enable fast and accurate simulation.

### 5.4.1 Algorithm Overview

FATA is an adaptive asynchronous time marching finite-difference method. Compared to a synchronous method, FATA permits different elements to have different step sizes with their own local times. Figure 5.2 illustrates the algorithm in FATA. During dynamic thermal analysis, the algorithm takes, as input, an initial thermal profile, a power profile, and various

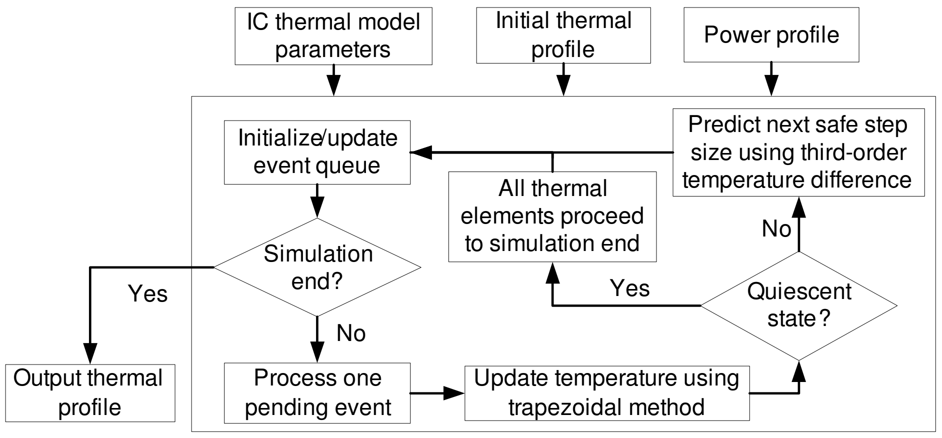

IC thermal model parameters
Initial thermal profile
Power profile
Initialize/update event queue
Predict next safe step size using third-order temperature difference
All thermal elements proceed to simulation end
No
Simulation end?
Yes
Yes
Quiescent state?
No
Output thermal profile
Process one pending event
Update temperature using trapezoidal method

Figure 5.2: Overview of asynchronous time marching algorithm in FATA.

IC thermal model parameters such as material thermal conductivities and heat capacities. After determining the initial step size for each element, FATA initializes an event queue containing temperature update events sorted by their target times, i.e., the element's current time plus its step size. In each iteration, the event with the earliest target time is selected and the corresponding element's temperature is updated. It then determines whether the thermal profile of the chip has reached quiescent state and if so, advances the local times of all the thermal elements to the user specified simulation end time. Otherwise, FATA calculates the element's next safe step size and reinserts the temperature update event into the event queue with a new target time. This process is repeated until the user specified simulation end time is reached. As illustrated in Figure 5.2, the major components in FATA are temperature update, step size adaptation, and quiescence detection. The following sections explain these components.

### 5.4.2 Temperature Update

Existing asynchronous methods [115] use exponential functions to update the temperature of the target element $i$, i.e., the element under consideration. The derivation is based on the assumption that the temperature of the neighbors of element $i$ do not change over

$[t_i, t_i + h_i]$, where $t_i$ is element $i$'s local time and $h_i$ is element $i$'s current step size. This assumption can lead to a large error when the neighboring nodes experience significant temperature changes during that period. Worse yet, these errors can accumulate as time advances, resulting in errors in temperature and step size calculation that degrade performance or accuracy. We therefore propose modeling temperature change using a variant of trapezoidal method that is tailored to asynchronous time marching. First, note that Equation 5.4 can be simplified as follows:

$$\text{Let } \beta = \sum_{n \in N} \frac{T_n(t)}{R_{in}} + P_i(t) = \alpha T_i(t) + C_i \frac{dT_i(t)}{dt}. \tag{5.32}$$

The trapezoidal method can be used to extrapolate element $i$'s voltage at the target time $t_i + h_i$:

$$T_i(t_i + h_i) = T_i(t_i) + \frac{h_i}{2}(f_i'(t_i) + f_i'(t_i + h_i)), \tag{5.33}$$

where $f_i'(t_i)$ is the element $i$'s temperature derivative at $t_i$ and $f_i'(t_i + h_i)$ is the corresponding derivative at $t_i + h_i$. Given target time $t = t_i + h_i$, combining Equation 5.32 and Equation 5.33 yields

$$T_i(t_i + h_i) = \frac{\beta h_i + C_i h_i f_i'(t_i) + 2C_i T_i(t_i)}{\alpha h_i + 2C_i} \tag{5.34}$$

However, we still face the problem of computing $T_n(t_i + h_i)$ ($n \in N_i$) to obtain $\beta$ at target time $t_i + h_i$. The trapezoidal method cannot be used to compute neighbor temperatures, for that would result in circular dependency problems. More specifically, $T_n(t_i + h_i)$ must be known before $T_i(t_i + h_i)$ is computed. Similarly, $T_n(t_i + h_i)$ depends on $T_i(t_i + h_i)$. To solve this problem, we use the forward Euler method to extrapolate $T_n(t_i + h_i)$ based on $T_n(t_n)$ and $f'_n(t_n)$, where $T_n(t_n)$ represents element $n$'s temperature at $t_n$ and $f'_n(t_n)$ represents the derivative of element $n$'s temperature at $t_n$. We experimented with approximation functions with different orders and determined combining the trapezoidal method with the forward Euler method achieves a good balance between accuracy and performance.

### 5.4.3 Step Size Adaptation

Given the trapezoidal method we use to estimate $T_i(t)$, the local truncation error in step $n$ of element $i$ can be expressed as $\epsilon_{in} = \frac{h_{i,n}^3 f_i'''(\zeta)}{12}$, where $h_{i,n}$ is the size of step $n$, $\zeta$ is between local times $t_n$ and $t_{n+1}$, and $f_i'''(\zeta)$ is the third-order derivative of $i$'s temperature at time $\zeta$. In practice, we approximate the step size using a third-order divided difference. For element $i$ at time step $n$, the third-order finite difference is expressed as follows:

$$\begin{aligned} DD_1(t_n) &= (T_i(t_n) - T_i(t_{n-1}))/(t_n - t_{n-1}), && (5.35) \\ DD_2(t_n) &= (DD_1(t_n) - DD_1(t_{n-1})/(t_n - t_{n-1}), && (5.36) \\ DD_3(t_n) &= (DD_2(t_n) - DD_2(t_{n-1})/(t_n - t_{n-1}), && (5.37) \\ \epsilon_{in} &= h_{i,n}^3 f_i'''(\zeta)/12 = DD_3(t_n)/2, && (5.38) \end{aligned}$$

where $t_n$ and $t_{n-1}$ are the local times at time step $n$ and $n-1$. The $(n+1)th$ step size estimation is thus given by

$$h_{i,n+1} = k_1 * \left( \frac{(\text{RELTOL} * |T_i(t_n)| + \text{ABSTOL})}{\max(\text{ABSTOL}, \epsilon_{in})} \right)^{k_2}. \tag{5.39}$$

Table 5.1: Comparison of GARK4, ISAC, and FATA

| Problem | GARK4 CPU time (s) | GARK4 Error (%) | ISAC CPU time (s) | ISAC Error (%) | ISAC Speedup (×) | FATA CPU time (s) | FATA Error (%) | FATA Speedup (×) |
|---|---|---|---|---|---|---|---|---|
| dct_ijpeg | 2.05 | 0.01 | 10.67 | 0.04 | 0.19 | 0.05 | 0.03 | 37.86 |
| dct_lee | 32.74 | 0.00 | 43.16 | 0.08 | 0.76 | 0.27 | 0.03 | 122.55 |
| dct_wang | 36.93 | 0.00 | 32.02 | 0.1 | 1.15 | 0.27 | 0.02 | 138.30 |
| jacobi | 2.70 | 0.01 | 8.82 | 0.02 | 0.31 | 0.04 | 0.1 | 70.68 |
| mac | 2.04 | 0.01 | 11.13 | 0.03 | 0.18 | 0.05 | 0.03 | 38.33 |
| pr2 | 31.71 | 0.00 | 85.15 | 0.03 | 0.37 | 0.39 | 0.09 | 80.56 |
| rand100 | 33.87 | 0.00 | 47.51 | 0.08 | 0.71 | 0.31 | 0.05 | 109.58 |
| rand200 | 14.60 | 0.00 | 48.14 | 0.02 | 0.30 | 0.28 | 0.05 | 52.57 |

where ABSTOL and RELTOL are the maximum tolerable absolute and relative temperature errors. $k_1$ and $k_2$ are determined empirically to achieve a good balance between accuracy and speed. In practice, we use a $k_1$ of 1.5 and a $k_2$ of 0.3. This method of computing a new step size is similar to that in SPICE3 [81]. However, that formula uses a more complicated step control algorithm that also considers the maximum number of iteration at a given time point during its iterative solving process.

When the power profile is mostly static for a long time and the IC thermal profile approaches its steady state, i.e., the system becomes quiescent, we advance all nodes to the simulation end time. This step is called *quiescence detection*.[1]

## 5.5 Evaluation

In this section, we evaluate FATA and compare it with existing thermal analysis techniques. Experiments were conducted on a Linux workstation equipped with a 1 GHz AMD Sempron 3100 Processor and 1 GB memory. We use a non-adaptive lock-step RK4 method with a very small step size as our accuracy (but not speed) reference; error values are computed relative to this reference. We first compare the accuracy and analysis times of the step

[1]We omit the details due to space constraints. A extended technical report will be published.

Table 5.2: Comparison Among Different Combinations of Temperature Update Functions and Step Size Adaptation Techniques

| Problem | TR w. Step Doubling | | | FE w. Step Doubling | | | FE w. DD3 | | | TR w. DD3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | CPU time (s) | Error (%) | Slowdown (×) | CPU time (s) | Error (%) | Slowdown (×) | CPU time (s) | Error (%) | Slowdown (×) | CPU time (s) | Error (%) |
| dct_ijpeg | 0.39 | 0.02 | 6.93 | 9.84 | 0.06 | 174.68 | 5.65 | 0.07 | 100.23 | 0.06 | 0.04 |
| dct_lee | 1.57 | 0.05 | 3.29 | 47.62 | 0.03 | 99.90 | 40.60 | 0.03 | 85.17 | 0.48 | 0.02 |
| dct_wang | 1.46 | 0.06 | 2.64 | 49.00 | 0.02 | 88.23 | 41.22 | 0.02 | 74.22 | 0.56 | 0.02 |
| jacobi | 0.27 | 0.01 | 5.17 | 6.52 | 0.03 | 123.29 | 4.88 | 0.04 | 92.31 | 0.05 | 0.07 |
| mac | 0.39 | 0.01 | 7.06 | 9.79 | 0.05 | 177.20 | 5.66 | 0.07 | 102.48 | 0.06 | 0.05 |
| pr2 | 2.17 | 0.05 | 4.34 | 66.70 | 0.06 | 133.25 | 50.29 | 0.07 | 100.47 | 0.50 | 0.05 |
| rand100 | 1.69 | 0.07 | 3.33 | 50.32 | 0.03 | 98.86 | 43.52 | 0.04 | 85.49 | 0.51 | 0.02 |
| rand200 | 1.40 | 0.02 | 4.17 | 39.33 | 0.06 | 117.20 | 33.47 | 0.06 | 99.75 | 0.34 | 0.03 |

doubling based GARK4 method, ISAC, and FATA. We also compare temperature update functions and step adaptation methods.

### 5.5.1 Comparison of GARK4, ISAC, and FATA

This section reports the accuracy and analysis time of the GARK4, ISAC, and FATA. Note that speedup over GARK4 claimed for ISAC in prior work [115] was incorrect due to an implementation error in the step size adjustment algorithm of the reference GARK4 method. We used eight real and synthetic behavioral synthesis benchmarks with different grid structures to evaluate the candidate analysis techniques. The dynamic power profiles are generated using a switching model proposed in the literature [77]. Three different power profiles were used to simulate different input patterns during behavioral synthesis and the average runtime for a power profile was reported. To permit a fair comparison among different techniques, we set the parameters for each technique to maximize performance while constraining the peak temperature error over all benchmarks and all time to 0.1%.

Table 5.1 presents the experimental results. Each row shows the results for a specific benchmark. Columns two, four, and seven indicate the CPU time used by GARK4, ISAC, and FATA. Columns six and nine indicate the speedups achieved by ISAC and FATA compared to GARK4 given the same error constraint. FATA speeds up analysis by 37.86–138.30$\times$ compared to GARK4 method and 118.59–222.60$\times$ compared to ISAC, while maintaining accuracy.

We also examined the performance of synchronous methods. We first compared the accuracy and performance of non-adaptive lock-step synchronous methods, namely forward Euler (FE), backward Euler (BE), trapezoidal (TR), and explicit RK4. Due to space limitations, we omit the full results. FE is inaccurate, with a peak error of 0.52% regardless of step size. FE and RK4 are impractical because they require manual specification of safe step size. FATA is 4.4–20.6$\times$ faster than all the non-adaptive synchronous methods. Adaptive implicit methods are prohibitive because they require matrix-vector solve at each time step.

### 5.5.2 Combining Temperature Update Functions with Step Adaptation Methods

This section analyzes the impact of temperature update function and step size adaptation method on asynchronous method performance. We use FE and the variant of TR describe in Section 5.4 as temperature update candidates and consider step doubling and third-order divided difference (DD3) as potential step size adaptation methods. We use the approach adopted by FATA, i.e., TR combined with DD3 as the base case.

Columns two, five, eight, and eleven of Table 5.2 indicate the runtime for each combination, while columns four, seven, and ten show the slowdown using the corresponding combination compared to the base case, i.e., TR combined with DD3. Note that the runtime for TR combined with DD3 is slightly larger than FATA because quiescence detection is not used. In comparison, DD3 is generally better than step doubling, resulting in a speedup of

2.61–6.5$\times$ for TR and a speedup of 1.16–1.74$\times$ for FE. The temperature update function is critical and TR is consistently faster than FE. This explains why FATA is much faster than ISAC, which makes the inaccurate assumption that neighboring element temperatures do not change during the integration interval of the current element. This can cause temperature estimation errors that decrease step size and degrade performance. The combination of TR and DD3 based step size adaptation is the best among all candidates. Although a higher-order temperature update function might further improve accuracy, this would impose additional computational overhead. In addition, high-order numerical methods are generally more likely to cause numerical instability problems in asynchronous methods. We experimented with a third-order temperature update function. Experimental results indicate the increase in the computational cost outweighs the improvement in accuracy: their use is not recommended.

## 5.6 CONCLUSIONS

This chapter proves that step doubling based globally adaptive fourth-order Runge-Kutta and Runge-Kutta-Fehlberg methods will fail to appropriately adapt step sizes regardless of initial power profile, thermal profile, and user-specified error threshold, i.e., the steady-state step size will converge to a constant value even when the thermal profile has converged, leading to poor performance. We proposed FATA, an asynchronous time marching thermal analysis technique that corrects this problem and uses other ideas to improve speed and accuracy. Experimental results indicate that FATA speeds up dynamic thermal analysis by 37.86–138.30$\times$ compared to an existing synchronous globally-adaptive fourth-order Runge-Kutta method and by 118.59–222.60$\times$ relative to an existing asynchronous adaptive dynamic thermal analysis technique, while maintaining accuracy. FATA will replace the time-domain solver used in ISAC; a new version will be released after integration [14]. Note that our findings do not imply that Runge-Kutta based thermal analysis methods are inaccurate, only that they may be inefficient due to inappropriately adapting step sizes to problem conditions.

# CHAPTER 6
# Optimization Technique 1: A High-Performance Microprocessor Cache Compression Algorithm

In Section 2.2, we explore the memory hierarchies of different system architectures such as single-core systems and CMP systems and argue that the memory hierarchy can significantly affect system performance. This claim is supported by the findings in Chapter 3. As illustrated in Chapter 3, accurate performance estimation is impossible without considering the cache access patterns of cache-sharing processes. In particular, CAMP, the cache contention aware performance model proposed in Chapter 3, estimates the performance degradations due to cache contention based on last-level cache access related information such as cache reuse distance histograms, cache access frequencies, and the relationship between the throughput and the cache miss rate of each cache-sharing process. This indicates that the last-level cache plays a significant role in determining system performance; optimization techniques applied to the last-level cache has the potential to significantly improve performance with little software or hardware overhead. However, such optimization techniques need to be carefully designed and evaluated to justify their design trade-offs.

This chapter presents an optimization technique that targets at the last-level on-chip cache in high-performance processors to addresses the increasingly important issue of controlling off-chip communication in computer systems in order to maintain good performance and energy efficiency. This technique is motivated by the continuously increasing processor–memory performance gap [56]: accessing off-chip memory generally takes an order of magnitude more time than accessing on-chip cache, and two orders of magnitude more time than executing an instruction. In addition, the microprocessor performance improves at a much faster speed than off-chip memory performance. The current transition from single-core processors to CMP processors further aggravates the problem: more processors require more memory accesses, but the performance of the processor–memory bus is not keeping pace. Therefore, the microprocessor designers have been torn between tight constraints on the amount of on-chip cache memory and the high latency of off-chip memory, such as dynamic random access memory.

Computer systems and microarchitecture researchers have proposed using hardware data compression units within the memory hierarchies of microprocessors in order to improve performance, energy efficiency, and functionality. However, most past work, and all work on cache compression, has made unsubstantiated assumptions about the performance, power consumption, and area overheads of the proposed compression algorithms and hardware. It is not possible to determine whether compressing the on-chip caches is beneficial without understanding its costs. Furthermore, as we show in this chapter, raw compression ratio is not always the most important metric.

In this chapter, we present a lossless compression algorithm that has been designed for fast on-line data compression, and cache compression in particular. The algorithm has a number of novel features tailored for this application, including combining pairs of compressed lines into one cache line and allowing parallel compression of multiple words while using a single dictionary and without degradation in compression ratio. We reduced the proposed algorithm to a register transfer level hardware implementation, permitting performance, power consumption, and area estimation. Experiments comparing our work to previous work are described.

The rest of this chapter is organized as follows. Section 6.1 describes the cache compression problem and presents the design challenges. Section 6.2 summarizes related work and our contributions. Section 6.4 briefly describes the C-Pack compression algorithm and several design trade-offs that permit efficient hardware compression. We also introduce the concept of effective system-wide compression ratio and discuss pair-matching based compressed line organization. Section 6.5 describes the proposed hardware implementation of C-Pack. Section 6.6 presents the evaluation results of the C-Pack hardware. We also discuss the implications of our findings regarding using C-Pack based cache compression algorithm in high-performance systems. Finally, Section 6.7 concludes this chapter.

## 6.1 INTRODUCTION

Microprocessor speeds have been increasing faster than off-chip memory latency, raising a "wall" between processor and memory. The ongoing move to CMPs is further increasing the problem; more processors require more accesses to memory, but the performance of the processor–memory bus is not keeping pace. Techniques that reduce off-chip communication without degrading performance have the potential to solve this problem. *Cache compression* is one such technique; data in last-level on-chip caches, e.g., L2 caches, are compressed, resulting in larger usable caches. In the past, researchers have reported that cache compression can improve the performance of uniprocessors by up to 17% for memory-intensive commercial workloads [11] and up to 225% for memory-intensive scientific workloads [42]. Researchers have also found that cache compression and prefetching techniques can improve CMP throughput by 10%–51% [12]. However, past work did not demonstrate whether the proposed compression/decompression hardware is appropriate for cache compression, considering the performance, area, and power consumption requirements. This analysis is also essential to permit the performance impact of using cache compression to be estimated.

Cache compression presents several challenges. First, decompression and compression must be extremely fast: a significant increase in cache hit latency will overwhelm the advantages of reduced cache miss rate. This requires an efficient on-chip decompression hardware implementation. Second, the hardware should occupy little area compared to the corresponding decrease in the physical size of the cache, and should not substantially increase the total chip power consumption. Third, the algorithm should losslessly compress small blocks, e.g., 64-byte cache lines, while maintaining a good compression ratio (throughout this chapter we use the term *compression ratio* to denote the ratio of the compressed data size over the original data size). Conventional compression algorithm quality metrics, such as block compression ratio, are not appropriate for judging quality in this domain. Instead, one must consider the effective system-wide compression ratio (defined precisely in Section 6.4.3). This chapter will point out a number of other relevant quality metrics for cache compression algorithms, some of which are new. Finally, cache compression should not increase power consumption substantially. The above requirements prevent the use of high-overhead compression algorithms such as the PPM family of algorithms [72] or Burrows-Wheeler transforms [23]. A faster and lower-overhead technique is required.

## 6.2 Related Work and Contributions

Researchers have commonly made assumptions about the implications of using existing compression algorithms for cache compression and the design of special-purpose cache compression hardware.

A number of researchers have assumed the use of general-purpose main memory compression hardware for cache compression. IBM's MXT (Memory Expansion Technology) [105] is a hardware memory compression/decompression technique that improves the performance of servers via increasing the usable size of off-chip main memory. Data are compressed in main memory and decompressed when moved from main memory to the off-chip shared L3 cache. Memory management hardware dynamically allocates storage in small sectors to accommodate storing variable-size compressed data block without the need for garbage collection. IBM reports compression ratios (compressed size divided by uncompressed size) ranging from 16% to 50%. X-Match is a dictionary-based compression algorithm that has been implemented on an FPGA [74]. It matches 32-bit words using a content addressable memory that allows partial matching with dictionary entries and outputs variable-size encoded data that depends on the type of match. To improve coding efficiency, it also uses a move-to-front coding strategy and represents smaller indices with fewer bits. Although appropriate for compressing main memory, such hardware usually has a very large block size (1 KB for MXT and up to 32 KB for X-Match), which is inappropriate for compressing cache lines. It is shown that for X-Match and two variants of Lempel-Ziv algorithm, i.e., LZ1 and LZ2, the compression ratio for memory data deteriorates as the block size becomes smaller [74]. For example, when the block size decreases from 1 KB to 256 B, the compression ratio for LZ1 and X-Match increase by 11% and 3%. It can be inferred that the amount of increase in compression ratio could be even larger when the block size decreases from 256 B to 64 B. In addition, such hardware has performance, area, or power consumption costs that contradict its use in cache compression. For example, if the MXT hardware were scaled to a 65 nm fabrication process and integrated within a 1 GHz processor, the decompression latency would be 16 processor cycles, about twice the normal L2 cache hit latency.

Other work proposes special-purpose cache compression hardware and evaluates only the compression ratio, disregarding other important criteria such as area and power consumption costs. Frequent pattern compression (FPC) [9] compresses cache lines at the L2 level by storing common word patterns in a compressed format. Patterns are differentiated by a 3-bit prefix. Cache lines are compressed to predetermined sizes that never exceed their original size to reduce decompression overhead. Based on logical effort analysis [102], for a 64-byte cache line, compression can be completed in three cycles and decompression in five cycles, assuming 12 fan-out-four (FO4) gate delays per cycle. To the best of our knowledge, there is no register-transfer-level hardware implementation or FPGA implementation of FPC, and therefore its exact performance, power consumption, and area overheads are unknown. Although the area cost for FPC [9] is not discussed, our analysis shows that FPC would have an area overhead of at least 290 K gates, almost eight times the area of the approach proposed in this chapter, to achieve the claimed 5-cycle decompression latency. This will be examined in detail in Section 6.6.3.3.

In short, assuming desirable cache compression hardware with adequate performance and low area and power overheads is common in cache compression research [64, 60, 42, 116, 69, 35, 78]. It is also understandable, as the microarchitecture community is more interested in microarchitectural applications than compression. However, without a cache compression algorithm and hardware implementation designed and evaluated for effective system-wide compression ratio, hardware overheads, and interaction with other portions of the cache compression system, one can not reliably determine whether the proposed architectural schemes are beneficial.

In this work, we propose and develop a lossless compression algorithm, named C-Pack, for on-chip cache compression. The main contributions of our work follow:

1. C-Pack targets on-chip cache compression. It permits a good compression ratio even when used on small cache lines. The performance, area, and power consumption overheads are low enough for practical use. This contrasts with other schemes such as X-match which require complicated hardware to achieve an equivalent effective system-wide compression ratio [74].

2. We are the first to fully design, optimize, and report performance and power consumption of a cache compression algorithm when implemented using a design flow appropriate for on-chip integration with a microprocessor. Prior work in cache compression does not adequately evaluate the overheads imposed by the assumed cache compression algorithms.

3. We demonstrate when line compression ratio reaches 50%, further improving it has little impact on effective system-wide compression ratio.

4. C-Pack is twice as fast as the best existing hardware implementations potentially

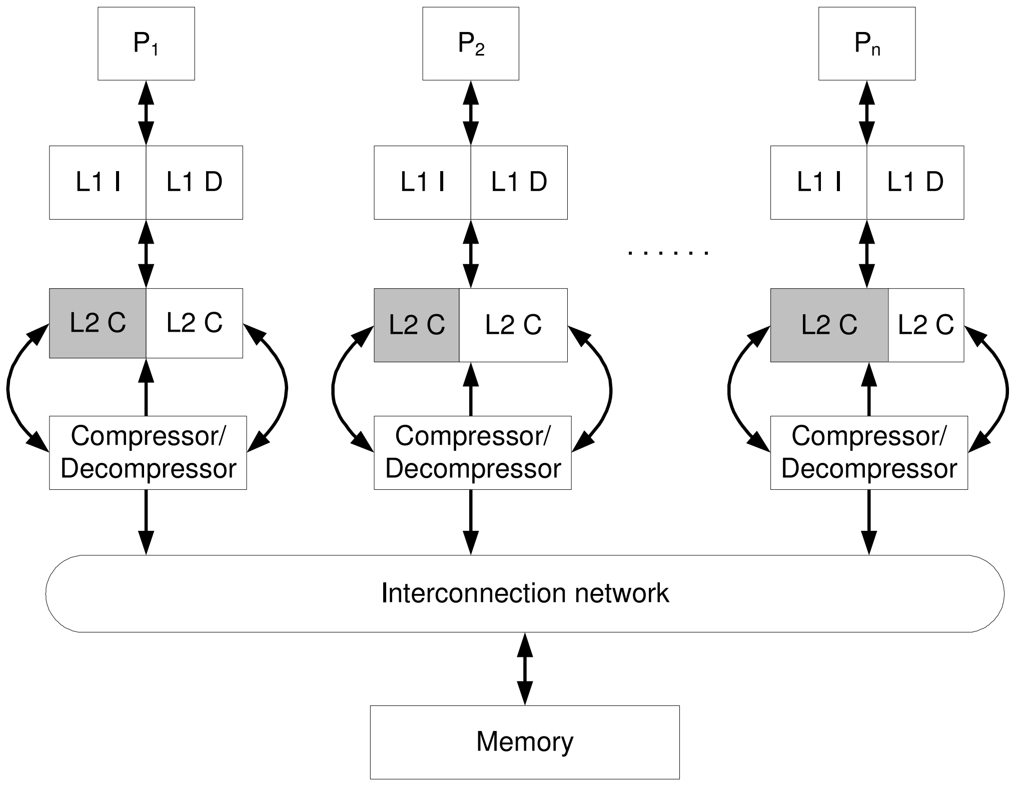


Figure 6.1: System architecture in which cache compression is used.

suitable for cache compression. For FPC to match this performance, it would require at least 8$\times$ the area of C-Pack.

5. We address the challenges in design of high-performance cache compression hardware while maintaining some generality, i.e., our hardware can be easily adapted to other high-performance lossless compression applications.

## 6.3 Cache Compression Architecture

In this section, we describe the architecture of a CMP system in which the cache compression technique is used. We consider private on-chip L2 caches, because in contrast to a shared L2 cache, the design styles of private L2 caches remain consistent when the number of processor cores increases. We also examine how to integrate data prefetching techniques into the system.

Figure 6.1 gives an overview of a CMP system with $n$ processor cores. Each processor has private L1 and L2 caches. The L2 cache is divided into two regions: an uncompressed region (L2 in the figure) and a compressed region (L2C in the figure). For each processor, the sizes of the uncompressed region and compression region can be determined statically or adjusted to the processor's needs dynamically. In extreme cases, the whole L2 cache is compressed due to capacity requirements, or uncompressed to minimize access latency. We assume a three-level cache hierarchy consisting of L1 cache, uncompressed L2 region, and compressed L2 region. The L1 cache communicates with the uncompressed region of the L2 cache, which in turn exchanges data with the compressed region through the compressor and decompressor, i.e., an uncompressed line can be compressed in the compressor and placed in the compressed region, and vice versa. Compressed L2 is essentially a virtual layer in the memory hierarchy with larger size, but higher access latency, than uncompressed L2. Note that no architectural changes are needed to use the proposed techniques for a shared L2 cache. The only difference is that both regions contain cache lines from different processors instead of a single processor, as is the case in a private L2 cache.

## 6.4 C-Pack Compression Algorithm

This section gives an overview of the proposed C-Pack compression algorithm. We first briefly describe the algorithm and several important features that permit an efficient hardware implementation, many of which would be contradicted for a software implementation. We also discuss the design trade-offs and validate the effectiveness of C-Pack in a compressed-cache architecture.

Table 6.1: Pattern Encoding For C-Pack

| Code | Pattern | Output | Length (b) | Freq. (%) |
|---|---|---|---|---|
| 00 | zzzz | (00) | 2 | 39.7 |
| 01 | x | (01)BBBB | 34 | 32.1 |
| 10 | mmmm | (10)bbbb | 6 | 7.6 |
| 1100 | mmxx | (1100)bbbbBB | 24 | 6.1 |
| 1101 | zzzx | (1100)B | 12 | 7.3 |
| 1110 | mmmx | (1110)bbbbB | 16 | 7.2 |

### 6.4.1 Design Constraints and Challenges

We first point out several design constraints and challenges particular to the cache compression problem:

1. Cache compression requires hardware that can de/compress a word in only a few CPU clock cycles. This rules out software implementations and has great influence on compression algorithm design.

2. Cache compression must be lossless to maintain correct microprocessor operation.

3. The block size for cache compression applications is smaller than for other compression applications such as file and main memory compression. Therefore, achieving a low compression ratio is challenging.

4. The complexity of managing the locations of cache lines after compression influences feasibility. Allowing arbitrary, i.e., bit-aligned, locations would complicate cache design to the point of infeasibility. A scheme that permits a pair of compressed lines to fit within an uncompressed line is advantageous.

### 6.4.2 C-Pack Algorithm Overview

C-Pack (for Cache Packer) is a lossless compression algorithm designed specifically for high-performance hardware-based on-chip cache compression. It achieves a good com-

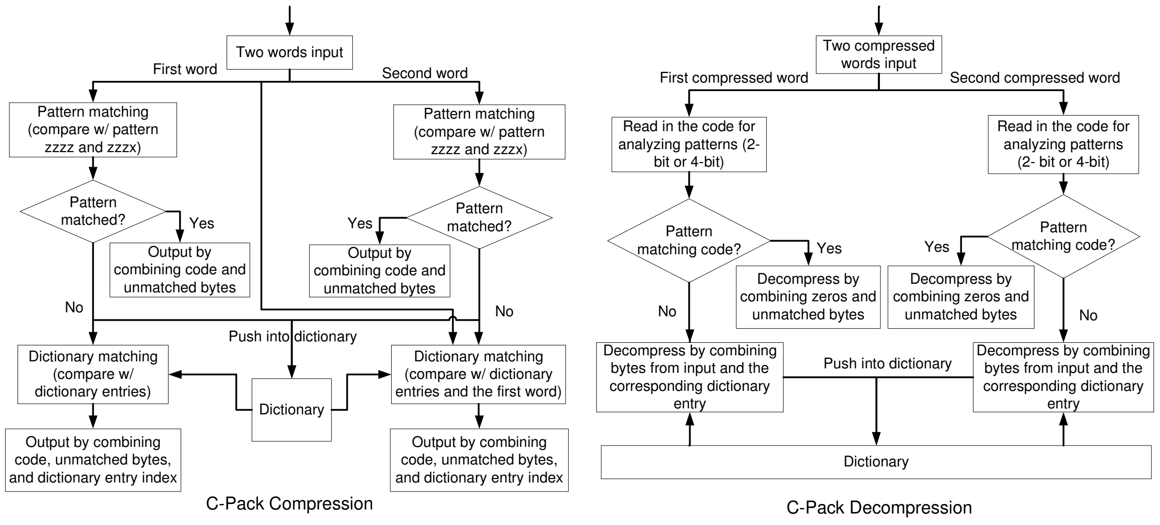

Two words input
First word
Second word
Pattern matching (compare w/ pattern zzzz and zzzx)
Pattern matched?
Yes
No
Output by combining code and unmatched bytes
Push into dictionary
Dictionary matching (compare w/ dictionary entries)
Dictionary
Dictionary matching (compare w/ dictionary entries and the first word)
Output by combining code, unmatched bytes, and dictionary entry index
C-Pack Compression
Two compressed words input
First compressed word
Second compressed word
Read in the code for analyzing patterns (2-bit or 4-bit)
Pattern matching code?
Decompress by combining zeros and unmatched bytes
Decompress by combining bytes from input and the corresponding dictionary entry
Dictionary
C-Pack Decompression

Figure 6.2: C-Pack (a) compression and (b) decompression.

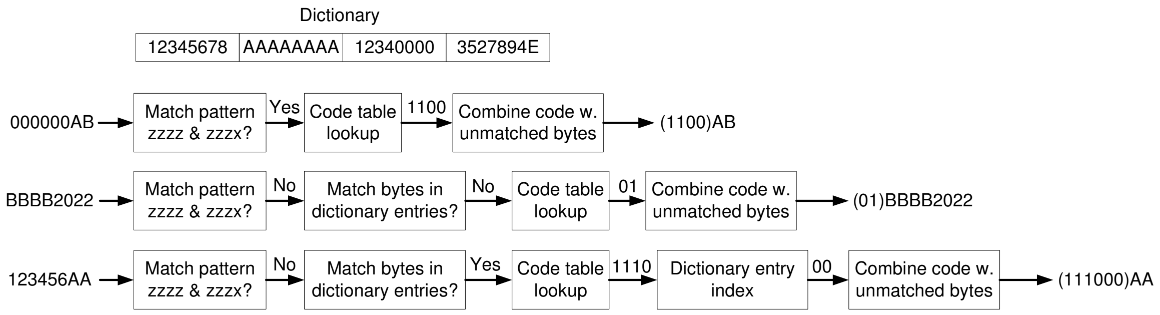


Figure 6.3: Compression examples for different input words.

pression ratio when used to compress data commonly found in microprocessor low-level on-chip caches, e.g., L2 caches. Its design was strongly influenced by prior work on pattern-based partial dictionary match compression [41]. However, this prior work was designed for software-based main memory compression and did not consider hardware implementation.

C-Pack achieves compression by two means: (1) it uses statically decided, compact encodings for frequently appearing data words and (2) it encodes using a dynamically updated dictionary allowing adaptation to other frequently appearing words. The dictionary supports partial word matching as well as full word matching. The patterns and coding schemes used by C-Pack are summarized in Table 6.1, which also reports the actual frequency of each pattern observed in the cache trace data file mentioned in Section 6.4.4. The 'Pattern' column describes frequently appearing patterns, where 'z' represents a zero byte, 'm' represents a byte matched against a dictionary entry, and 'x' represents an unmatched byte. In the 'Output' column, 'B' represents a byte and 'b' represents a bit.

The C-Pack compression and decompression algorithms are illustrated in Figure 6.2. We use an input of two words per cycle as an example in Figure 6.2. However, the algorithm can be easily extended to cases with one, or more than two, words per cycle. During one iteration, each word is first compared with patterns "zzzz" and "zzzx". If there is a match, the compression output is produced by combining the corresponding code and unmatched bytes as indicated in Table 6.1. Otherwise, the compressor compares the word with all dictionary entries and finds the one with the most matched bytes. The compression result is then obtained by combining code, dictionary entry index, and unmatched bytes, if any. Words that fail pattern matching are pushed into the dictionary. Figure 6.3 shows the compression results for several different input words. In each output, the code and the dictionary index, if any, are enclosed in parentheses. Although we used a 4-word dictionary in Figure 6.3 for illustration, the dictionary size is set to 64 B in our implementation. Note that the dictionary is updated after each word insertion, which is not shown in Figure 6.3.

During decompression, the decompressor first reads compressed words and extracts the codes for analyzing the patterns of each word, which are then compared against the codes defined in Table 6.1. If the code indicates a pattern match, the original word is recovered by combining zeroes and unmatched bytes, if any. Otherwise, the decompression output is given by combining bytes from the input word with bytes from dictionary entries, if the code indicates a dictionary match.

The C-Pack algorithm is designed specifically for hardware implementation. It takes advantage of simultaneous comparison of an input word with multiple potential patterns and dictionary entries. This allows rapid execution with good compression ratio in a hardware implementation, but may not be suitable for a software implementation. In general, software must process operations sequentially. For example, matching against multiple patterns can be prohibitively expensive for software implementations when the number of patterns or dictionary entries is large. C-Pack's inherently parallel design allows an efficient hardware implementation, in which pattern matching, dictionary matching, and processing multiple words are all done simultaneously. In addition, we chose various design parameters such as dictionary replacement policy and coding scheme to reduce hardware complexity, even if our choices slightly degrades the effective system-wide compression ratio. Details are described in Section 6.4.4.

In the proposed implementation of C-Pack, two words are processed in parallel per cycle. Achieving this, while still permitting an accurate dictionary match for the second word, is challenging. Let us consider compressing two similar words that have not been encountered by the compression algorithm recently, assuming the dictionary uses first-in first-out (FIFO) as its replacement policy. The appropriate dictionary content when processing the second word depends on whether the first word matched a static pattern. If so, the first word will not appear in the dictionary. Otherwise, it will be in the dictionary, and its presence can be used to encode the second word. Therefore, the second word should be compared with the first word and all but the first dictionary entry in parallel. This improves compression ratio compared to the more naïve approach of not checking with the first word. Therefore, we can compress two words in parallel without compression

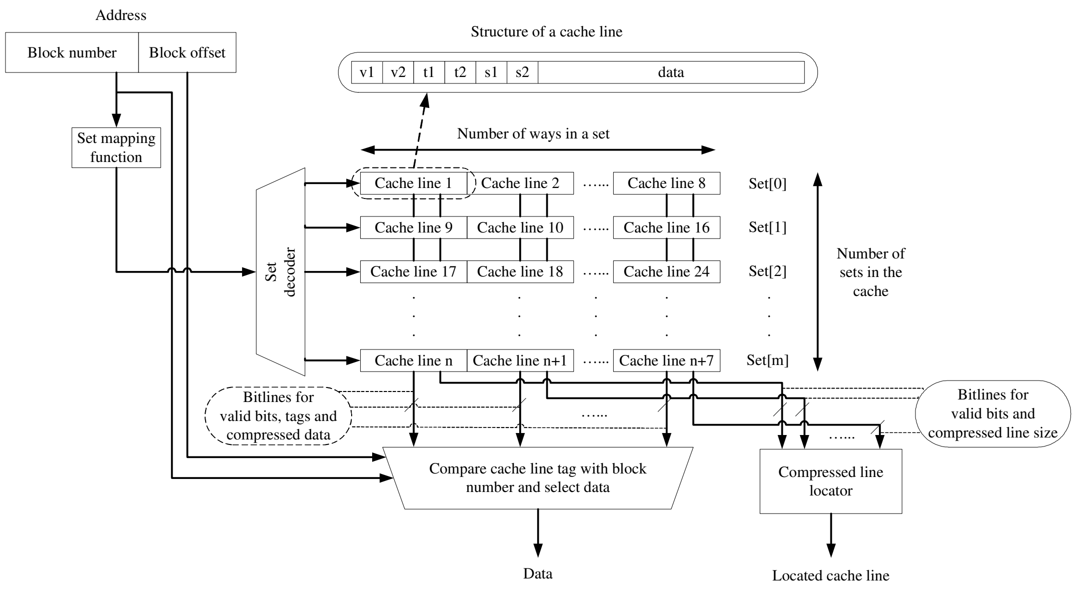


Figure 6.4: Structure of a pair-matching based cache.

ratio degradation.

### 6.4.3 Effective System-Wide Compression Ratio and Pair Matching Compressed Line Organization

Compressed cache organization is a difficult task because different compressed cache lines may have different lengths. Researchers have proposed numerous variants of line segmentation techniques [42, 64, 11] to handle this problem. The main idea is to divide compressed cache lines into fixed-size segments and use indirect indexing to locate all the segments for a compressed line. Hallnor et al. [42] proposed IIC-C, i.e., indirect index cache with compression. The proposed cache design decouples accesses across the whole cache, thus allowing a fully-associative placement. Each tag contains multiple pointers to smaller fixed-size data blocks to represent a single cache block. However, the tag storage overhead of IIC-C is significant, e.g., 21% given a 64 B line size and 512 KB cache size, compared to less than 8% for our proposed pair-matching based cache organization. In addition, the hardware overhead for addressing a compressed line is not discussed in the paper. The access latency in IIC-C is attributed to three primary sources, namely additional hit latency due to sequential tag and data array access, tag lookup induced additional hit and miss latency, and additional miss latency due to the overhead of software management. However, the authors do not report worst-case latency. Lee et al. [64] proposed selective compressed caches using a similar idea. Only the cache lines with a compression ratio of less than 0.5 are compressed so that two compressed cache lines can fit in the space required for one uncompressed cache line. However, this will inevitably result in a larger system-wide compression ratio compared to that of pair-matching based cache because each compression ratio, not the average, must be less than 0.5, for compression to occur. The hardware overhead and worst-case access latency for addressing a compressed cache line is not discussed. Alameldeen et al. [11] proposed decoupled variable-segment cache, where the L2 cache dynamically allocates compressed or uncompressed lines depending on whether compression eliminates a miss or incurs an unnecessary decompression overhead. However, this approach has significant performance and area overhead, discussed later in this section.

We propose the idea of *pair-matching* to organize compressed cache lines. In a pair-matching based cache, the location of a newly compressed line depends on not only its own compression ratio but also the compression ratio of its "partner". More specifically, the compressed line locator first tries to locate the cache line (within the set) with sufficient unused space for the compressed line without replacing any existing compressed lines. If no such line exists, one or two compressed lines are evicted to store the new line. A compressed line can be placed in the same line with a partner only if the sum of their compression ratios is less than 100%. Note that successful placement of a line does not require that it have a compression ratio smaller than 50%. It is only necessary that the line, combined with a “partner line” be as small as an uncompressed line. To reduce hardware complexity, the candidate partner lines are only selected from the same set of the cache. Compared to segmentation techniques which allow arbitrary positions, pair-matching simplifies designing hardware to manage the locations of the compressed lines. More specifically, line extraction in a pair-matching based cache only requires parallel address tag match and takes a single cycle to accomplish. For line insertion, neither LRU list search nor set compaction is involved.

Figure 6.4 illustrates the structure of an 8-way associative pair-matching based cache. Since any line may store two compressed lines, each line has two *valid* bits and *tag* fields to indicate status and indexing. When compressed, two lines share a common *data* field. There are two additional *size* fields to indicate the compressed sizes of the two lines. Whether a line is compressed or not is indicated by its *size* field. A *size* of zero is used to indicate uncompressed lines. For compressed lines, *size* is set to the line size for an empty line, and the actual compressed size for a valid line. For a 64-byte line in a 32-bit architecture the tag is no longer than 32 bits, hence the worst-case overhead is less than 32 (tag) + 1 (valid) + 2 $\times$ 7 (size) bits, i.e., 6 bytes.

As we can see in Figure 6.4, the compressed line locator uses the bitlines for valid bits and compressed line sizes to locate a newly compressed line. Note that only one compressed line locator is required for the entire compressed cache. This is because for a given address, only the cache lines in the set which the specific address is mapped to are activated thanks to the set decoder. Each bitline is connected to a sense amplifier, which usually requires several gates [84], for signal amplification and delay reduction. The total area overhead is approximately 500 gates plus the area for the additional bitlines, compared to an uncompressed cache.

Based on the "pair-matching" concept, a newly compressed line has an effective compression ratio of 100% when it takes up a whole cache line, and an effective compression ratio of 50% when it is placed with a partner in the same cache line. Note that when a compressed line is placed together with its partner without evicting any compressed lines, its partner's effective compression ratio decreases to 50%. The *effective system-wide compression ratio* is defined as the average of the effective compression ratios of all cache lines in a compressed cache. It indicates how well a compression algorithm performs for pair-matching based cache compression. The concept of effective compression ratio can also be adapted to a segmentation based approach. For example, for a cache line with 4 fixed-length segments, a compressed line has an effective compression ratio of 25% when it takes up one segment, 50% for two segments, and so on. Varying raw compression ratio between 25% and 50% has little impact on the effective cache capacity of a four-part segmentation based technique. Figure 6.5 illustrates the distribution of raw compression ratios for different cache lines derived from real cache data. The $x$-axis shows different compression ratio intervals and $y$-axis indicates the percentage of all cache lines in each compression ratio interval. For real cache trace data, pair-matching generally achieves a better effective system-wide compression ratio (58%) than line segmentation with four segments per line (62%) and the same compression ratio as line segmentation with eight segments, which would impose substantial hardware overhead.

We now compare the performance and hardware overhead of pair-matching based cache with decoupled variable-segment cache. The hardware overhead can be divided into two parts: tag storage overhead and compressed line locator overhead. For a 512 KB L2 cache with a line size of 64 bytes, the tag storage overhead is 7.81% of the total cache size for both decoupled variable-segment cache and pair-matching based cache. The area overhead of the compressed line locator is significant in a decoupled variable-segment cache. During line insertion, a newly inserted line may be larger than the LRU line plus the unused segments. In that case, prior work proposed replacing two lines by replacing the LRU line and searching the LRU list to find the least-recently used line that ensures enough space for the newly arrived line [11]. However, maintaining and updating the LRU list will result in great area overhead. Moreover, set compaction may be required after line insertion to maintain the contiguous storage invariant. This can be prohibitively expensive in terms of area cost because it may require reading and writing all the set's data segments. Cache compression techniques that assume it are essentially proposing to implement kernel memory allocation and compaction in hardware [21]. However, for pair-matching based cache, the area of compressed line locator is negligible (less than 0.01% of the total cache size).

The performance overhead comes from two primary sources: addressing a compressed line and compressed line insertion. The worst-case latency to address a compressed line in a pair-matching based cache is 1 cycle. For a 4-way associative decoupled variable-segment cache with 8 segments per line, each set contains 8 compression information tags and 8 address tags because each set is constrained to hold no more than eight compressed lines. The compression information tag indicates (1) whether the line is compressed and (2) the compressed size of the line. Data segments are stored contiguously in address tag order. In order to extract a compressed line from a set, eight segment offsets are computed in parallel

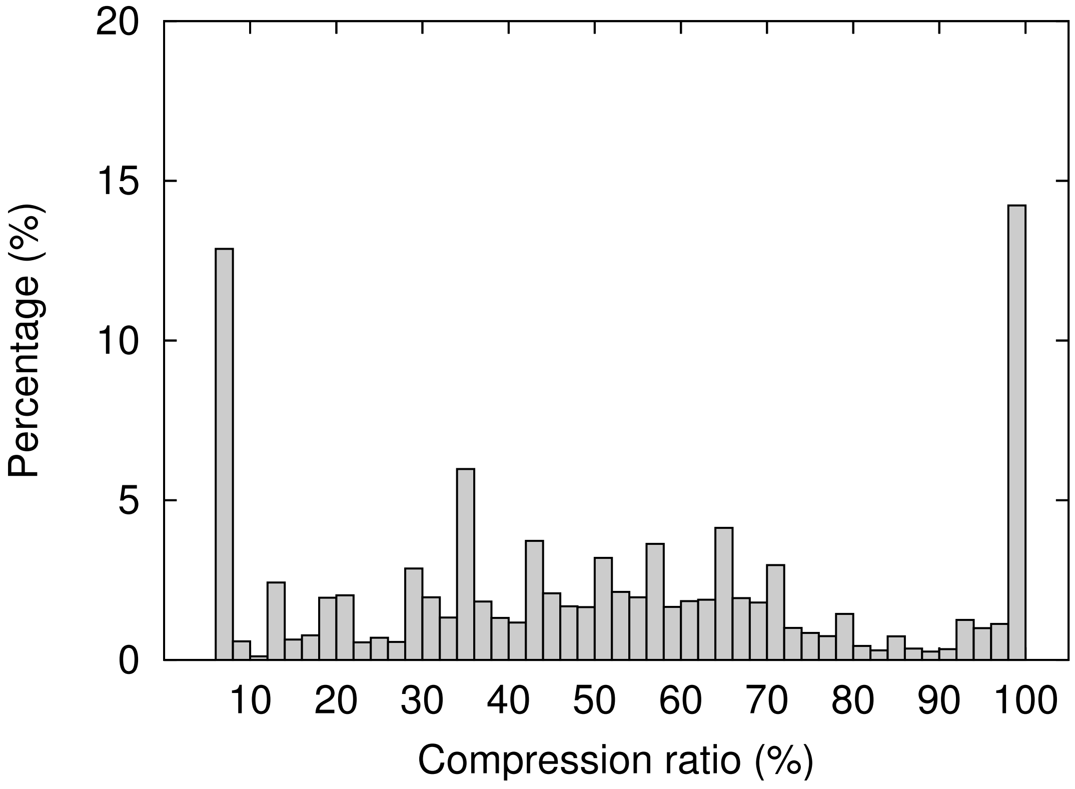


Figure 6.5: Distribution of compression ratios.

with the address tag match. Therefore, deriving the segment offset for the last line in the set requires summing up all the previous 7 compressed sizes, which incurs a significant performance overhead. In addition, although the cache array may be split into two banks to reduce line extraction latency, addressing the whole compressed line may still take 4 cycles in the worst case. To insert a compressed line, the worst-case latency is 2 cycles for pair-matching based cache with a peak frequency of more than 1 GHz. The latency of a decoupled variable-segment cache is not reported [11]. However, as explained in the previous paragraph, LRU list searching and set compaction introduce great performance overhead. Therefore, we recommend pair-matching and use the pair-matching effective system-wide compression ratio as a metric for comparing different compression algorithms.

Table 6.2: Effective System-Wide Compression Ratios For C-Pack

| Effective system-wide compression ratio (%) | | | | | | | |
|---|---|---|---|---|---|---|---|
| Dictionary size (B) | | 16 | 32 | 64 | 128 | 256 | 512 |
| FIFO | Huffman | 58.14 | 57.56 | 57.46 | 57.46 | 57.66 | 57.73 |
| | Two-level | 58.81 | 58.47 | 57.95 | 58.30 | 58.29 | 58.68 |
| LRU | Huffman | 58.13 | 57.70 | 57.61 | 57.91 | 58.07 | 58.17 |
| | Two-level | 58.97 | 58.54 | 58.38 | 58.73 | 58.72 | 58.92 |
| Two FIFOs | Huffman | 58.05 | 57.61 | 57.48 | 57.46 | 57.66 | 57.73 |
| | Two-level | 58.71 | 58.49 | 57.97 | 58.30 | 58.29 | 58.68 |
| RLE w/ LRU | Huffman | 57.20 | 56.68 | 56.63 | 56.73 | 56.75 | 56.87 |
| | Three-level | 57.66 | 57.31 | 57.08 | 57.11 | 57.35 | 57.44 |

### 6.4.4 Design Tradeoffs and Details

In this section, we present several design tradeoffs encountered during the design and implementation of C-Pack. We also validate C-Pack's effectiveness in pair-matching.

**Dictionary design and pattern coding**:

We evaluated the impact of different parameters during algorithm and hardware design and optimization, including dictionary replacement policy, dictionary size, and pattern coding scheme. The effective system-wide compression ratio of C-Pack was evaluated based on cache trace data collected from a full microprocessor, operating system, and application simulation of various workloads, e.g., media applications and SPEC CPU2000 benchmarks on a simulated 1 GHz processor. The cache is set to 8-way associative with a 64 B line size. The evaluation results are shown in Table 6.2. The candidates for different parameters and the final selected values are shown in Table 6.3, in which the first column shows various parameters, the second column shows the corresponding candidates for each parameter, and the third column shows the selected values. Note that the two or three level coding scheme in Table 6.3 refers to one in which the code length is fixed within the same level, but differs from level to level. For example, a two-level code can contain 2-bit and 4-bit codes only.

The criteria of choosing design parameters can be explained as follows. For design parameters that have only have small impact on design complexity, we choose values to optimize compression ratio. For design parameters that have great impact on design complexity, we choose the simplest design when the design complexity varies a lot as the current design parameter changes. For replacement policy, hardware LRU algorithm maintains a table of $n \times n$ bits given a dictionary size of $n$ words. The $k$th row and $k$th column is updated on every access to word $k$, which complicates hardware design. Using 2 FIFO queues to simulate LRU essentially doubles the dictionary size. In addition, moving the words between two queues adds additional hardware complexity. Combining FIFO with RLE also complicates hardware design because special care is needed when the current word has been encountered several times before. For all of the replacement policies except the simplest FIFO, dictionary updates depend not only on the current word but also on recently processed words. This complicates algorithm design because decompression of the second word depends on the dictionary updates due to decompression of the first word. A replacement policy that records the exact sequence of recently processed words would incur a large area overhead during decompression hardware design. This is also true

Table 6.3: Design Choices For Different Parameters

| Parameters | Candidates | Selected Candidate |
|---|---|---|
| Dictionary replacement policy | (1) First-in first out (FIFO)<br>(2) Least recently used (LRU)<br>(3) Using FIFO queues to simulate LRU<br>(4) FIFO with run-length encoding (RLE) | FIFO – least HW complexity<br>only 1.32% higher CR than best case |
| Coding scheme | (1) Huffman coding<br>(2) Two or Three-level coding | Two-level coding due to best HW complexity with at most 0.95% increase in CR given the same dictionary size and replacement policy |
| Dictionary size | Ranging from 16 B to 512 B | 64 B – optimal CR for FIFO and low HW cost |

when selecting a coding scheme because for Huffman coding, there is a larger variance in the length of a compressed word, thus making it more difficult to determine the location of the second compressed word. Therefore, choosing Huffman coding also negatively affects the decompression hardware design. However, varying the dictionary size only affects the area by a negligible amount (in the order of several hundred gates). Moreover, it has no impact on the structure of compression or decompression hardware. Therefore, we choose the dictionary size that minimizes the compression ratio. With the selected parameters, the effective system-wide compression ratio for a 64 byte cache line is 58.47% for our test data.

**Trade-Off Between Area and Decompression Latency**:

Decompression latency is a critically important metric for cache compression algorithms. During decompression, the processor may stall while waiting for important data. If the decompression latency is high, it can undermine potential performance improvements. It is possible to use increased parallelism to increase the throughput of a hardware implementation of C-Pack, at the cost of increased area. For example, decompressing the second word by combining bytes from the input and the dictionary is challenging because the locations of bytes that compose the decompression output depend on the codes of the first word and second word. Given that each code can have 6 possible values, as indicated in Table 6.1, there are 36 possible combinations of the two codes, each of which corresponds to a unique combination of bytes from the input and dictionary. If we double the number of words processed in one cycle, i.e., 4 words per cycle, there can be 1,296 possible combinations for decompressing the fourth word, thereby dramatically increasing the area cost. To achieve a balance between area and throughput, we decided to compress or decompress two words per cycle.

**Evaluating Compression Ratio for C-Pack Pair-Matching**:

In order to determine whether the mean and variance of the compression ratio achieved by C-Pack is sufficient for most lines to find partners, we simulated a "pair-matching" based cache using the cache trace data described above to compute the probability of two cache lines fitting within one uncompressed cache line. The simulated cache size ranges from 64 KB to 2 MB and the set associativity ranges from 4 to 8. We adopt a "best fit + best fit" policy: for a given compressed cache line, we first try to find the cache line with minimal but sufficient unused space. If the attempt fails, the compressed line replaces one or two compressed lines. This scheme is penalized only when two lines are evicted to store the new line. Experimental results indicate that the worst-case probability of requiring the eviction of two lines is 0.55%, i.e., the probability of fitting a compressed line into the cache without additional penalty is at least 99.45%. We implemented and synthesized this line replacement policy in hardware. The delay and area cost are reported in Table 6.4.

## 6.5 C-Pack Hardware Implementation

In this section, we provide a detailed description of the proposed hardware implementation of C-Pack. Note that although the proposed compressor and decompressor mainly target on-line cache compression, they can be used in other data compression applications, such as memory compression and network data compression, with few or no modifications.

### 6.5.1 Compression Hardware

This section describes the design and optimization of the proposed compression hardware. It first gives an overview of the proposed compressor architecture, and then discusses the data flow among different pipeline stages inside the compressor.

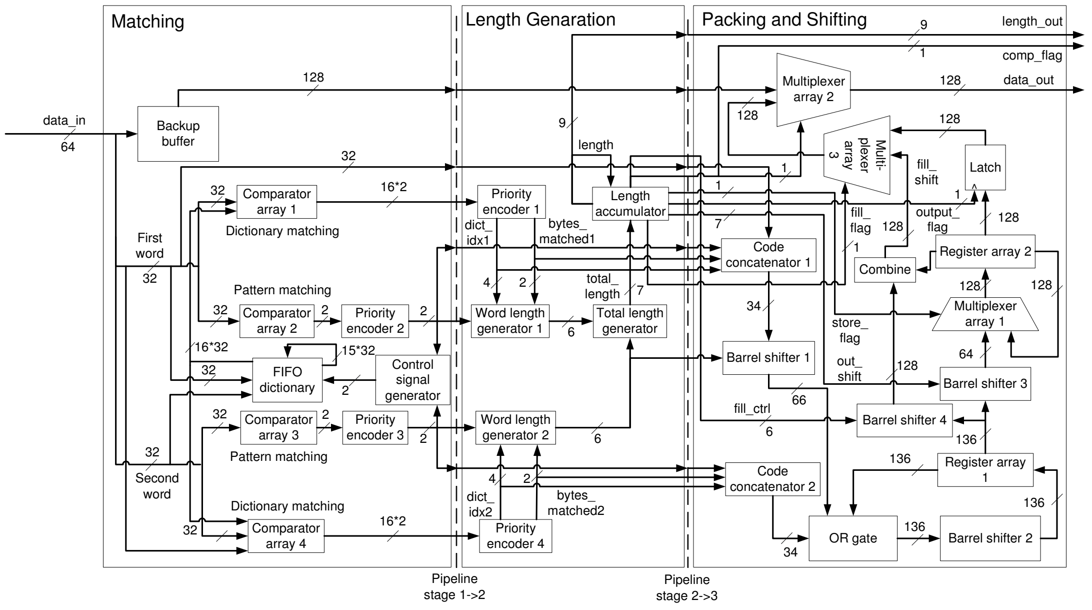


Figure 6.6: Compressor architecture.

**Compressor Architecture**:

Figure 6.6 illustrates the hardware compression process. The compressor is decomposed into three pipeline stages. This design supports incremental transmission, i.e., the compressed data can be transmitted before the whole data block has been compressed. This reduces compression latency. We use bold and italic fonts to represent the devices and signals appearing in figures.

1. **Pipeline Stage 1**: The first pipeline stage performs pattern matching and dictionary matching on two uncompressed words in parallel. As illustrated in Figure 6.6, **comparator array 1** matches the first word against pattern “zzzz” and “zzzx” and **comparator array 2** matches it with all the dictionary entries, both in parallel. The same is true for the second word. However, during dictionary matching, the second word is compared with the first word in addition to the dictionary entries. The pattern matching results are then encoded using **priority encoders 2 and 3**, which are used to

determine whether to push these two words into the **FIFO dictionary**. Note that the first word and the second word are processed simultaneously to increase throughput.

2. **Pipeline Stage 2**: This stage computes the total length of the two compressed words and generates control signals based on this length. Based on the dictionary matching results from Stage 1, **priority encoders 1 and 4** find the dictionary entries with the most matched bytes and their corresponding indices, which are then sent to **word length generators 1 and 2** to calculate the length of each compressed word. The **total length calculator** adds up the two lengths, represented by signal *total_length*. The **length accumulator** then adds the value of *total_length* to two internal signals, namely *sum_partial* and *sum_total*. *Sum_partial* records the number of compressed bits stored in **register array 1** that have not been transmitted. Whenever the updated *sum_partial* value is larger than 64 bits, *sum_partial* is decreased by 64 and signal *store_flag* is generated indicating that the 64 compressed bits in **register array 1** should be transferred to either the left half or the right half of the 128-bit **register array 2**, depending on the previous state of **register array 2**. It also generates signal *out_shift* specifying the number of bits **register array 1** should shift to align with **register array 2**. *Sum_total* represents the total number of compressed bits produced since the start of compression. Whenever *sum_total* exceeds the original cache line size, the compressor stops compressing and sends back the original cache line stored in the **backup buffer**.

3. **Pipeline Stage 3**: This stage generates the compression output by combining codes, bytes from input word, and bytes from dictionary entries depending on the pattern

and dictionary matching results from previous stages.

Placing the compressed pair of words into the right location within **register array 1**, denoted by $\text{Reg}_1$[135:0], is challenging. Since the length of a compressed word varies from word to word, it is impossible to pre-select the output location statically. In addition, **register array 1** should be shifted to fit in the compressed output in a single cycle without knowing the shift length in advance. We address this problem by analyzing the output length. Notice that a single compressed word can only have 7 possible output lengths, with the maximum length being 34 bits. Therefore, we use two 34-bit buffers, denoted by A[33:0] and B[33:0], to store the first and second compressed outputs generated by **code concatenators 1 and 2** in the lower bits, with the higher unused bits set to zero. $\text{Reg}_1$[135:0] is shifted by *total_length* using **barrel shifter 2**, with the shifting result denoted by $\text{Reg}_{1s}$[135:0]. At the same time, A[33:0] is shifted using **barrel shifter 1** by the output length of the second compressed word. The result of this shift is held by S[65:0], also with all higher (unused) bits set to zero. Note that $\text{Reg}_1$[135:68] only has one input source, i.e., $\text{Reg}_{1s}$[135:68], because the maximum total output length is 68. However, $\text{Reg}_1$[67:2] can have multiple input sources: B, S, and $\text{Reg}_{1s}$. For example, $\text{Reg}_1$[4] may come from B[4], S[2], or $\text{Reg}_{1s}$[0]. To obtain the input to $\text{Reg}_1$[135:0], we OR the possible inputs together because the unused bits in the input sources are all initialized to zero, which should not affect the result of an OR function.

Meanwhile, $\text{Reg}_1$[135:0] is shifted by *out_shift* using **barrel shifter 3** to align with **register array 2**, denoted by $\text{Reg}_2$[135:0]. **Multiplexer array 1** selects the shifting result as the input to Reg$_2$[135:0] when *store_flag* is 1 (i.e., the number of accumulated compressed bits has exceeded 64 bits) and the original content of Reg$_2$[135:0] otherwise. Whether **Latch** is enabled depends on the number of compressed bits accumulated in Reg$_2$[135:0] that have not been transmitted. When *output_flag* is 1, indicating that 128 compressed bits have been accumulated in Reg$_2$[135:0], Reg$_2$[135:0] is passed to **Multiplexer array 1**. **Multiplexer array 3** selects between *fill_shift* and the output of **latch** using *fill_flag*. *Fill_shift* represents the 128-bit signal that pads the remaining compressed bits that have not been transmitted with zeros and *fill_flag* determines whether to select the padded signal. **Multiplexer array 2** then decides the output data based on the total number of compressed bits. When the total number of compressed bits has exceeded the uncompressed line size, the contents of **backup buffer** are selected as the output. Otherwise, the output from **Multiplexer array 3** is selected.

### 6.5.2 Decompression Hardware

This section describes the design and optimization of the proposed decompression hardware. We describe the data flow inside the decompressor and point out some challenges specific to the decompressor design. We also examine how to integrate data prefetching with cache compression.

#### 6.5.2.1 Decompressor Architecture:

Figure 6.7 illustrates the decompressor architecture. Recall that the compressed line, which may be nearly 512 bits long, is processed in 128-bit blocks, the width of the bus used for L2 cache access. The use of a fixed-width bus and variable-width compressed words

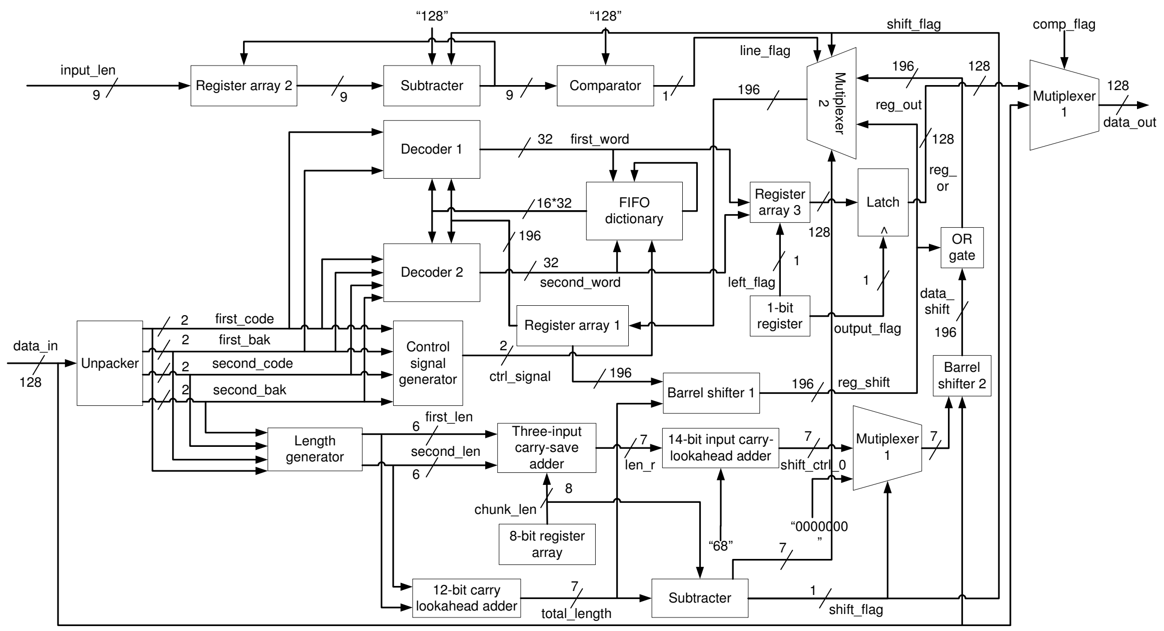


Figure 6.7: Decompressor architecture.

implies that a compressed word may sometimes span two 128-bit blocks. This complicates decompression. In our design, two words are decompressed per cycle until fewer than 68 bits remain in **register array 1** (68 bits is the maximum length of two compressed words). The decompressor then shifts in more compressed data using **barrel shifter 2** and concatenates them with the remaining compressed bits. In this way, the decompressor can always fetch two whole compressed words per cycle. The decompressor also supports incremental tranmission, i.e., the decompression results can be transmitted before the whole cache line is decompressed provided that there are 128 decompressed bits in **register array 3**. The average decompression latency is 5 cycles.

1. **Word Unpacking:** When decompression starts, the **unpacker** first extracts the two codes of the first and second word. Signals *first_code* and *second_code* represent the first two bits of the codes in the two compressed words. Signal *first_bak* and *second_bak* refer to the two bits following *first_code* and *second_code*, respectively.

They are mainly useful when the corresponding code is a 4-bit code.

2. **Word Decompressing: Decoders 1 and 2** compare the codes of the first and second word against the static codes in Table 6.1 to derive the patterns for the two words, which are then decompressed by combining zero bytes, bytes from **FIFO dictionary**, and bytes from **register array 1** (which stores the remaining compressed bits). The way the bytes are combined to produce the decompression results depends on the values of the four code-related signals. The decompressed words are then pushed into the **FIFO dictionary**, if they do not match pattern "zzzz" and "zzzx", and **register array 3**. Note that the decompression results will be transmitted out as soon as **register array 3** has accumulated four decompressed words, given the input line is a compressed line.

3. **Length Updating: Length generator** derives the compressed lengths of the two words, i.e., *first_len* and *second_len*, based on the four code-related signals. The two lengths are then subtracted from *chunk_len*, which denotes the number of the remaining bits to decompress in **register array 1**. As we explained above, the subtraction result *len_r* is then compared with 68, and more data are shifted in and concatenated with the remaining compressed bits in **register array 1** if *len_r* is less than 68. Meanwhile, **register array 1** is shifted by *total_length* (the sum of *first_len* and *second_len*) to make space for the new incoming compressed bits.

**6.5.2.2 Decompressor & Data Prefetching**: Data prefetching [107] has been proposed as a technique to hide data access latency. It anticipates future cache misses and fetches the associated data into the cache in advance of expected memory references. In or-

Table 6.4: Synopsys Design Compiler Synthesis Results

| Parameters | 180 nm | | | 90 nm | | | 65 nm | | |
|---|---|---|---|---|---|---|---|---|---|
| | Compressor | Decompressor | Loc. | Compressor | Decompressor | Loc. | Compressor | Decompressor | Loc. |
| Worst-case delay (cycles) | 13 | 8 | 2 | 13 | 8 | 2 | 13 | 8 | 2 |
| Max. frequency (GHz) | 0.38 | 0.31 | 0.60 | 1.09 | 0.91 | 1.79 | 1.25 | 1.20 | 2.00 |
| Area (mm$^2$) | 0.34 | 0.25 | 0.063 | 0.076 | 0.076 | 0.013 | 0.043 | 0.043 | 0.007 |
| Power consumption at max. internal freq. (mW) | 111.78 | 75.18 | 110.03 | 73.88 | 51.50 | 15.96 | 32.63 | 24.14 | 5.20 |

der to integrate data prefetching with cache compression, resource conflicts must be taken into account: a processor may request a line in the compressed region of the L2 cache while the corresponding decompressor prefetches data from the compressed region into the uncompressed region. Although we can disable data prefetching from the compressed region of an L2 cache, i.e., only allowing prefetching data from off-chip memory into the uncompressed region of L2, this may result in higher average data prefetching latency and lower performance benefit compared to a scheme where prefetching from both off-chip memory and compressed region of L2 caches are enabled. One possible solution is to add an extra decompressor for each processor. This enables simultaneously serving processor requests and prefetching data from the compressed region into the uncompressed region.

## 6.6 EVALUATION

In this section, we present the evaluation of the C-Pack hardware. We first present the performance, power consumption, and area overheads of the compression/decompression hardware when synthesized for integration within a microprocessor. Then, we compare the compression ratio and performance of C-Pack to other algorithms considered for cache compression: MXT [105], Xmatch [74], and FPC [10]. Finally, we describe the implications of our findings on the feasibility of using C-Pack based cache compression within a microprocessor.

### 6.6.1 C-Pack Synthesis Results

We synthesized our design using Synopsys Design Compiler with 180 nm, 90 nm, and 65 nm libraries. Table 6.4 presents the resulting performance, area, and power consumption at maximum internal frequency. "Loc" refers to the compressed line locator/arbitrator in a pair-matching compressed cache and "worst-case delay" refers to the number of cycles required to compress, decompress, or locate a 64 B line in the worst case. As indicated in Table 6.4, the proposed hardware design achieves a throughput of 80 Gb/s (64 B $\times$ 1.25 GHz) for compression and 76.8 Gb/s (64 B $\times$ 1.20 GHz) for decompression in a 65 nm technology. Its area and power consumption overheads are low enough for practical use. The total power consumption of the compressor, decompressor, and compressed line arbitrator at 1 GHz is 48.82 mW (32.63 mW/1.25 GHz + 24.14 mW/1.20 GHz + 5.20 mW/2.00 GHz) in a 65 nm technology. This is only 7% of the total power consumption of a 512 KB cache with a 64 B block size at 1 GHz in 65 nm technology, derived using CACTI 5 [1].

### 6.6.2 Comparison of Compression Ratio

We compare C-Pack to several other hardware compression designs, namely X-Match, FPC, and MXT, that may be considered for cache compression. We exclude other compression algorithms because they either have not been implemented in hardware or are not suitable for cache compression. Although the proposed hardware implementation mainly targets online cache compression, it can also be used in other high-performance lossless data compression applications with few or no changes.

We tested the compression ratios of different algorithms on four cache data traces gathered from a full system simulation of various workloads from the Mediabench [63] and

Table 6.5: Compression Ratio Comparison

| | Raw compression ratio (%) | | | | System-wide compression ratio (%) | | | |
|---|---|---|---|---|---|---|---|---|
| Benchmark | MXT | FPC | X-Match | C-Pack | MXT | FPC | X-Match | C-Pack |
| mpeg2 | 70.88 | 63.39 | 49.50 | 52.10 | 75.55 | 64.28 | 57.97 | 58.47 |
| mesa | 49.50 | 69.81 | 42.80 | 51.97 | 60.50 | 66.18 | 53.59 | 55.80 |
| art | 57.69 | 59.27 | 46.60 | 51.74 | 64.84 | 66.67 | 60.63 | 61.40 |
| twolf | 84.09 | 80.73 | 70.20 | 77.40 | 85.90 | 75.60 | 62.37 | 69.92 |
| Average | 65.54 | 68.30 | 52.28 | 58.30 | 71.70 | 68.18 | 58.64 | 61.40 |

SPEC CPU2000 benchmark suites. The block size and the dictionary size are both set to 64 B in all test cases. Since we are unable to determine the exact compression algorithm used in MXT, we used the LZSS Lempel-Ziv compression algorithm to approximate its compression ratio [40]. The raw compression ratios and effective system-wide compression ratios in a pair-matching scheme are summarized in Table 6.5. Each row shows the raw compression ratios and effective system-wide compression ratios using different compression algorithms for an application. As indicated in Table 6.5, raw compression ratio varies from algorithm to algorithm, with X-Match being the best and MXT being the worst on average. The poor raw compression ratios of MXT are mainly due to its limited dictionary size. The same trend is seen for effective system-wide compression ratios: X-Match has the lowest (best) and MXT has the highest (worst) effective system-wide compression ratio. Since the raw compression ratios of X-Match and C-Pack are close to 50%, they achieve better effective system-wide compression ratios than MXT and FPC. On average, C-Pack's system-wide compression ratio is 2.76% worse than that of X-Match, 6.78% better than that of FPC, and 10.3% better than that of MXT.

### 6.6.3 Comparison of Hardware Performance

This subsection compares the decompression latency, peak frequency, and area of C-Pack hardware to that of MXT, X-Match, and FPC. Power consumption comparisons are excluded because they are not reported for the alternative compression algorithms. Decompression latency is defined as the time to decompress a 64 B cache line.

**6.6.3.1 Comparing C-Pack with MXT** : MXT has been implemented in a memory controller chip operating at 133 MHz using 0.25 µm CMOS ASIC technology [106]. The decompression rate is 8 B/cycle with 4 decompression engines. We scale the frequency up to 511 MHz, i.e., its estimated frequency based on constant electrical field scaling if implemented in a 65 nm technology. 511 MHz is below a modern high-performance processor frequency. We assume an on-chip counter/divider is available to clock the MXT decompressor. However, decompressing a 64 B cache line will take 16 processor cycles in a 1 GHz processor, twice the time for C-Pack. The area cost of MXT is not reported.

**6.6.3.2 Comparing C-Pack with X-Match**:

X-Match has been implemented using 0.25 µm field programmable gate array (FPGA) technology. The compression hardware achieved a maximum frequency of 50 MHz with a throughput of 200 MB/s. To the best of our knowledge, the design was not synthesized using a flow suitable for microprocessors. Therefore, we ported our design for C-Pack for synthesis to the same FPGA used for X-Match [74] in order to compare the peak frequency and the throughput. Evaluation results indicate that our C-Pack implementation is able to achieve the same peak frequency as X-Match and a throughput of 400 MB/s, i.e., twice as high as X-Match's throughput. Note that in practical situations, C-Pack should be implemented using an ASIC flow due to performance requirement for cache compression.

**6.6.3.3 Comparing C-Pack with FPC** :

FPC has not been implemented on a hardware platform. Therefore, no area or peak frequency numbers are reported. To estimate the area cost of FPC, we observe that the FPC compressor and decompressor are decomposed into multiple pipeline stages as described in its tentative hardware design [10]. Each of these stages imposes area overhead. For example, assuming each 2-to-1 multiplexer takes 5 gates, the fourth stage of the FPC decompression pipeline takes approximately 290 K gates or 0.31 mm$^2$ in 65 nm technology, more than the total area of our compressor and decompressor. Although this work claims that time-multiplexing two sets of barrel shifters could help reduce area cost, our analysis suggest that doing so would increase the overall latency of decompressing a cache line to 12 cycles, instead of the claimed 5 cycles. In contrast, our hardware implementation achieves much better compression ratio and a comparable worst-case delay at a high clock frequency, at an area cost of 0.043 mm$^2$ compressor and 0.043 mm$^2$ decompressor in 65 nm technology.

### 6.6.4 Implications on Claims in Prior Work

Many prior publications on cache compression assume the existence of lossless algorithms supporting a consistent good compression ratio on small (e.g., 64-byte) blocks and allowing decompression within a few microprocessor clock cycles (e.g., 8 ns) with low area and power consumption overheads [64, 69, 116]. Some publications assume that existing Lempel–Ziv compression algorithm based hardware would be sufficient to meet these requirements [42]. As shown in Section 6.6.3.1, these assumptions are not supported by evidence or analysis. Past work also placed too much weight on cache line compression ratio instead of effective system-wide compression ratio (defined in Section 6.4.3). As a result, compression algorithms producing lower compressed line sizes were favored. However, the hardware overhead of permitting arbitrary locations of these compressed lines prevents arbitrary placement, resulting in system-wide compression ratios much poorer than predicted by line compression ratio. In fact, the compression ratio metric of merit for cache compression algorithms should be effective system-wide compression ratio, not average line compression ratio. Alameldeen et al. proposed *segmented compression ratio*, an idea similar to system-wide compression ratio. However, segmented compression ratio is only defined for a segmentation-based approach with fixed-size segments. Effective system-wide compression ratio generalizes this idea to handle both fixed size segments (segmentation-based schemes) and variable length segments (pair-matching based schemes). C-Pack was designed to optimize performance, area, and power consumption under a constraint on effective system-wide compression ratio.

C-Pack meets or exceeds the requirements assumed in former microarchitectural research on cache compression. It therefore provides a proof of concept supporting the system-level conclusions drawn in much of this research. Many prior system-wide cache compression results hold, provided that they use a compression algorithm with characteristics similar to C-Pack.

## 6.7 Conclusions

This chapter has proposed and evaluated an algorithm for cache compression that honors the special constraints this application imposes. The algorithm is based on pattern matching and partial dictionary coding. Its hardware implementation permits parallel compression of multiple words without degradation of dictionary match probability. The proposed algorithm yields an effective system-wide compression ratio of 61%, and permits a hardware implementation with a maximum decompression latency of 6.67 ns in 65 nm process technology. These results are superior to those yielded by compression algorithms considered for this application in the past. Although the proposed hardware implementation mainly targets online cache compression, it can also be used in other high-performance lossless data compression applications with few or no modifications.

CHAPTER 7

# Optimization Technique 2: Memory Access Aware On-Line Voltage Control for Performance and Energy Optimization

In the previous chapter, we present C-Pack, a high-performance cache compression algorithm that increases effective last-level cache size by adding low-overhead compression and decompression hardware. C-Pack is twice as fast as the best existing hardware implementations potentially suitable for cache compression. The design of C-Pack is inspired by the observation that last-level cache is critical to system performance. Therefore, it is desirable to significantly increase usable cache capacity at a low area cost, thereby reducing the amount of time spent on off-chip communication.

In this chapter, we will explore the impact of the memory hierarchy on another important metric: energy. Since energy can be expressed as the product of performance (in terms of execution time) and power, we first examine the performance and power implications of the memory hierarchy when deriving the CAMP model (see Chapter 3) and the system-wide power model (see Chapter 4). We first note that CAMP expresses SPI as a linear function of last-level cache miss rate. More specifically, the execution time of a process can be decomposed into on-chip time, i.e., the time spent waiting for on-chip resources such as integer unit and floating point unit to finish executing instructions that are dispatched to them, and the off-chip time, i.e., the time spent waiting for off-chip resources such as off-chip memory and hard disk to transfer data to the chip. While the on-chip time is usually inversely proportional to CPU frequency because most on-chip resources share the same clock, the off-chip time is independent of CPU frequency because the off-chip resources have their own clocks. Therefore, the same computer system may exhibit dramatically different performances to CPU-bound processes and memory-bound processes. Regarding the power model, we observed that the coefficient associated with L2MPS, i.e., the number of L2 cache misses per second, is negative. This implies that the processor is stalled during a last-level cache miss, thereby reducing the power consumption. Combining these two observations, we found a great energy saving opportunity at no performance cost: if we can adjust CPU voltage and CPU frequency during runtime with no performance overhead, we can reduce the voltage and frequency to the lowest level on every last-level cache miss and raise them back to the highest level as soon as the processor resumes execution. Although this is an ideal case since for example, adjusting voltage and frequency will always incur some performance penalty in the real world, it indicates the last-level cache miss rate is potentially a good metric to evaluate the trade-offs between performance and energy consumption. This intuition motivates a predictive on-line DVFS control algorithm that can produce close-to-optimal results with the aid of a performance model and a power model, as explained later in this chapter.

This chapter describes an off-chip memory access-aware runtime dynamic voltage and frequency scaling (DVFS) control technique that minimizes energy consumption subject to constraints on application execution times. 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 based on formulating the throughput-constrained energy minimization problem as a multiple-choice knapsack problem (MCKP). This algorithm uses a power model that adapts to application phase changes by observing processor hardware performance counter values. The solutions it produces provide upper bounds on the energy savings achievable under a performance constraint. However, this algorithm assumes a priori (oracle or profiling-based) knowledge of application phase change behavior. To relax this assumption, we propose P-DVFS, an predictive DVFS algorithm for on-line minimization of energy consumption under a performance constraint without requiring a priori knowledge of an application's behavior. P-DVFS uses hardware performance counter based performance and power models. It predicts remaining execution time online in order to control voltage and frequency settings to optimize energy consumption and performance. The P-DVFS problem is formulated 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 for our problem instances on average with a maximum deviation of 4.83%. In addition to producing results approaching those of an oracle formulation, P-DVFS reduces power consumption for our problem instances by 9.93% on average, and up to 25.64%, compared with the most advanced related work.

The rest of the chapter is organized as follows. Section 7.1 motivates the performance-constrained energy minimization problem and summarizes our contributions. Section 7.2 discusses the performance and energy trade-offs and presents the formal problem definition. Section 7.3 briefly describes the performance model, the power model, and their roles in the problem formulation. We also present the oracle algorithm which assumes a priori knowledge about application behavior and a practical predictive on-line DVFS algorithm. Section 7.4 describes the experimental results for the oracle algorithm and the predictive DVFS algorithm. We also compare the results with those produced by the most advanced related work. Section 7.5 concludes this chapter.

## 7.1 Introduction and Related Work

Energy consumption is important in 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 [28]. Zhang et al. developed a two-phase technique that integrates task assignment, task scheduling, and voltage selection for energy minimization [119]. Varatkar et al. proposed a communication-aware task scheduling and voltage selection algorithm to minimize the overall system energy consumption in a multiprocessor environment [108]. 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 and power consumption. One of the most promising of these is dynamic voltage and frequency scaling (DVFS). Two characteristics are important to DVFS control policies. First, 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. Second, if a DVFS control policy is to guarantee that a particular application consistently runs with adequate performance, e.g., honoring 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 [55]. However, this technique does not bound performance degradation. Wu et al. proposed dynamic compiler driven DVFS for controlling microprocessor energy and performance [113]. However, their work requires changes to the underlying compilation infrastructure. In addition, their technique does not attempt to honor performance constraints. Liu et al. proposed a technique to optimize peak temperature subject to a real-time performance constraint using DVFS [68]. However, their assumption that the execution time of a task is inversely proportional to CPU frequency is correct only for systems in which all layers of the memory hierarchy operate at the same frequency, as we will demonstrate in Section 7.2.1. The technique proposed by Choi et al. is the closest to ours [30]. The goal of their technique is to minimize energy consumption under a constraint on the total program execution time. Detailed comparisons with their work can be found in Section 7.4.2. 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. There also exist numerous prior work on DVFS-related scheduling and power budgeting in virutalized environment [109, 73].

In this paper, we propose

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, for our formulation and problem instances. However, it assumes access to oracle or profiling-based information about application behavior. In the rest of the chapter, we will use "optimal solution" in the context of our problem formulation when this does not introduce ambiguity.

2. We also propose P-DVFS, a predictive online DVFS algorithm that requires no a priori knowledge of application behavior. P-DVFS uses hardware performance counter based power and performance models to adapt to the behavior of running applications. 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. [30], 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 they have 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 optimal on average with a maximum deviation of 4.83%, for our problem instances. It improves energy consumption by 9.80% on average, and up to 29.86%, compared to the most advanced related DVFS control technique. P-DVFS also reduces power consumption by up to 25.64% (9.93% on average) compared with the most advanced related work.

## 7.2 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.

### 7.2.1 Performance and Energy Trade-Offs

The execution time of a task can be decomposed into on-chip and off-chip latencies. The latencies of on-chip components scale linearly with CPU frequency, because they 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 one or more separate clock.

The power consumption of a task can be divided into dynamic power and static power. Dynamic power consumption is caused by switching transistors charging and discharging capacitive loads. It generally scales superlinearly with CPU clock frequency [26]. Static power consumption is primarily due to gate and subthreshold leakage currents of transistors. It does not directly depend on CPU frequency but depends on the voltage. In general, reducing frequency and voltage reduces both dynamic and static power consumption.

Many modern processors support 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 7.4) implies that adjustments should happen no more frequently than once every hundred microseconds, thus limiting overhead.

### 7.2.2 Problem Formulation

The performance-constrained energy minimization problem can be formulated as follows: Given that $\alpha$ 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 performance constraint is soft, i.e., it is highly desirable to meet it. However, violating the constraint does not mean failure: a cost function may be associated with the degree of constraint violation.

As indicated in Section 7.2.1, the energy saving potential directly relates to the proportion of total execution time resulting from waiting for off-chip data access, which are primarily L2 cache misses in our experiments. 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 for intervals with low MPIs (to improve performance) and lower frequencies for intervals with high MPIs (to save energy).

In real operating systems, power control policies are usually implemented using adjustments at discrete time intervals. Discretized MPI values are used. We define a *control point* as a time at which control decisions are made and a *scaling point* as a 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 and use time multiplexing to emulate continuous DVFS within a control period.

Given an MPI distribution within a control period, $S$ is the set of all MPI slots and $F$ is the set of all available frequency levels. 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 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 $E_{total}$, i.e.,

$$E_{total} = I_{total} \cdot \sum_{i \in S} P_i(f_i) \cdot poi_i \cdot \text{SPI}_i(f_i) \text{ and} \tag{7.1}$$

$$T_{act} \leq (1+\alpha)T_{fmax}. \tag{7.2}$$

The goal is to minimize $E_{total}$ subject to Equation 7.2. Since the DVFS switching overhead ranges from 50 µs to 200 µs, the performance (or energy) overhead due to a frequency change 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 7.1 depends on both the CPU frequency and application behavior, e.g., the number of last-level cache misses per second (see Section 7.3.2).

## 7.3 System Modeling

In this section, we first explain our task performance and power models. We then formulate the energy minimization problem as a multiple-choice knapsack problem (MCKP) and solve it optimally, assuming knowledge of the average SPI at the maximum frequency ($\text{SPI}_{fmax}$) and the exact application MPI distribution. We then relax our assumptions and propose an execution time predictor that is accurate 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.

### 7.3.1 Performance Modeling

Equation 7.2 depends on a formula that accurately expresses the relationship between SPI, MPI, and CPU frequency. Intuitively, the amount of time consumed per instruction can also be decomposed into on-chip and off-chip latencies. On-chip latency is inversely proportional to frequency, while off-chip latency, captured by MPI, is independent of frequency. Prior work has reached the same conclusion [55]. SPI can be expressed as

$$\begin{aligned} \text{SPI}(\text{MPI}, f) &= c_1 \cdot \text{MPI} + c_2/f, \text{ or equivalently,} && (7.3) \\ \text{CPI}(\text{MPI}) &= c_1 \cdot f \cdot \text{MPI} + c_2, && (7.4) \end{aligned}$$

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 via fitting.

Most modern processors have built-in hardware performance counters that record information about architectural events, e.g., number of instructions retired and cache misses [3]. By gathering these two event counts, we can compute SPI and MPI during application execution. Therefore, given the last $N$ data points reported by hardware performance counters, we can determine $c_1$ and $c_2$ can be determined using linear regression. The relevant formulæ follow.

$$c_1 = \frac{N \cdot (\sum_{i=1}^{N} x_i \cdot y_i) - (\sum_{i=1}^{N} x_i) \cdot (\sum_{i=1}^{N} y_i)}{N \cdot (\sum_{i=1}^{N} x_i^2) - (\sum_{i=1}^{N} x_i)^2} \text{ and} \tag{7.5}$$

$$c_2 = \left( \sum_{i=1}^{N} y_i - c_1 \cdot \sum_{i=1}^{N} x_i \right) / N, \tag{7.6}$$

where $x_i$ denotes the product of MPI and CPU frequency for the $i$th data point and $y_i$ represents the CPI for the $i$th data point. Note that $N$ should be carefully chosen to capture changes in memory access pattern quickly and support accurate regression-based modeling. In our experiments, varying $N$ between 10 and 50 has insignificant impact on 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.

### 7.3.2 Power Modeling

Equation 7.1 indicates the necessity of having an accurate formula to describe the relationship between power consumption and MPI. Since an L2 cache misses are time consuming, 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 hardware performance counter events and observed that following five were sufficient to permit accurate estimation of power consumption:

1. number of L1 data cache references per second (L1DPS),
2. number of L2 cache references per second (L2PS),
3. number of L2 cache misses per second (L2MPS),
4. number of floating point instructions executed per second (FPPS), and
5. 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 depends linearly on these five events. In addition, the dynamic power consumption depends nonlinearly on CPU frequency [83]. Given that $f$ is the CPU frequency, the power consumption can be estimated as follows:

$$P = b_0 + b_1 \cdot \text{L1DPS} + b_2 \cdot \text{L2PS} + b_3 \cdot \text{L2MPS} + b_4 \cdot \text{FPPS} + b_5 \cdot \text{BRPS} + b_6 \cdot f^{1.5}, \quad (7.7)$$

where $b_i, i = 0, \cdots, 6$ are task-specific constants that can be determined during pre-characterization. The frequency exponent of 1.5 was determined empirically. It is worth mentioning that $b_0$ accounts for system idle and leakage power. For example, the formula for the “mcf” benchmark (see Section 7.4) follows:

$$P = 4.778 + 2.2864 \times 10^{-9} \cdot \text{L1DPS} + 6.517 \times 10^{-8} \cdot \text{L2PS} - 3.596 \times 10^{-7} \cdot \text{L2MPS} + \\ 0.6342 \cdot \text{FPPS} - 3.136 \times 10^{-9} \cdot \text{BRPS} + 4.308 \cdot f^{1.5}. \qquad (7.8)$$

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 [76], 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 Section 7.3.1.

### 7.3.3 Cost Minimization

This section describes formulation of the DVFS power management state control problem as a multiple-choice knapsack problem (MCKP). Given multiple sets, each containing multiple items, each of which 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 instance by treating each potential frequency level as 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 energy at the highest frequency. Note that, depending on whether we have a priori knowledge $\text{SPI}_{fmax}$ and the MPI distribution throughout program execution, the DVFS problem instance can be formulated as different MCKP instances, as explained in Section 7.3.3.2 and Section 7.3.3.3.

**7.3.3.1 Cost Function:** Equations 7.3 and 7.7 can be substituted into Equation 7.1. 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 on the frequency level assigned to MPI slot $i$. However, both are nonlinear due to the nonlinearity of SPI and power consumption in CPU frequency. The resulting nonlinear optimization problem cannot be efficiently solved online.

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

$$x_{ij} = \begin{cases} 1, & f_j \text{ is assigned to MPI slot } i \text{ and} \\ 0, & \text{otherwise.} \end{cases} \tag{7.9}$$

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.

$$\text{SPI}_i = \sum_{f_j \in F} x_{ij} \cdot (c_1 \cdot \text{MPI}_i + c_2/f_j) = c_1 \cdot \text{MPI}_i + \sum_{f_j \in F} c_2/f_j \cdot x_{ij}. \tag{7.10}$$

Since constants $c_1$, $c_2$, and $F$ are known at the control point, Equation 7.10 can be simplified as follows.

$$\begin{aligned} \text{Letting } s_0 &= c_1 \cdot \text{MPI}_i \text{ and} \\ s_j &= c_2/f_j, \forall f_j \in F, \\ \text{SPI}_i &= s_0 + \sum_{j=1}^{|F|} s_j x_{ij}. \end{aligned} \tag{7.11}$$

where $|F|$ is the number of elements in $F$. Similarly, the value of the five events in Equation 7.7 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

$$\text{L1DPS}_i(f_j) = \text{L1DPI}_i/\text{SPI}_i(f_j) \triangleq m_{ij,1}. \tag{7.12}$$

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

$$P_i = w_0 + \sum_{j=1}^{|F|} w_{ij} x_{ij}. \tag{7.13}$$

Combining Equations 7.11 and 7.13, Equation 7.1 can be rewritten as follows:

$$E_{total} = I_{total} \sum_{i \in S} poi_i \cdot (w_0 + \sum_{j=1}^{|F|} w_{ij} x_{ij})(s_0 + \sum_{k=1}^{|F|} s_k x_{ik}). \tag{7.14}$$

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

$$x_{ij} \cdot x_{ik} = \begin{cases} x_{ij}, \text{ if and only if } j = k \text{ and} \\ 0, \text{ otherwise.} \end{cases} \tag{7.15}$$

Therefore, Equation 7.14 can be simplified as follows.

$$\begin{aligned}
\text{Letting } e_0 &= I_{total} \cdot w_0 s_0 \text{ and} \\
e_{ij} &= poi_i(w_0 s_j + w_{ij} s_0 + w_{ij} s_j), \\
E_{total} &= e_0 + \sum_{i \in S} \sum_{f_j \in F} e_{ij} x_{ij}. \qquad (7.16)
\end{aligned}$$

**7.3.3.2 Performance Constraint – the Oracle 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. Our solution has two stages: profiling and evaluation. During profiling, 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 minimize energy consumption while honoring the performance constraint. The formulation we have just described computes 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 pre-characterization. We will evaluate the quality of this prediction-based technique, called P-DVFS, by comparing its results with those of the optimal oracle formulation.

Assuming the number of instructions associated with MPI slot $i$ is denoted as $I_i$, Equation 7.2 can be rewritten as

$$\sum_{i \in S} \sum_{f_j \in F} I_i \cdot \text{SPI}_i(f_j) \cdot x_{ij} \leq (1+\alpha) T_{fmax}. \tag{7.17}$$

Dividing both sides by $I_{total}$ yields

$$\sum_{i \in S} \sum_{f_j \in F} poi_i \cdot \text{SPI}_i(f_j) \cdot x_{ij} \leq (1+\alpha) \text{SPI}_{fmax}. \tag{7.18}$$

Although we can use Equation 7.3 to express SPI as a function of MPI and frequency, in reality we record $\text{SPI}_i(f_j)$ 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 closest lower level. When the frequency cannot be reduced further, we increase the frequency to the highest level. This process is repeated until the program under profiling finishes. We then compute the average $\text{SPI}_i(f_j)$ associated with each MPI slot $i$ and each frequency $f_j$. Hence, we can treat $\text{SPI}_i(f_j)$ as a constant $k_{ij}$. Equation 7.18 thus becomes

$$\sum_{i \in S} \sum_{f_j \in F} poi_i \cdot k_{ij} \cdot x_{ij} \leq (1+\alpha) \text{SPI}_{fmax}. \tag{7.19}$$

$I_{total}$ and $e_0$ are constants. Thus, the problem can be formulated as follows:

$$\text{Minimize } \textstyle\sum_{i \in S} \sum_{f_j \in F} e_{ij} x_{ij} \tag{7.20}$$

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

$$x_{ij} \in \{0,1\},\ \textstyle\sum_{f_j \in F} x_{ij} = 1, \forall i \in S \tag{7.21}$$

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 7.21 positive integers. Thus, the formulation can be treated as an multiple-choice knapsack problem (MCKP) [91]. We solve this problem optimally using "lp_solve" [5]. We record the frequencies assigned to each MPI value in an $|S| \times |F|$ lookup table. During evaluation, we use the current MPI value to look up and adjust the frequency at each scaling point.

**7.3.3.3 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 between present and future MPI distributions. It is known that most programs have repeated phases with periods ranging from 200 ms–2 s [53]. Therefore, this assumption holds given a reasonable observation duration. In our experiments, we use performance counter values during the most recent control period when deriving the optimal frequency settings for the next control period. We will also discuss our using solutions when the total number of instructions are known or unknown. In the rest of the chapter, we will use P-DVFS (*predictive DVFS*) to indicate the online predictive DVFS technique.

At each control point, the number of instructions retired is known. It is therefore 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 that Equation 7.16 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 elapsed time and $T_r$ is the remaining execution time, Equation 7.2 can be written as

$$T_r = I_r \cdot \sum_{i \in S} poi_i \cdot \mathrm{SPI}_i(f_i) \leq (1+\alpha)T_{fmax} - T_{elap}. \tag{7.22}$$

Equation 7.3 allows us to rewrite the left side of Equation 7.22 as

$$I_r \cdot \sum_{i \in S} poi_i \cdot \mathrm{SPI}_i(f_i) = I_r \cdot \sum_{i \in S} \sum_{f_j \in F} d_{ij} x_{ij}, \tag{7.23}$$

where $d_{ij} = poi_i / \left(c_1 \cdot \mathrm{MPI}_i + {}^{c_2}/_{f_j}\right), \forall f_j \in F$. Therefore, Equation 7.22 can be simplified as follows:

$$\sum_{i \in S} \sum_{f_j \in F} d_{ij} x_{ij} \leq \frac{(1+\alpha)T_{fmax} - T_{elap}}{I_r}. \tag{7.24}$$

**Execution Time Prediction:** Equation 7.24 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. 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 rapid and significant performance fluctuations and (2) the prediction algorithm should impose little overhead.

In order to derive a fast and accurate prediction method, 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 7.26. $f_k$ 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, i.e., the amount of time required to execute the same number of instructions in period $k$ when the highest frequency is employed.

$$T_{k,max} = T_{k,f_k} \cdot \frac{\text{SPI}(\text{MPI}_k, f_{max})}{\text{SPI}(\text{MPI}_k, f_k)}. \tag{7.25}$$

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

$$T_{elap,max} = \sum_k T_{k,max} = \sum_k \left( T_{k,f_k} \cdot \frac{\text{SPI}(\text{MPI}_k, f_{max})}{\text{SPI}(\text{MPI}_k, f_k)} \right). \tag{7.26}$$

In order to determine $T_{remain,max}$, we first assume the instruction count of the current task is known, e.g., by examining the input data 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 SPI$(f)$ is the amount of time per instruction at frequency $f$, we can express $T_{remain,max}$ as follows.

$$I_r = I_{total} - I_{elap} \text{ and} \tag{7.27}$$

$$T_{remain,max} = I_r \cdot \text{SPI}(f_{max}) \tag{7.28}$$

Combining Equations 7.26 and 7.28, $T_{fmax}$ can be expressed as

$$T_{fmax} = T_{elap,max} + T_{remain,max}. \tag{7.29}$$

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, in billions. We start with an $I_r$ of 1. At every scaling point, we subtract, from the current $I_r$, the number of instructions retired since the last reset of $I_r$. 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 7.28 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 (billion) 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 if $I_{total}$ is known beforehand, i.e., from pre-characterization, 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.

$$\text{Minimize } \textstyle\sum_{i\in S}\sum_{f_j\in F} e_{ij}x_{ij} \tag{7.30}$$

$$\text{subject to } \textstyle\sum_{i\in S}\sum_{f_j\in F} d_{ij}x_{ij} \le \frac{(1+\alpha)T_{fmax}-T_{elap}}{I_r} \text{ and} \tag{7.31}$$

$$x_{ij}\in\{0,1\},\ \textstyle\sum_{f_j\in F} x_{ij}=1, \forall i\in S. \tag{7.32}$$

We can treat the right hand side of the constraint in Equation 7.31 as positive. Otherwise, the constraint is trivially satisfied. Unlike the oracle scenario, the P-DVFS technique re-

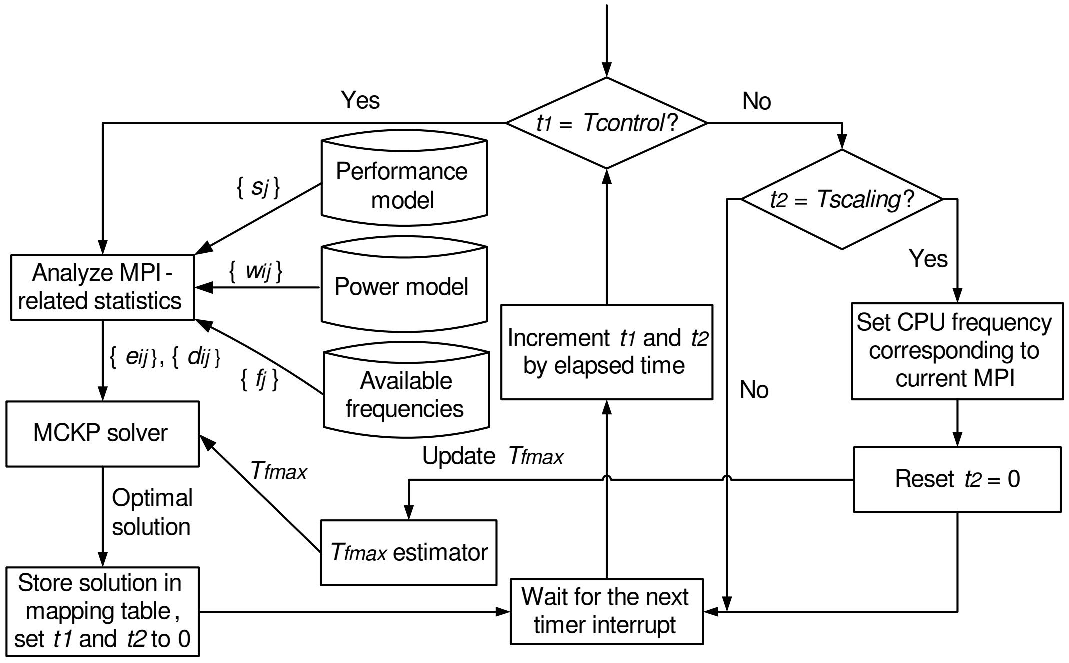

Yes
t1 = Tcontrol?
No
{ sj }
Performance model
t2 = Tscaling?
Analyze MPI-related statistics
{ wij }
Power model
Yes
Increment t1 and t2 by elapsed time
Set CPU frequency corresponding to current MPI
{ eij }, { dij }
{ fj }
Available frequencies
No
MCKP solver
Update Tfmax
Reset t2 = 0
Tfmax
Optimal solution
Tfmax estimator
Store solution in mapping table, set t1 and t2 to 0
Wait for the next timer interrupt

Figure 7.1: System architecture for P-DVFS.

quires solving MCKP online. Although MCKP is $\mathcal{NP}$-hard, there exist algorithms that can solve it in pseudo-polynomial time [75, 91]. We used "lp_solve" to obtain optimal solutions online. Our experiments had 15 MPI slots and 4 frequency levels. For each of the evaluated benchmarks, 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 the control period of 1 s in our experiments. Pisinger's MCKP solver implementation would permit an even more efficient solution in a production version of the control software [75].

### 7.3.4 P-DVFS System Architecture

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 7.1 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 7.1, 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., divide the range of MPI values into distinct MPI slots and calculating the percentage of instructions ($poi_i$) associated with each MPI slot $i$, and determine the values of coefficients such as $\{s_j\}$ in Equation 7.11 and $\{w_{ij}\}$ in Equation 7.13 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 Equations 7.30 and 7.31, which are then provided to the MCKP solver along with estimates of $T_{fmax}$ and $I_r$ in Equation 7.27 and 7.28. 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 the value 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%).

## 7.4 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 previous work [30].

### 7.4.1 Experimental Setup

We implemented our techniques on a Pentium Dual Core E2220 processor running Linux 2.6.25 and operates at 1.2, 1.6, 2.0, and 2.4 GHz. Experimental results indicate the switching overhead ranges from 50 µs to 200 µs. We use PAPI 3.6.2 [6] for hardware performance counter measurement and experimentally determined that the performance overhead for accessing hardware performance counter is negligible. Due to hardware limitations, 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. The same MPI slots are used throughout the execution of a benchmark.

To determine power consumption, we use a Fluke i30 current clamp on the 12 V processor power supply lines, the output of which is sampled at 10 kHz 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 of 90%.

### 7.4.2 Comparison with Prior Work

Choi et al. [30] 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 adapt to changes in the rate 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 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 to minimize system energy consumption instead of processor power consumption. In contrast, P-DVFS automatically models and optimizes leakage power consumption and can be easily extended to handle the energy

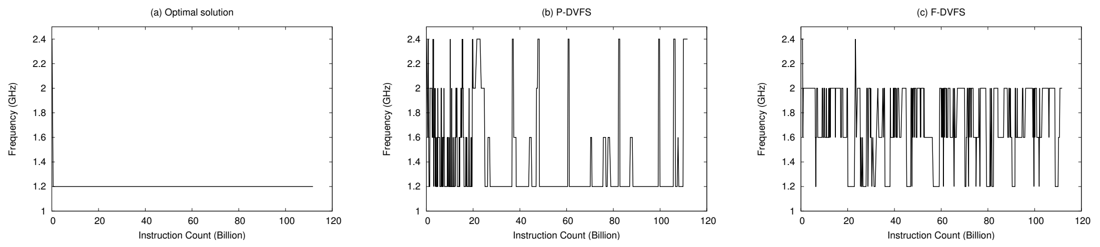


Figure 7.2: Processor frequency as a function of the number of instructions retired for (a) the oracle solution, (b) P-DVFS, and (c) F-DVFS for "mcf" with a performance degradation ratio of 20%.

consumptions of other components such as main memory and disk.

### 7.4.3 Experimental Results

We evaluated P-DVFS on the 8 SPEC2000 benchmarks that compiled on our evaluation platform and 3 ALPBench benchmarks [45, 15]. 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 40–425 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 (N-DVFS), F-DVFS, and the optimal oracle solution; we use the same window size for each to permit a fair comparison. Both techniques use 4 discrete frequency levels.

Table 7.1 shows the actual performance degradation for both F-DVFS and P-DVFS compared with the user-specified performance degradation ratios. The first column specifies the benchmarks we evaluated. The "P-DVFS" and "F-DVFS" columns indicate the

Table 7.1: Performance Degradations of F-DVFS and P-DVFS in terms of total execution time

| 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 |

performance degradation ratios resulting from using the two techniques, with the user-specified performance degradation constraint listed on the second "Goal" row. Given that the performance constraint is satisfied, a larger performance degradation usually corresponds to more energy savings; this was confirmed by our experiments. Experimental results indicate that P-DVFS approaches the user-specified performance level more closely than F-DVFS, implying greater energy savings. P-DVFS has finer-grained control over the trade-offs between performance and energy given a user-desired performance constraint. F-DVFS 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, F-DVFS 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 F-DVFS does not work as well as P-DVFS, as we will explain later in this section. Note that both techniques may violate the soft timing constraint due to inaccuracies in the online performance model.

Table 7.2: 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 |

However, for P-DVFS, the maximum violation for these benchmarks is less than 1%, which could be eliminated by using a 1% guard band for the constraint.

We compared the energy savings of N-DVFS, F-DVFS, and P-DVFS with those of the 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 7.2. 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

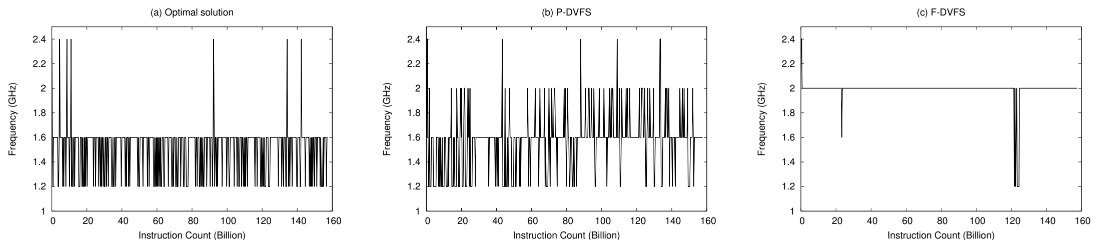


Figure 7.3: Processor frequency as a function of the number of instructions retired for (a) the optimal solution, (b) P-DVFS, and (c) F-DVFS during "art" execution with a performance degradation ratio of 20%.

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 7.2, 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 9.80% 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 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.

For benchmarks "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 for these benchmarks.

**Analyzing *Mcf* Results:**Figure 7.2 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 indicates the number of billion instructions retired and the Y-axis indicates the frequency. Figure 7.2(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). 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. 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, which approach those of the optimal oracle solution.

**Analyzing *Art* Results:**Figure 7.3 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%. 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. The coefficients in the regression formula remain nearly constant due to the small period and large window size. 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 solves it to get the optimal solution. This solution uses high frequency (2 GHz) for sampling points with low MPI and low frequency (1.2 GHz) for sampling points with high MPI. This permits significant reduction in energy consumption compared to F-DVFS. Since F-DVFS is not distribution-oriented, it cannot determine how SPI and power consumption change with MPI. Therefore, it cannot 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. P-DVFS achieves the greatest energy savings compared to past work for applications with phases during which the energy cost per instruction differ.

## 7.5 Conclusions

This chapter has described 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 soft performance constraint. P-DVFS requires no a priori knowledge of application behavior. P-DVFS also uses a 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 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 solutions on average with a maximum deviation of 4.83%, whereas the most advanced related DVFS control technique (F-DVFS) results in energy consumptions within 9.80% 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 9.93% on average and up to 25.64% compared to F-DVFS. These energy and power savings are all directly measured on a real system.

CHAPTER 8

# Optimization Technique 3: Power-Constrained Throughput Maximization in CMPs With Chip-Wide DVFS

In the previous chapter, we investigate the impact of performance and power models on local power state control techniques for throughput-constrained energy minimization. As demonstrated in Chapter 7, with the aid of an accurate performance model and power model, a predictive on-line DVFS control algorithm can precisely determine the trade-off between performance and energy consumption, thus achieving close-to-optimum results under a performance constraint by adapting to an application's off-chip memory access patterns.

In this chapter, we will explore the implications of accurate performance and power models for performance optimization in CMP processors with chip-wide DVFS. This work builds upon our observations and experience gathered from modeling and optimizing high-performance processors. Having demonstrated (1) CAMP can accurately predict the cache contention level in single-chip processors in Chapter 3 and (2) accurate power models can aid online power control techniques for energy minimization in Chapter 7, we intend to examine the possibility of improving system-level performance in CMPs under the guidance of such models. In particular, we plan to solve a power-constrained throughput maximization problem for multi-chip, CMP platforms with chip-wide DVFS. Based on the conclusions of Chapter 3 and Chapter 7, it is plausible to anticipate significant performance improvement over prior techniques by (1) carefully selecting process-to-core mappings during assignment and (2) intelligently adjusting voltage and frequency at runtime while still honoring the power constraint.

With the pressing need to address on-chip power and thermal issues, CMPs are gradually taking the place of single-core processors. Various power management techniques have been proposed to maximize system performance under a given power constraint, most of which assume availability of DVFS mechanisms to dynamically adjust system power consumption. However, some solutions rely on open-loop ad hoc search without the aid of accurate performance or power models, while others adopt over-simplified models in their problem formulation, leading to suboptimal results (see Section 8.2. In addition, none of them considered combining assignment techniques with local power state control algorithms to systematically improve performance under a given power budget. This chapter proposes a system-level performance maximization technique (PerfMax) subject to constraints on processor power consumption for a multi-chip CMP platform with chip-wide DVFS. PerfMax consists of two major algorithms: a CAMP-guided assignment optimization algorithm and an online DVFS control algorithm based on a nonlinear system-level power model (NLP). The assignment optimization algorithm uses CAMP to predict the performance impact of individual tentative assignment decisions and chooses the one that maximizes system-level throughput, accounting for cache contention, process phases, and off-chip memory contention. The online DVFS control algorithm builds upon an application-dependent performance model and a system-level power model, assuming the existence of a power monitoring device. It formulates the power-constrained performance maximization problem as a constrained nonlinear programming problem, solves it online, and uses the solutions to guide per-chip frequency selection. PerfMax combines the solutions from both algorithms to produce a global solution that spans both the assignment layer and the local power state control layer. We evaluate PerfMax using direct measurement of a real DVFS-equipped system. When bounding power consumption to at most 87.5% of that at the maximum frequency and voltage, PerfMax improves performance by 13.94% on average and up to 23.87% for SPEC CPU2000 benchmarks, and 16.7% on average and up to 34.03% for BioBench benchmarks when compared to the most closely related work, while still honoring the power constraint.

The rest of the chapter is organized as follows. Section 8.1 motivates the power-constrained performance maximization problem and summarizes our contributions. Section 8.2 describes related work on process assignment and power control. Section 8.3 presents the problem definition and the proposed system architecture. Section 8.4 describes CAMP-guided assignment optimization algorithm and explains how to handle multi-phase processes and off-chip memory contention in addition to cache contention. Section 8.5 formulates the target problem as a constrained nonlinear optimization problem by integrating the performance model and the power model into the problem formulation. We also explain how the power model adjusts itself during runtime based on the readings from the power monitoring device to correct for model inaccuracies. Section 8.6 provides the experimental setup and empirical experimental results conducted for PerfMax on a physical testbed with different benchmark suites. We also compare the results with those produced by the most related work. Section 8.7 concludes this chapter.

## 8.1 INTRODUCTION

With the pressing need to address on-chip power and thermal issues without sacrificing performance, CMPs are gradually taking the place of single-core processors. However, power consumption still remains the major constraint for further performance improvement of CMP processors due to the ever-increasing demand for single-thread performance growth. In addition, the power consumption of a high-performance processor often needs to be controlled to meet a predetermined design constraint.For example, the maximum power density of computing equipment in a server room is usually limited, which can be translated into different power constraints for each server in it. Similarly, a datacenter is designed with a pre-determined power capacity, thus implicitly imposing a per-server power constraint. In addition, the electricity bill of the datacenter directly depends on the peak power it draws. By constraining the peak power of each server in the datacenter, we can control and potentially cut down the eletricity cost. Finally, large-scale datacenters usually employ over-subscription to maximally utilize the provisioned power capacity. In over-subscribed datacenters, the sum of the possible peak power consumptions of the servers is greater than the provisioned capacity. This indicates the necessity of having an intelligent power management technique to dynamically control the power consumption of each server to ensure that the datacenter power consumption never exceeds its capacity. However, a poorly designed power management scheme may be overly conservative, resulting in unnecessary performance degradations. Therefore, an online power control technique needs to be carefully designed to maximize system performance while ensuring the power consumption stays below the allowed threshold.

Researchers have proposed various power management mechanisms that trade off performance for power consumption, among which DVFS remains the most popular method due to its low overhead and high availability on modern CMP systems [71, 118, 66]. However, although researchers have studied the benefits and overheads associated with per-core DVFS by implementing one on-chip regulator per core [62], most current processors are still using off-chip voltage regulators (or a single on-chip regulator) to control the chip voltage. Therefore, all the cores sitting on the same chip are forced to use the same CPU voltage, although they may or may not use different frequencies [80]. Given the additional design complexity and hardware overhead of adding per-core on-chip voltage regulators, multi-chip CMPs with chip-wide DVFS capability are commonly seen. Such processors often adopt a symmetric multiprocessor design with multiple cores sitting on each chip. On these machines, non-uniform DVFS can still be achieved on a per-chip basis. Recent work [118, 71] recognized this design constraint and proposed various optimization techniques for these platforms.

There exist several major challenges to designing an effective power control algorithm for a multi-chip CMP with chip-wide DVFS. First, during process assignment, the performance and power implication of each tentative assignment decision must be carefully evaluated. This is essential because assignment decisions can have a significant impact on the overall power efficiency; serious missteps made during assignment might be too difficult to correct during local power state control (see Section 8.6). Second, processes running on different cores may have different performance and power characteristics. For instance, some cores may be running CPU-intensive processes with few cache misses, while others are hosting memory-intensive processes with a large number of cache misses. Hence, the power control algorithm should be able to handle heterogeneous concurrently running processes by simultaneously adjusting the power states of different cores to maximize power efficiency under the given power budget. Third, the control strategy must adapt to temporal workload variations on different cores to achieve optimal control performance. Control algorithms that rely on static models may lead to unnecessary performance degradations and runtime power constraint violations due to model inaccuracies. Fourth, the control algorithm must be extremely fast; performance overhead associated with the controller leads to diminishing returns or even overwhelms the benefits of online power control. Finally, the algorithm should require no changes to the underlying hardware or operating system so that it is applicable to platforms with different architectures or operating systems.

Our work makes the following main contributions:

- We propose PerfMax, a two-stage algorithm for power-constrained performance maximization. Unlike most existing work that treats process assignment and local power state control as two different problems, PerfMax considers them as two optimization stages in our problem. It combines assignment decisions with power control techniques to maximize performance on both levels, therefore achieving better results than those when only a single stage is considered.

- While most prior work uses simple heuristics to guide process assignment such as similarity grouping or complementary mixing (see Section 8.2), PerfMax uses CAMP to predict the performance impact of individual tentative assignment decisions and chooses the one that maximizes system-level throughput, accounting for cache contention, process phases, and off-chip memory contention, which yields better prediction accuracy.

- Unlike most existing work that relies on open-loop search or heuristics for dynamic power management, we formulate the power-constrained performance maximization problem as a constrained nonlinear programming problem in PerfMax by integrating application-dependent performance models and a system-level power model into the problem formulation. PerfMax then solves the problem periodically during runtime and uses the solutions to guide per-chip frequency selection.

- To accommodate temporal workload variations, the power model dynamically adjusts its parameters based on the feedback from the power monitoring device, enabling accurate power estimation. In addition, both the performance model and the power model only require standard hardware performance counters (HPCs) for prediction, results, thus requiring no changes to the underlying hardware or operating system.

- While most existing work is validated using simulation, we evaluated PerfMax on real physical platforms for different benchmark suites. We also compared PerfMax with two state-of-the-art techniques: "Random + Priority" and "Similarity + MPC" (see Section 8.6.5). Experimental results indicate that PerfMax is able to achieve an average performance improvement of 13.94% for SPEC CPU2000 benchmarks and 16.7% for BioBench benchmarks compared to the most related work, while still honoring the desired power constraint.

## 8.2 RELATED WORK

The proposed power-constrained performance maximization technique builds upon prior work on performance-oriented process assignment optimization and dynamic power control.

Researchers have proposed performance maximization techniques by carefully selecting process-to-core mappings during process assignment. Merkel and Bellosa proposed a memory-aware application scheduling technique for CMPs with chip-wide DVFS to minimize energy-delay product [71]. Zhang et al. proposed grouping applications with similar frequency-to-performance effects to create opportunities for setting a chip-wide desirable frequency level in the later stage [118]. However, both techniques are solving different problems from ours. In particular, they intend to maximize energy-delay product or energy savings with bounded performance degradations. In contrast, PerfMax solves a performance maximization problem under a given power budget, thus requiring an accurate power model. In addition, the "similarity" metric proposed by Zhang et al. is ad hoc and thus may not applicable to other benchmarks or CMP systems with different architectures. In contrast, PerfMax uses CAMP to accurately predict the performance implications of each tentative assignment to select the best assignment, making it applicable to a wide range of applications and platforms. Teodorescu and Torrellas proposed process variation (PV) aware algorithms for application scheduling and power management by formulating the power-constraint throughput maximization problem as a linear programming problem [104]. They proposed assigning processes with highest IPC to cores with highest peak frequency. However, our target platform is a multi-chip homogeneous CMP, in which different cores have the same peak frequency, making the PV-aware assignment algorithm inapplicable. In addition, they model the performance and processor power as linear functions of frequency, leading to 132% and 9.1% error in performance and power estimation, respectively [46]. In contrast, PerfMax relies on accurate performance and power models to adapt to different application behavior (e.g., cache miss rates), leading to better prediction accuracy and thus better results.

There is prior work on DVFS-based power control techniques in CMP systems. Isci et al. analyzed several different policies for global CMP power management assuming per-core DVFS is available [54]. They proposed MaxBIPS, a prediction based algorithm to choose the combination of power modes for each core to maximize throughput while adhering to a chip-level power constraint. However, they assume that performance is a linear function of frequency and power is a cubic function of frequency, potentially resulting in large estimation error and constraint violations. MaxBIPS uses exhaustive search to find the best combination, making it inapplicable to CMPs with a large number of cores. Wang et al. proposed a chip-level power control algorithm for performance optimization based on model predictive control theory [111]. To our knowledge, their work is the closest to ours. In their problem formulation, they intend to minimize the weighted sum of the quadratic difference between each core's frequency and the peak frequency, with each core's weight being its CPU utilization. In addition, they assume power consumption is a linear function of frequency and dynamically adjusts the coefficients in their power model based on the feedback from the on-chip current sensor. However, their technique suffers from inaccurate performance and power models and relatively large solver overhead, leading to suboptimal results (see Section 8.6). Finally, none of the aforementioned power control algorithms considered the impact of process assignment on system-level throughput.

Some researchers considered the trade-off between performance and power. Li and Martinez proposed dynamically adjusting the number of active cores and applying DVFS to optimize power for a parallel application under a given performance constraint [66]. Lee et al. proposed a linear runtime performance projection model for dynamic power management [65]. However, power is not a constraint in their problem formulation.

## 8.3 PROBLEM DEFINITION AND SYSTEM ARCHITECTURE

In this section, we first describe the formal definition for the power-constrained performance maximization problem. We then explain the system architecture of PerfMax.

The power constrained throughput maximization problem in a chip-wide DVFS enabled, multi-chip CMP can be formulated as follows: given (1) $K$ processes to be assigned to a $N$-core, $M$-chip processor with chip-wide DVFS and (2) a power constraint $P_{max}$, determine (1) the process assignment and (2) the chip-wide CPU frequency schedules for each chip such that the the total system power is no more than $P_{max}$ while the system-level throughput (measured in terms of IPS, i.e., number of instructions retired per second) is maximized.In this dissertation, we focus on the scenario where $K = N$ because it allows us to simplify our explanations. We note that the same analysis can also be applied to cases where $K \neq N$. In addition, we impose several constraints on the target problem: (1) we examine the problem in a multi-programmed CMP environment, in which there is limited communication among processes. (2) The processes of interest have limited I/O access. (3) Process migration is not considered in this work.

In order to solve this problem, we make three important observations: (1) different process assignments have different impact on system performance. For example, when “art” and “gzip” are running on the two cores of chip 0 and “mcf” and “mesa” are running on the two cores of chip 1 on our two-chip, four-core testbed, the system-level throughput is 18.6% higher than that when “art” and “mcf” are running on chip 0 and “gzip” and “mesa” are running on chip 1. (2) Different power controllers can affect system performance differently. For instance, we implemented two power controllers in prior work (see Section 8.6). When “vpr”, “mesa”, “gzip”, and “parser” are simultaneously running on our testbed with MPC being the power controller, the throughput is 7.76% higher than that when using Priority to control the processor power. (3) When optimizing system throughput, the assignment layer and the power control layer are weakly coupled. For example, we conduct two experiments on our four-core, two-chip testbed (see Section 8.6) using 30 process combinations, each of which contains four processes randomly selected from SPEC CPU2000 benchmarks. Hence, each process combination is associated with three different assignments. In the first experiment, we determine the system throughput associated with three assignments for each process combination and identify the assignment with the highest throughput. We note that no power controllers exist in this experiment, i.e., all cores are running at the maximum frequency after the four processes are assigned. The resulting 30 assignments are denoted as *group A*. In the second experiment, we determine the amount of performance degradation (compared to the oracle solutions) as a result of ignoring the inter-dependency between the assignment layer and the power control layer when assigning processes. We first implemented the model predictive control (MPC) theory based technique (see Section 8.6) as our power controller. The power constraint is set to 87.5% of the full CMP power. For each process combination, we then determine the differ-

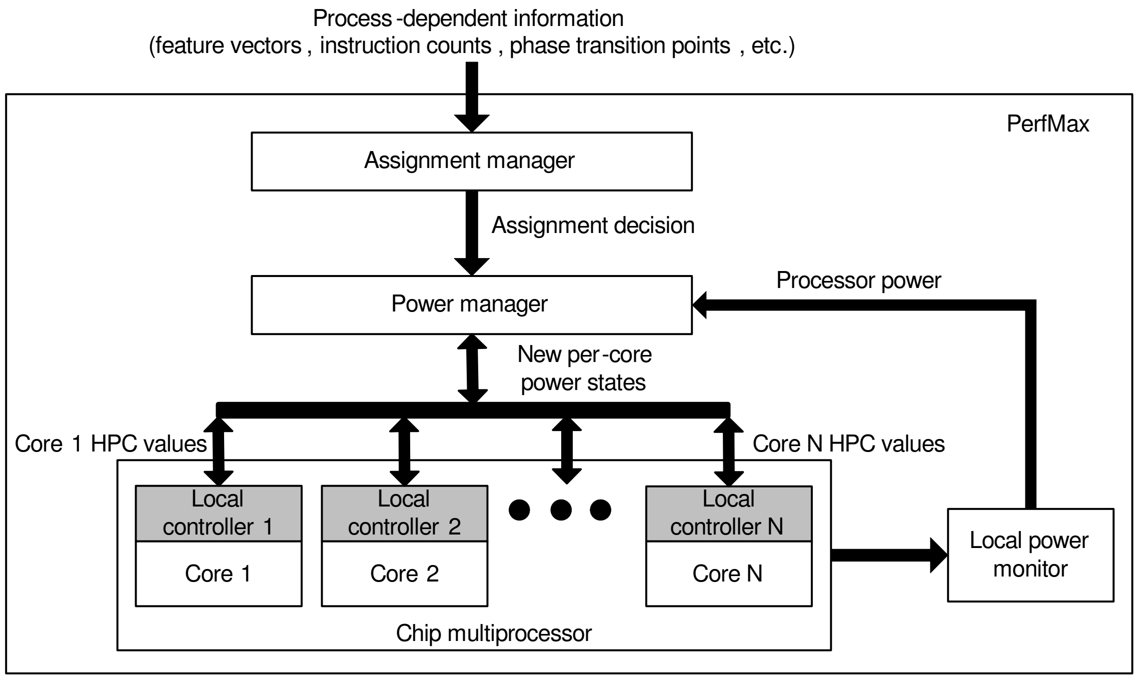


Figure 8.1: System architecture of PerfMax.

ence between the highest throughput achieved across all three assignments (denoted as the oracle solution) and that achieved by using the corresponding assignment in group A, with the MPC power controller dynamically adjusting the power to meet the power constraint after the processes are assigned. Our experiments indicate that compared to the oracle solutions, using the assignments in group A only degrades performance by 0.97% on average. Combining these observations, we conclude that the power-constrained performance optimization problem can be solved by dividing it into two subproblems, namely assignment optimization and power control optimization, solving the two subproblems separately, and combining the solutions together.

Figure 8.1 illustrates the system structure of PerfMax, the proposed solution to the power-constrained throughput maximization problem. PerfMax consists of three components: the assignment manager, the power manager, and the power monitoring and control layer. A $N$-core, $M$-chip machine has a single assignment manager, a single power manager, a single local power monitor, and $N$ core-level local controller.

The assignment manager (see Section 8.4) takes process-dependent information such as feature vectors, instruction counts, and phase transition points (see Chapter 3) as inputs. Such information can be gathered by profiling each individual process separately on a single core of the target machine. The assignment manager is responsible for identifying the best process-to-core mapping among all possible mappings. Ideally, it should be able to account for the performance impact due to cache contention and off-chip memory contention as a result of each assignment to select the best candidate.

After the assignment decision is made, the power manager (see Section 8.5) is notified. The power manager periodically communicates with the local controllers sitting on each core to collect statistics about currently running processes. Such information is then converted to power estimations based on the system-level power model stored internally. In order to adapt to temporal workload variations and compensate for inaccuracies in the power model, the power manager also takes periodic readings from the local power monitor as feedback and adjusts the power model accordingly. Finally, the power state control algorithm embedded in the power manager uses per-core performance and power data to determine the appropriate power states for each core while ensuring the power constraint is not violated. Among the different functionalities of the power manager, the design of the power state control algorithm is a major challenge, as explained in Section 8.1.

Within each core, the local controller collects process-dependent information such as last-level cache miss rates using hardware performance counters and periodically sends it to the power manager. In return, it receives power state updates from the power manager and

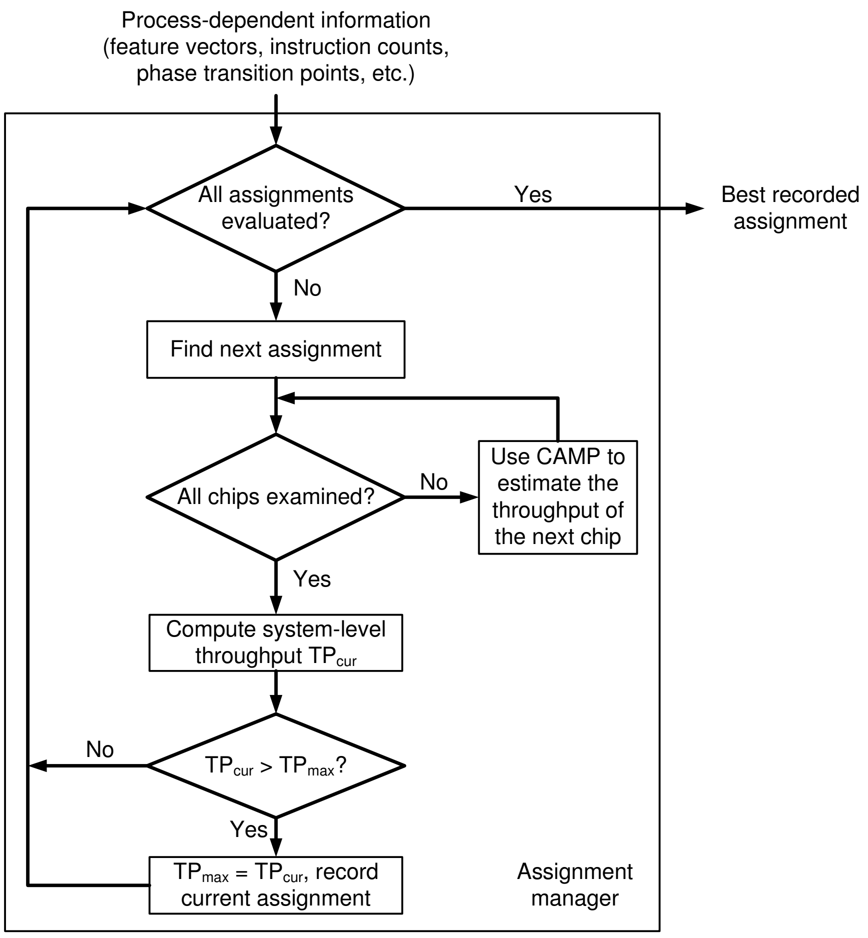

Process-dependent information (feature vectors, instruction counts, phase transition points, etc.)
All assignments evaluated?
Yes
Best recorded assignment
No
Find next assignment
All chips examined?
No
Use CAMP to estimate the throughput of the next chip
Yes
Compute system-level throughput TPcur
No
TPcur > TPmax?
Yes
TPmax = TPcur, record current assignment
Assignment manager

Figure 8.2: System architecture of the assignment manager.

adjusts its frequency and voltage level accordingly. PerfMax also assumes the existence of a local power monitor to provide processor power information during runtime. Such monitoring module could be an on-chip current sensor similar to that available in the Foxton controller of the Montecito chip [70] or an external measurement device such as a data acquisition card connected to a current clamp that measures processor current.

## 8.4 CAMP-Guided Process Assignment Optimization

In this section, we describe the assignment manager in PerfMax in more detail. We first explain how to use CAMP to guide process assignment. We then discuss how to handle multi-phase processes and off-chip memory contention.

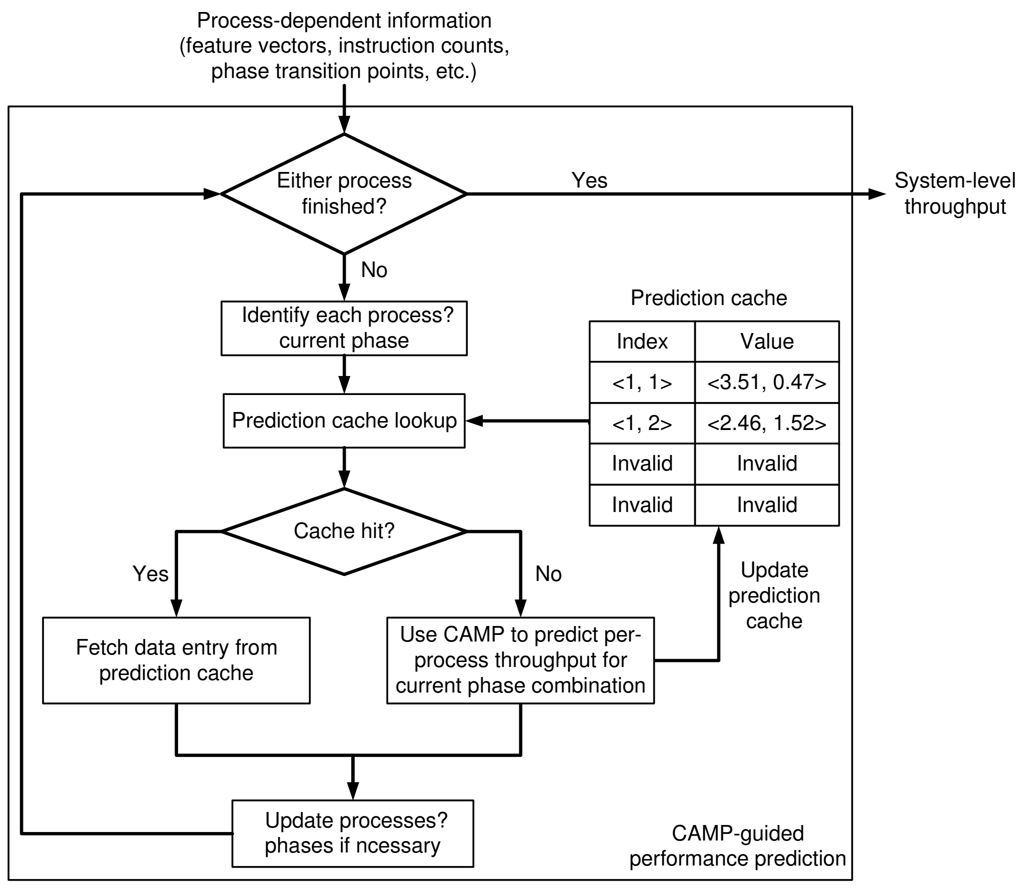

Process-dependent information (feature vectors, instruction counts, phase transition points, etc.)
Either process finished?
Yes
System-level throughput
No
Identify each process? current phase
Prediction cache
Index
Value
<1, 1>
<3.51, 0.47>
<1, 2>
<2.46, 1.52>
Invalid
Invalid
Invalid
Invalid
Prediction cache lookup
Cache hit?
Yes
No
Update prediction cache
Fetch data entry from prediction cache
Use CAMP to predict per-process throughput for current phase combination
Update processes? phases if ncessary
CAMP-guided performance prediction

Figure 8.3: CAMP-guided performance prediction for multi-phase processes.

As explained in Chapter 3, CAMP provides us with a fast and accurate way to estimate cache contention, and therefore instruction throughput, as a result of each tentative assignment. It takes (1) process reuse distance histogram, (2) cache access frequency, and (3) the relationship between performance and cache miss rate of each process as input and accurately predicts the effective cache size of each process when they are simultaneously running on different cores of the same chip. Hence, the assignment manager uses CAMP to evaluate the performance implications of each individual assignment and selects the best candidate.

Figure 8.2 illustrates the system architecture of the assignment manager in PerfMax. The input to the assignment manager includes process-dependent information such as the feature vectors, instruction counts, and phase transition points. It then checks whether all the assignments have been evaluated and if so, outputs the best recorded assignment. Otherwise, it finds the next tentative assignment and uses CAMP to predict the instruction throughput of each chip until all the chips have been examined. Finally, it computes the aggregated system-level instruction throughput and saves or discards the current assignment, depending on whether the current throughput $TP_{cur}$ is larger than the best recorded throughput $TP_{max}$. Note that $TP_{max}$ is initialized to -1, which is not shown in Figure 8.2.

### 8.4.1 Handling Multi-Phase Processes

When predicting the throughput of a single chip, CAMP uses a single cache reuse distance histogram for each process. However, processes can have phases due to loop-oriented execution semantics, with each phase associated with a distinct cache reuse distance histogram [54]. Therefore, it is necessary to profile each phase of a process separately and treat it as if it were a new process with its own feature vector during performance prediction. Process phase detection has been well studied in the past [34, 55]. Since automatically detecting phase transitions is beyond the scope of our work, we assume the phase transition points are known a priori. In our experiment, we sample the hardware performance counters every 100 million instructions when profiling a process and identify the phase transition points by grouping sample points with similar L2 cache miss rate patterns.

Figure 8.3 illustrates how the assignment manager uses CAMP to predict the performance of multi-phase processes on a single chip. Although a 2-core chip is shown in Figure 8.3, the algorithm is equally applicable to chips with more than two cores. Given process information such as feature vectors associated with each phase and phase transition points, the algorithm first checks whether either process has terminated and if so, outputs predicted system-level throughput information since we are interested in the system throughput when all processes are concurrently running. Otherwise, it identifies the current phase of each process and uses the phase combination as lookup index to query the prediction cache. Figure 8.3 shows a prediction cache with four entries, the first two of which store the mappings from a phase combination to predicted per-process throughput. If a cache hit occurs, the algorithm fetches the corresponding prediction results from the prediction cache. Otherwise, it uses CAMP to predict each process' throughput under current phase combination and updates the prediction cache. Finally, it updates the current phases of processes that encounter a phase transition point and repeats the aforementioned steps. The algorithm terminates when any process exhausts its instructions.

### 8.4.2 Off-Chip Memory Contention

In addition to cache contention, the assignment manager must also take off-chip memory contention into account when estimating system performance. When using CAMP for performance prediction, we assume that compared to cache contention, off-chip memory contention has second-order effects on performance and thus can safely be ignored when predicting performance. This assumption has been validated on multiple two-core processors in Chapter 3 and is consistent with our experimental results. More specifically, we examined all 55 pairwise combinations of 10 SPEC CPU2000 benchmarks (see Section 8.6) on two cache-sharing cores of chip 0 on our two-chip processor while leaving chip 1 idle (known as the *2-core case*). Experimental results indicate CAMP has an average of 3.94% performance estimation error over all 55 combinations, with a maximum error of 13.86%. However, this assumption fails when all 4 cores are simultaneously hosting memory-intensive processes. For example, when four instances of "mcf" (a program with high L2 cache miss rate) are concurrently running, the measured throughput is 28.52% lower than the predicted throughput. We hypothesize that this inaccuracy is due to off-chip memory contention. We first note that our target process has a 64-bit wide front side bus (FSB) operating at a frequency of 266 MHz that performs 4 transfers per cycle [4]. Thus, the memory bus transfer rate is $8 \times 266 \times 4 = 8.512$ GB/s, assuming the memory bus and FSB operate at the same frequency. On the other hand, the average MPS (number of L2 cache misses per second) of "mcf" is $1.517 \times 10^7$ in the 2-core case. In addition, the target processor always fetches two adjacent cache lines on a L2 cache miss [3]. Therefore, the average system-wide MPS when four instances of "mcf" are concurrently running can be calculate as $4 \times 1.517 \times 10^7 \times 128 = 7.767$ GB/s, which is very close to the memory bus transfer rate of 8.512 GB/s. As a related experiment, we simultaneously run four instances of each SPEC CPU2000 benchmark on four cores (known as the 4-core case) and analyze the performance prediction errors. The experimental results are consistent with our hypothesis: (1) the prediction errors are larger for processes with larger MPS and (2) the prediction errors for processes with low MPS are the same for the 2-core case and the 4-core case.

Although the predicted system throughput deviates from the measurement results when off-chip memory contention becomes severe due to increased number of L2 cache misses when all cores are active, we note that the ultimate goal of the assignment manager is to identify the best assignment among all candidates rather than produce accurate performance estimations. Intuitively, a bad assignment, i.e., one that results in low throughput, causes more cache contention among processes and thus is more likely to generate more L2 cache misses, which in turn leads to a higher off-chip memory contention level, further deteriorating system performance. Therefore, we expect that CAMP has very good fidelity when compared with an oracle performance model that takes both cache contention and off-chip memory contention into account. To test this hypothesis, we first divide 10 SPEC CPU2000 benchmarks into two groups, with group A being the high cache miss-rate group and group B being the low cache miss-rate group. We then evaluated 100 process combinations, each of which consists of 3 benchmarks randomly selected from group A and 1 benchmark randomly selected from either group A or group B. Experimental results indicate that the best assignment selected by CAMP agrees with the measurement results 96% of the time, implying CAMP is able to guide process assignment under both cache contention and off-chip memory contention.

The assignment manager solves Equation 3.7 using Newton–Raphson iteration, a standard numerical method for finding the roots of non-linear equations. In our prototype system, we implemented CAMP in C. For our 2-chip, 4-core processor, the experimental results indicate that the average performance overhead per iteration is 4 ms. Since the shortest execution time of the benchmarks we evaluated is approximately 40 s, we assume the process arrival rate to be one per 40 seconds. Given an average number of 5 iterations for each assignment, the performance overhead is 0.05% for each evaluation, or 0.15% per process combination since each process combination is associated with three assignments. In general, the number of chips under consideration influences the number of evaluations per process combination and thus the performance overhead of the assignment manager.

## 8.5 NLP-Based Power Control With Online Model Estimation

In this section, we describe the details of the power manager in PerfMax. We first present the problem formulation for power-constrained performance maximization. We then describe how to translate the target problem into a constrained nonlinear optimization problem using application-dependent performance models and a system-level power model. We also empirically determine the performance overhead of the algorithm. Finally, we explain how to adjust the parameters in the power model to account for workload variations and model inaccuracy.

### 8.5.1 Problem Formulation

Given that $N$ processes are simultaneously running on an $N$-core, $M$-chip CMP with chip-wide DVFS and $P_{max}$ is the power budget, the power-constrained throughput maximization problem can be formulated as follows:

$$\text{Maximize } \textstyle\sum_{i=1}^{N} Perf_i \tag{8.1}$$

$$\text{Subject to } \textstyle\sum_{i=1}^{N} P_i \leq P_{max}. \tag{8.2}$$

Here $Perf_i$ and $P_i$ represent the instruction throughput and the power consumption of core $i$. Note that there is a hard constraint on $P_i$.

In real systems equipped with power control techniques, power control policies are usually enforced at discrete time intervals. As in Chapter 7, we define a *control point* as a time at which control decisions are made and a *control period* as the duration between two consecutive control points. At each control point, the power manager solves Equation 8.1 subject to Equation 8.2 and uses the solution to choose per-chip frequency level for the next control period.

Since our control algorithm adjusts system performance and power consumption by tuning each chip's frequency and voltage, it is imperative to express $\text{Perf}_i$ and $P_i$ as functions of core $i$'s frequency $f_i$ (or equivalently, core $i$'s voltage). As indicated in Equation 7.3 in Chapter 7, $\text{Perf}_i$ (measured in BIPS, i.e., billion instructions per second) can be written as

$$\text{IPS}_i = \frac{f_i}{\alpha_i \cdot MPA_i \cdot f_i + \beta_i}, \tag{8.3}$$

where $\alpha_i$ and $\beta_i$ are part of the feature vector of the process running on core $i$ and $MPA_i$ corresponds to the cache miss rate associated with core $i$. Note that although $\alpha_i$ and $\beta_i$ are derived by the assignment manager through off-line characterization (see Section 3.4) in our experiment, they can also be computed online using linear regression to eliminate the need for profiling, as demonstrated in Chapter 7. Since $\alpha_i$, $MPA_i$, and $\beta_i$ in Equation 8.3 are either known a priori or easy to determine by sampling HPCs, $\text{IPS}_i$ only relies on $f_i$.

Similarly, we can determine the relationship between $P_i$ and $f_i$ following the approach proposed in Chapter 4. More specifically, we use 10 SPEC CPU2000 benchmarks along with micro-benchmarks that exercise individual architectural components with different frequencies to build a system-level model that is used to estimate power consumption based on HPC values and core frequency level. The average estimation error when building the model is $3.18\%$. Combined with online model estimation (see Section 8.5.2), the power model can accurately predict processor power consumption during runtime, thus helping the power controller to improve system performance, as indicated in Section 8.6. Therefore,

$$P_i = P_{idle} + c_1 \cdot \mathrm{L1DPS}_i + c_2 \cdot \mathrm{L2PS}_i + c_3 \cdot \mathrm{L2MPS}_i + c_4 \cdot \mathrm{FPPS}_i + c_5 \cdot \mathrm{BRPS}_i + c_6 \cdot f_i^{2.5}, \quad (8.4)$$

where $P_{idle}$ is the idle power of a core, while $\mathrm{L1DPS}_i$, $\mathrm{L2PS}_i$, $\mathrm{L2MPS}_i$, $\mathrm{FPPS}_i$, and $\mathrm{BRPS}_i$ are core $i$'s HPC values (see Section 7.3.2). In order to express $P_i$ as a function that only depends on $f_i$, we note that $\mathrm{L1DPS}_i = \mathrm{L1DPI}_i \cdot \mathrm{IPS}_i$. Given that (1) $\mathrm{L1DPI}_i$ is a process property and thus independent of $f_i$ and (2) $\mathrm{IPS}_i$ only depends on $f_i$, we can establish a one-to-one mapping between $\mathrm{L1DPS}_i$ and $f_i$. Thus, Equation 8.4 can be transformed into the following equation:

$$P_i = P_{idle} + (c_1 \cdot \mathrm{L1DPI}_i + c_2 \cdot \mathrm{L2PI}_i + c_3 \cdot \mathrm{L2MPI}_i + c_4 \cdot \mathrm{FPPI}_i + c_5 \cdot \mathrm{BRPI}_i) \cdot \mathrm{IPS}_i + c_6 \cdot f_i^{2.5}. \quad (8.5)$$

Let $b_0 = P_{idle}$, $b_1 = c_1 \cdot \mathrm{L1DPI}_i + c_2 \cdot \mathrm{L2PI}_i + c_3 \cdot \mathrm{L2MPI}_i + c_4 \cdot \mathrm{FPPI}_i + c_5 \cdot \mathrm{BRPI}_i$, and $b_2 = c_6$, we have

$$P_i = b_0 + b_1 \cdot \frac{f_i}{\alpha_i \cdot MPA_i \cdot f_i + \beta_i} + b_2 \cdot f_i^{2.5}. \quad (8.6)$$

Combining Equation 8.3 and Equation 8.6, a more concrete problem formulation follows.

$$\text{Maximize } \textstyle\sum_{i=1}^{N} \frac{f_i}{\alpha_i \cdot MPA_i \cdot f_i + \beta_i} \quad (8.7)$$

$$\text{Subject to } \textstyle\sum_{i=1}^{N} (b_0 + b_1 \cdot \frac{f_i}{\alpha_i \cdot MPA_i \cdot f_i + \beta_i} + b_2 \cdot f_i^{2.5}) \leq P_{max}, \quad (8.8)$$

$$f_i = f_j,\ \forall i,\ j \text{ on the same chip, and} \quad (8.9)$$

$$f_{min} \leq f_i \leq f_{max},\ 1 \leq i \leq N. \quad (8.10)$$

Equation 8.9 indicates cores on the same chip need to use the same frequency (and therefore voltage) level, while Equation 8.10 implies any feasible solution must fall within the range between the minimum frequency level $f_{min}$ and the maximum frequency level $f_{max}$. Note that $\alpha_i$, $\beta_i$, $MPA_i$, $b_0$, $b_1$, and $b_2$ are all positive constants that can be easily determined at each control point; the only unknown variables are $f_i, i = 1, \cdots, N$. Hence, the formulation can be treated as a constrained nonlinear optimization problem. Since DVFS has very low performance overhead (approximately 0.36% in our experiment), we do not account for it in our problem formulation. We also note that our problem formulation can be easily extended to incorporate other constraints such as temperature or handle CMPs with per-core DVFS by slightly changing Equation 8.9.

We solve the constrained nonlinear problem optimally during runtime using "NLopt" [8], an open-source library for nonlinear optimization. Our experiments indicate the nonlinear programming solver has a performance overhead of less than 0.1%, given a control period of 100 ms. More detailed analysis of solver overhead is presented in Section 8.6.4.

### 8.5.2 Online Model Estimation

The power manager is designed to achieve optimal results when the performance model has good fidelity and the power model is accurate. Since there is a hard constraint on power, accurately predicting power is of great importance to the power controller. However, it is unrealistic to expect the power model to track the real processor power very closely at all times without any feedback information. This is because (1) processes have phases and thus temporal behavior variations and (2) the power model only considers the activities of a subset of architectural components. To accommodate workload variations and compensate for inherent model inaccuracies, power manager periodically samples the local power monitor and corrects the internal power model based on the measured power data. At each control point, assuming $P_{\text{pred}}$ and $P_{\text{mesaure}}$ are predicted and measured processor power for the most recent control period, the power manager updates the power model in Equation 8.4 according to the following formula:

$$\begin{aligned}
\text{Let } e &= \frac{P_{\text{measure}} - P_{\text{idle}}}{P_{\text{pred}} - P_{\text{idle}}} \text{ and} \\
d_i &= c_i \cdot (\lambda \cdot e + (1 - \lambda)),\ i = 1, \cdots, 6, \\
P_i &= P_{idle} + d_1 \cdot \text{L1DPS}_i + d_2 \cdot \text{L2PS}_i + d_3 \cdot \text{L2MPS}_i \\
&\quad + d_4 \cdot \text{FPPS}_i + d_5 \cdot \text{BRPS}_i + d_6 \cdot f_i^{2.5}. \qquad (8.11)
\end{aligned}$$

Here $e$ represents the discrepancy between the prediction results and the measured data. A perfect power model will always have an $e$ value of 1. Depending on whether the power model under-estimates or over-estimates processor power, we scale up or down the coefficients $c_i,\ i = 1, \cdots, 6$. $\lambda$ is the constant forgetting factor with $0 \leq \lambda \leq 1$. A large $\lambda$ allows the power manager to forget the history faster. In our experiments, we set $\lambda$ to 0.8. It is worth mentioning that we do not scale $P_{idle}$ during model update because $P_{idle}$ can be accurately determined during offline model construction, therefore eliminating the need for online estimation. To verify the benefit of online model estimation, we randomly pick four processes, namely “ammp”, “bzip2”, “mesa”, and “twolf”, and run them simultaneously on our testbed. Our experiments indicate that the average estimation error is 2.35%, compared to 7.37% with a static power model. This suggests that adjusting power model during runtime can effectively reduce model inaccuracies and thus potentially help the power controller to improve performance.

## 8.6 Experimental Results

In this section, we first provide the experimental setup and the implementation details of each component in PerfMax. We then present the experimental results regarding the assignment manager, the power manager, and PerfMax, including comparisons with most related work.

### 8.6.1 Experimental Setup

Our testbed is an Intel Core 2 Quad Q6600 processor running Linux 2.6.35, with core 0 and core 1 sitting on chip 0 and core 2 and core 3 sitting on chip 1. We note that cores on the same chip must use the same voltage and frequency level, i.e., the same power state. Our processor has a 4 MB, 16-way set-associative on-chip L2 cache with a cache line size of 64 B. In addition, it supports four DVFS levels: 2.4 GHz, 2.13 GHz, 1.87 GHz, and 1.6 GHz. Experimental results indicate the frequency transition overhead ranges from 50 µs to 100 µs.

We evaluated our technique on (1) 10 SPEC CPU2000 benchmarks that compiled on our testbed (denoted as SPEC) and 6 BioBench benchmarks (denoted as BioBench) [13]. We did not consider “blast” in the BioBench benchmark suite because it is very disk I/O intensive: we make no claims for disk intensive processes. SPEC includes four floating-point programs and six integer programs, four of which have high cache miss rates. The execution time of each benchmark ranges from 40 s to 276 s. BioBench includes six bioinformatics workloads, each of which represents a different application area within the larger domain of data mining. Among the six programs, three of them have high cache miss rates.

Their execution times range from 56 s to 328 s. For comparisons in Section 8.6.3, Section 8.6.4, and Section 8.6.5, we conducted two sets of experiments. The first set includes 100 process combinations, each of which consists of four processes randomly selected from SPEC CPU2000 (referred to as SPEC process combinations). The second set includes 50 process combinations containing processes randomly selected from BioBench (referred to as BioBench process combinations).

We now introduce the implementation details of each component in PerfMax.

**Assignment Manager**: In our experiment, the assignment manager takes each process' phase transition points and its feature vector, i.e., a 4-entry tuple consisting of API, $\alpha$, $\beta$ (see Equation 3.3), and cache reuse distance program as input. For each process-to-core mapping, it then uses Newton–Raphson iteration to solve the nonlinear equilibrium equation, i.e., Equation 3.7. Finally, it finds the assignment with the highest throughput and notifies the power manager.

**Power Manager**: In our experiment, the power manager is implemented as a C program running on the same processor as the benchmarks to account for its performance and power overhead. When started, it spawns a child process to communicate with a power logging process sitting on another machine through a Unix domain socket. The child process listens for new power samples and sends them to the power manager through a FIFO (a.k.a. named pipe). In addition, the power manager accepts hardware performance counter value samples from each core's local controller through the same FIFO and stores them internally. At each control point, the power manager computes the difference between the predicted power and measured power for the last control period and updates the power model. It then uses the globally–convergent method-of-moving-asymptotes (MMA) algorithm in NLopt to solve the constrained nonlinear optimization problem and notifies the local controller of power state changes. In our experiment, the control period for the power manager is set to 100 ms. However, much shorter control periods can be used due to the low performance overhead of our solver (0.064% overhead per invocation on our testbed).

**Local Controller**: The local controller is responsible for (1) periodically collecting hardware performance counter values and sending them to the power manager and (2) tuning the frequency and voltage levels after being notified by the power manager.

1. We use PAPI 4.1.2.1 [7] for hardware performance counter sampling and experimentally determined that the overhead for accessing hardware performance counters is negligible. Due to hardware limitations, we can only sample two architectural events at a time. Given that five architectural event rates are needed for power estimation, we time multiplex architectural event sampling with a switching interval of 10 ms.

2. We control the CPU frequency and voltage via the Linux CPUFreq driver. More specifically, we use cpufreq_set_frequency() to control each chip's power state. Please note that when the frequency is changed, the voltage will be automatically adjusted by the CPUFreq driver. It is also worth mentioning that the frequency solutions provided by the power manager are continuous variables. Since the processor only supports four discrete frequency levels, we divide each control period into $K$ shorter periods (known as "scaling periods") and use first-order delta-sigma modulation to generate the frequency sequence in each scaling period to emulate continuous DVFS in a control period. In our experiment, we set $K$ to 4. Given a frequency transition overhead of 100 µs and a control period of 100 ms, the overhead of frequency modu-

lation is 0.4% even in the worst case when the frequency level needs to be changed in every scaling period and is thus negligible.

**Local Power Monitor**: 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. A power logging process samples the data acquisition card at a frequency of 10 kHz. It then computes the average power consumption and sends it to the power controller every 10 ms.

### 8.6.2 Comparisons with Prior Work

In this section, we describe the most related work for the assignment manager, the power manager, and PerfMax.

**8.6.2.1 Assignment Manager**: Our first baseline, referred to as Random, resembles a typical industrial solution that intends to balance the workload on different chips. For each assignment request, Random randomly groups processes into pairs and assigns each pair to a different chip.

The second baseline, referred to as Similarity, is a heuristic-based assignment technique proposed by Zhang et al. [118]. To the best of our knowledge, their work is the closest to ours. They claim that most high cache miss rate processes do not benefit from an increased cache size and yet aggressively occupy the cache space when running concurrently with other processes on sibling cores. Therefore, Similarity groups applications with similar on-chip cache miss ratios to run on the same chip to reduce the adverse impact of high miss ratio processes on low miss ratio processes. In our experiment, for each assignment request, Similarity identifies the top two processes with higher miss ratios and schedules them to run on the same chip, while the remaining two processes are paired together and run on the other chip.

Both Random and Similarity are ad-hoc approaches that rely on simple and incomplete observations to guide their assignment decisions. In particular, Random did not consider the performance implications of each tentative assignment in terms of cache contention or memory contention, leading to suboptimal results. Similarity tries to account for cache contention by looking at a process' cache miss ratio. However, as demonstrated in Chapter 3, cache contention level not only depends upon the miss ratio when a process is running alone, but also its cache access frequency and reuse distance histogram. Therefore, two co-running processes might generate many cache misses even if each has a low miss ratio when running alone. The quality of the results produced by Similarity depends on whether the cache miss ratio is sufficient to estimate cache contention: Similarity may achieve good results when this is true and yet do poorly (even worse than Random) when it is false, as demonstrated in Section 8.6.3. In contrast, our CAMP-guided assignment manager (referred to as CAMP hereafter when there is no ambiguity) takes all these factors into account when predicting cache contention levels, thus generating near-optimal results (see Section 8.6.3).

**8.6.2.2 Power Manager**: Our first baseline, referred to as Priority, is a heuristic-based approach for power control proposed by Isci et al. [54]. Priority represents a typical ad hoc dynamic power management scheme when feedback information is available. In our experiment, Priority first assigns priorities to different chips. It then periodically checks the current processor power and compares it with the power budget. When the power constraint is violated, Priority identifies the core with the lowest priority whose frequency level exceeds $f_{min}$ and reduces its frequency level by one step. On the other hand, when the current power is below the power budget, Priority chooses the core with the highest frequency whose frequency level is below $f_{max}$ and increases its frequency level by one step.

As a proactive approach, Priority reacts after the power constraint is violated, making it inappropriate for power control when the constraint is hard constraint. In addition, it only lowers frequency by one step even when the current power consumption is significantly higher than the power constraint, thereby producing long-lasting constraint violations. Similarly, Priority may miss precious opportunities for performance improvement because it only raises frequency level by one step per control period even when there is a large power slack. Both indicate Priority cannot adapt to workload variations. In contrast, our power manager (referred to as NLP hereafter when no ambiguity exists) solves the constrained nonlinear programming problem, determines the exact frequency levels each chip should use, and simultaneously adjusts the frequency levels of all the chips to the desired value. Therefore, NLP can achieve better performance and respond to constraint violations more quickly.

Our second baseline, known as MPC, is a power control algorithm based on model predictive control theory. To the best of our knowledge, their work is the closest to ours. MPC assumes that the power consumption of a core is a linear function of its frequency whose parameters can be determined through offline characterization. Based on this assumption, MPC formulates the power-constrained performance maximization problem as a standard constrained least squares problem whose objective function consists of two terms: the first term represents the tracking error, i.e., the difference between the total power and a reference trajectory along which the total power should change from the current power to the power constraint $P_{max}$. The second term represents the control penalty, which forces MPC to optimize system performance by minimizing the difference between $f_{max}$ and the new frequency levels. In addition, MPC uses a control vector to represent the preference among chips. In our experiment, we use a core's CPU utilization level as its weight in the control vector, as used by Wang et al. [111]. In addition, MPC uses recursive least square estimator with directional forgetting to estimate and update its internal power model based on power measurements. Compared with MPC, NLP uses more accurate performance and power models and has a smaller performance overhead, both of which lead to better performance than MPC. More detailed comparison between MPC and NLP are presented in Section 8.6.4.

**8.6.2.3 Combining Assignment and Power Control**: To the best of our knowledge, no prior work combines assignment techniques with local power control to address the performance maximization problem in a power-constrained environment. Therefore, we compare PerfMax with "Random + Priority" and "Similarity + MPC" to demonstrate the effectiveness of PerfMax.

### 8.6.3 Evaluation Results – Assignment Manager

In this section, we provide detailed evaluation results for the assignment manager (CAMP) and compare them with those of Random and Similarity (see Section 8.6.2). When

Table 8.1: Comparison Among Random, Similarity, and CAMP

| Benchmarks | Random | | Similarity | | CAMP | |
|---|---|---|---|---|---|---|
| | Accuracy (%) | (Avg., max.) degradation (%) | Accuracy (%) | (Avg., max.) degradation (%) | Accuracy (%) | (Avg., max.) degradation (%) |
| SPEC | 47 | (2.94, 23.34) | 61 | (1.46, 17.62) | 93 | (0.02, 2.07) |
| BioBench | 46 | (3.99, 21.45) | 46 | (2.4, 12.66) | 90 | (0.41, 12.66) |

evaluating the assignment manager, we compare each assignment technique with an oracle assignment manager that maximizes throughput. For each assignment technique, the accuracy is defined as the percentage of prediction results that agree with the oracle solutions, while the performance degradation is defined with regard to the throughput (measured in terms of IPS) of the oracle solutions.

Table 8.1 shows our evaluation results for CAMP and its competitors, where each row shows the results for one benchmark suite. Columns two, four, and six present the accuracies of Random, Similarity, and CAMP, while columns three, five, and seven show the average and maximum performance degradation of each technique. For SPEC, CAMP achieves the highest accuracy of 93%, compared to 47% for Random and 61% for Similarity. This indicates CAMP can accurately account for cache contention, cache miss induced memory contention, and process phases. In addition, CAMP results in the smallest average performance degradation of 0.02% (relative to the oracle results) with a maximum degradation of 2.07%, compared to an average degradation of 2.94% with a maximum degradation of 23.34% for Random and an average degradation of 1.46% with a maximum degradation of 17.62% for Similarity. The average difference between the best assignment and the worst assignment across all 100 SPEC process combinations is 5.1%, i.e., there is little throughput difference between the best assignment and the worst assignment for most of the combinations we evaluated. For the combinations where the best assignment and the worst assignment differs by more than 5%, CAMP leads to 0.04% performance degradation on average, while Random and Priority result in an average degradation of 4.85% and 2.43%, respectively. The results are similar for BioBench benchmarks. In particular, CAMP achieves the highest accuracy of 90%, while Random and Similarity achieve the same accuracy of 46%. In addition, CAMP leads to an average performance degradation of 0.41% and up to 12.66%, compared to an average degradation of 3.99% and up to 21.45% for Random and an average degradation of 2.4% and up to 12.66% for Similarity. We also note that the average difference between the best assignment and the worst assignment across all 50 BioBench process combinations is 5.72%.

We now examine the performance of the three techniques in more detail for a specific benchmark, namely "art". "art" is a low miss-ratio process when it runs alone. However, when it runs together with other processes such as other instances of "art" itself on sibling cores, it generates lots of cache misses because it is very sensitive to effective cache size. We evaluated all 45 process combinations, each of which consists two instances of "art" and two processes selected from the other 9 SPEC benchmarks. Experimental results indicate that CAMP is able to predict correctly for all process combinations, while Random incurs an average performance degradation of 4.95% with a maximum degradation of 21.01% and Similarity leads to an average degradation of 7.8% with a maximum degradation of 20.26%. Random outperforms Similarity for this benchmark because Similarity is based on the erroneous assumption that when two low miss-ratio processes are assigned to the same chip, their behavior is close to that when either runs alone. Hence, Similarity cannot correctly handle cache-sensitive processes such as "art" in SPEC and "fasta" in BioBench.

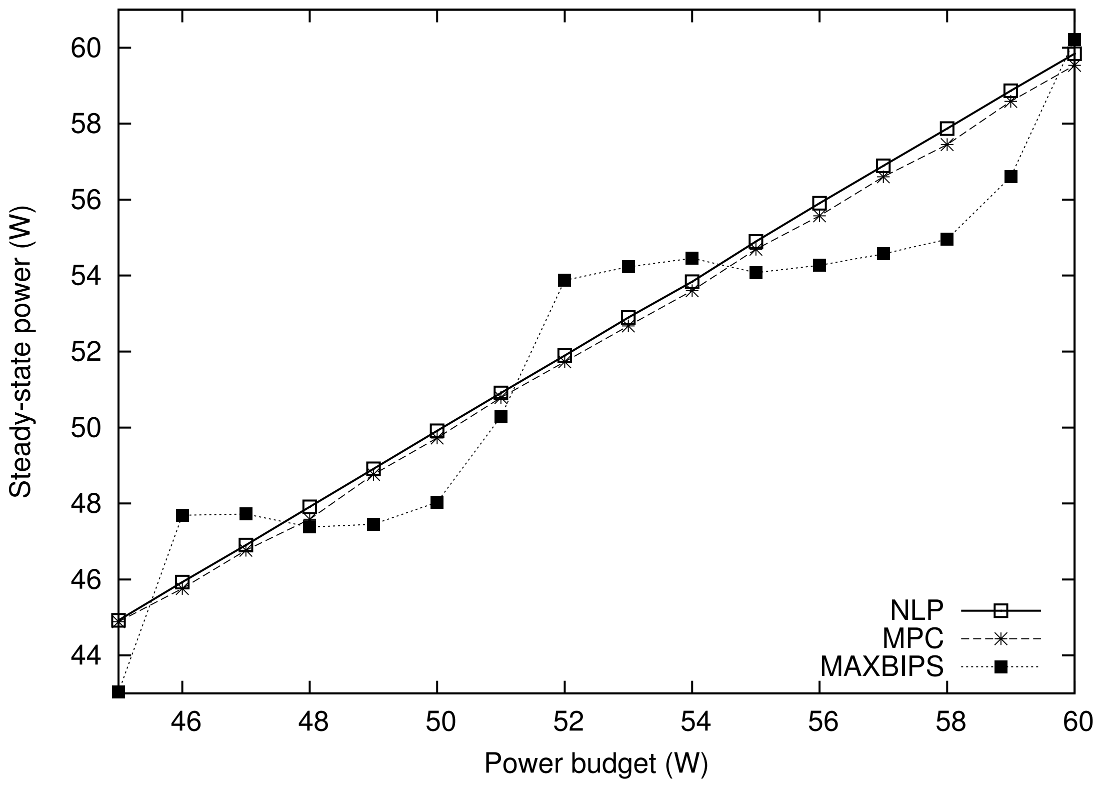


Figure 8.4: Steady-state power deviations of Priority, MPC, and NLP.

In contrast, CAMP accurately predicts the cache contention level produced by each assignment for processes with different cache behaviors, thus achieving better performance than Random and Similarity.

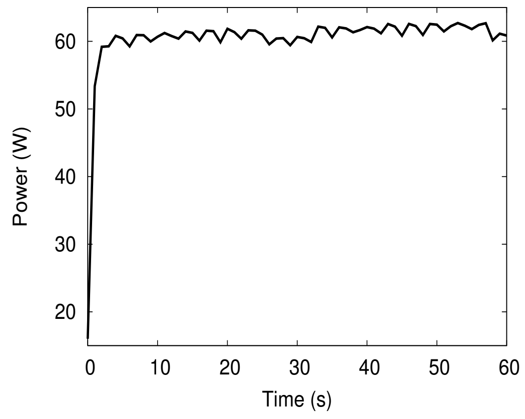
### 8.6.4 Evaluation Results – Power Manager

In this section, we compare the evaluations results for our Power Manager, NLP, with those of Priority and MPC.

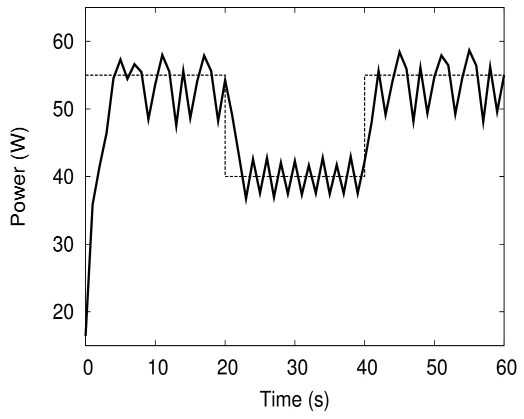
**8.6.4.1 Control Accuracy Comparison**: For each experiment, the steady-state power of a technique is defined as the average power during the entire execution, with the technique being the power controller. Ideally, the steady-state power should be very close to the power constraint and yet smaller than the constraint: the smaller the deviation is, the more power it uses and thus the higher the throughput might be (given a fixed power/performance ratio). The control accuracy is thus defined as the deviation between steady-state power and the power constraint. In order to compare the control accuracy of the three techniques, we randomly selected a SPEC benchmark “bzip2” and run four instances simultaneously on four cores. We then evaluate the steady-state power of each technique for a set of power constraints ranging from 45 W to 60 W with a step size of 1 W, as shown in Figure 8.4. The average differences between the steady-state power and the power budget are 1.47 W, 0.33 W, and 0.11 W for Priority, MPC, and NLP, respectively. Hence, both MPC and NLP are able to meet the power constraint, although NLP has a much smaller steady-state error, indicating potentially better throughput. Priority has very large steady-state error and thus cannot effectively control power. We note that the same argument also holds for other process combinations. In our experiment, the control periods of Priority and MPC are set to 1 s, the same as those used by Wang et al. [111]. We experimentally determined that MPC has an overhead of 8.315 ms. Hence, we hypothesize that they chose a control period of 1 s to restrict the power controller overhead to 0.83%.

#### 8.6.4.2 Analyzing A Typical Run:

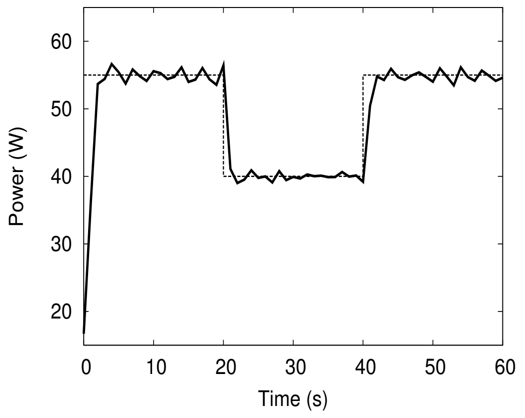
Figure 8.5 (b), (c), and (d) represent the results of Priority, MPC, and NLP for a typical run, during which “ammp” and “bzip2” are running on chip 0, while “mesa” and “twolf” are running on chip 1. We also include the original power consumption curve when no power control exists in Figure 8.5 (a) as a baseline. The power constraint is initially set to 55 W. At time 20 s, we artificially reduce the constraint to 40 W to resemble emergency situations in which the constraint must be suddenly reduced, e.g., due to hardware failures or another machine in the same cluster as the target machine needs more power and has a higher priority. At time 40 s, the power constraint is raised back to 55 W, indicating the



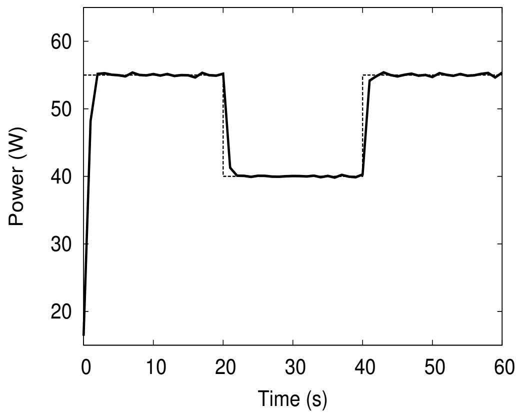
(a) Baseline



(b) Priority



(c) MPC



(d) NLP

Figure 8.5: A typical run of baseline, Priority, MPC, and NLP.

emergency has been resolved.

Priority starts with all cores running at the lowest frequency and gradually increases the frequency levels until the power consumption exceeds the power budget. Since Priority reduces frequency by one level when the constraint is violated and increases frequency by one level when there is power slack, the processor power oscillates around the power budget of 55 W because 55 W is between the power values of two adjacent frequency levels. Hence, the power consumption never settles to the power budget.

MPC achieves a better control performance than Priority because it adopts a formal

Table 8.2: Performance comparison among Priority, MPC, and NLP without guardbanding

| Benchmarks | | Max. violation (W) | | | (Avg., max.) performance improvement (%) | |
|---|---|---|---|---|---|---|
| | | Priority | MPC | NLP | Priority | MPC |
| SPEC | 87.5% | 7.51 | 3.99 | 1.26 | (6.71, 19.37) | (6.63, 16.97) |
| | 75% | 9.78 | 5.45 | 1.19 | (2.23, 8.08) | (3.54, 8.86) |
| BioBench | 87.5% | 11.52 | 7.66 | 1.44 | (7.04, 27.35) | (7.54, 29.05) |
| | 75% | 12.26 | 5.78 | 1.08 | (4.4, 17.69) | (5.03, 17.77) |

approach to systematically reduce the difference between current power and the power budget. In Figure 8.5 (c), MPC has a zero steady-state error with much smaller power spikes than Priority. However, the power fluctuations are still much larger than NLP. In particular, MPC has a maximum constraint violation of 1.63 W, while NLP only has a maximum constraint violation of 0.47 W.

NLP performs the best among all three techniques: it precisely controls the CMP power with very small steady-state errors (the maximum constraint violation is 0.47 W) by periodically solving the constrained nonlinear optimization problem and selecting appropriate frequency levels based on the solutions. In addition, it can quickly respond to changes in power constraint, as illustrated in Figure 8.5 (d), making it attractive for use in servers and large-scale datacenters.

#### 8.6.4.3 Performance Comparison:

In order to compare the control performance achieved by each technique, we evaluated the three techniques on 100 SPEC process combinations and 50 BioBench process combinations under different power constraints. Since both MPC and NLP require an initial power model to start with, we used the power model built from SPEC to control the power for BioBench in both techniques to examine how the offline power model accommodates unknown applications. Table 8.2 presents the experimental results when no power guard-

Table 8.3: Performance comparison among Priority, MPC, and NLP with guardbanding

| Benchmarks | | Max. violation (W) | | | (Avg., max.) performance improvement (%) | |
|---|---|---|---|---|---|---|
| | | Priority | MPC | NLP | Priority | MPC |
| SPEC | 87.5% | 0.0 | 0.0 | 0.0 | (16.34, 24.95) | (11.38, 20.88) |
| | 75% | 0.0 | 0.0 | 0.0 | (9.5, 14.85) | (8.95, 12.96) |
| BioBench | 87.5% | 0.0 | 0.0 | 0.0 | (18.14, 27.87) | (13.64, 29.44) |
| | 75% | 0.0 | 0.0 | 0.0 | (12.48, 23.22) | (10.91, 20.49) |

banding is used. Column one refers to the benchmark suite we used. Column two describes the power constraint we used as a percentage of the total power when all cores are running at the maximum frequency. For example, a percentage of 87.5% indicates that the power budget is set to 87.5% of the CMP power when all four processes are running at full speed. Columns three, four, and five represent the maximum constraint violations across all process combinations (100 for SPEC and 50 for BioBench). Columns six and seven show the average and maximum performance improvement when comparing NLP to Priority and MPC under the same power constraint. When the power constraint is set to 87.5% of the maximum power, NLP has the smallest violation of 1.26 W for SPEC process combinations, while the largest violations of Priority and MPC are 7.51 W and 3.99 W, respectively. When the same constraint is used, NLP results in an average performance improvement of 6.71% with a maximum improvement of 19.37% when compared to Priority and an average improvement of 6.63% with a maximum improvement of 16.97% when compared to MPC. Hence, NLP is able to achieve a better performance with smaller constraint violations given the same power budget as Priority and MPC. The same argument also holds for a different power constraint (i.e., 75%) or a different benchmark suite (i.e., BioBench). However, we note that without proper power guardbanding, all three techniques will violate the power constraint due to workload variations and inaccuracies in the power model.

Since the power constraint is a hard constraint, we need to set a power guardband and subtract it from the real power constraint so that each technique never violates the power budget. In our experiment, we use the maximum constraint violation across all the experiments in Section 8.6.4 and Section 8.6.5 as the guardband. More specifically, the guardbands for Priority, MPC, and NLP are 12.26 W, 7.69 W, and 1.46 W. Table 8.3 presents the experimental results when power guardbands are added. The columns have the same meanings as in Table 8.2. As indicated in Table 8.3, all three techniques are able to meet the power constraint when the power guardband is used. Since the results for both benchmark suites are similar, we only analyze the results for SPEC. When the power constraint is set to 87.5% of the maximum power, NLP improves performance by 16.34% on average and up to 24.95% when compared to Priority, and 11.38% on average and up to 20.88% when compared to MPC. There are three reasons why MPC is less effective than NLP. First, MPC uses a linear power model to map frequency to processor power. This is inaccurate because the power consumption is linearly dependent on frequency and quadratically dependent on voltage, which implies the power is a superlinear function of frequency, given that a frequency change is always accompanied with a corresponding voltage change. In contrast, NLP uses a nonlinear system-level power model that more accurately captures the relationship between HPC values and power consumption, leading to better prediction results and thus improved performance. Please note that the problem formulation in MPC does not permit the use of a nonlinear power model or any model that depends on variables other than frequency. Second, in MPC's problem formulation, the quadratic difference between chip frequency and $f_{max}$ is minimized to indirectly optimize performance. Hence, MPC implicitly assumes a performance model in which throughput is a linear function of frequency. However, this may lead to suboptimal results as instruction throughput depends on factors other than frequency, e.g., cache miss rates. In contrast, the objective function in NLP is the system-level throughput based on a more accurate performance model that accounts for not only core frequency but also last-level cache miss rate, thus yielding better results. Finally, MPC has a control period of 1 s, as used by Wang et al. [111] to restrict the controller performance overhead within 1%. Hence, it cannot respond to short-term temporal workload variations (e.g., on the order of hundred milliseconds) very quickly, thus leading to large constraint violations, or equivalently, worse performance. In contrast, NLP has a much smaller overhead (0.064% given a control period of 100 ms) and is thus able to use a shorter control period and adapt to temporal workload changes very quickly.

**8.6.4.4 Impact of Model Accuracy on Results**: In order to determine the performance improvement due to enhanced model accuracy, we replaced the nonlinear power model with the same linear model as that in MPC and evaluated the modified NLP (M-NLP) using 100 SPEC process combinations and 50 BioBench process combinations. The power constraint is set to 87.5% of the maximum power consumption. Without power guardbanding, M-NLP and NLP have maximum constraint violations of 4.78 W and 1.44 W, while NLP performs 3.09% better for SPEC process combinations and 1.73% better for BioBench process combinations on average under the same constraint compared to M-NLP. With power guardbanding, NLP has no constraint violations, while M-NLP violates the constraint by 0.02 W, implying that a larger power margin is needed. In addition, NLP improves performance over M-NLP by 7.52% for SPEC process combinations and 4.97% for BioBench process combinations on average. Compared with Table 8.2, 53.95% of the performance improvement is due to the improved power model for SPEC process com-

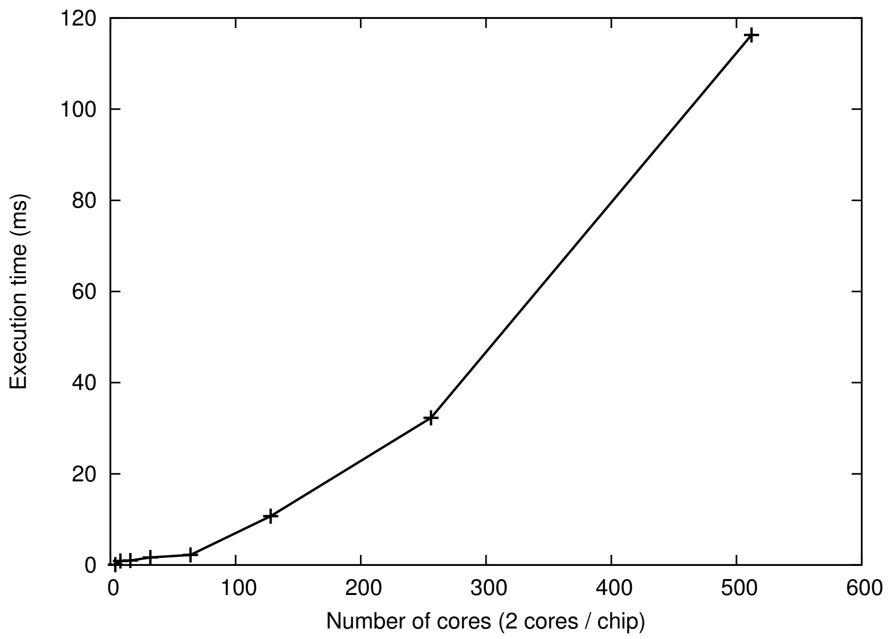


Figure 8.6: Performance overhead of the power manager.

binations, while 29.76% of the performance improvement is due to the improved power model for BioBench process combinations. Hence, we conclude that (1) the linear power model in MPC result in larger constraint violations and degraded performance compared to the nonlinear power model in NLP and (2) accurate power models are essential to power-constrained performance maximization.

**8.6.4.5 Solver Overhead Analysis**: We analyze the controller overhead of MPC and NLP and discuss their scalability. Figure 8.6 illustrates the average execution time of our solver as a function of the number of cores in the CMP, assuming two cores per chip. For MPC, we use the data reported by Wang et al. [110]. For our testbed, NLP costs 0.064 ms per invocation, i.e., a performance overhead of 0.064% given a control period of 100 ms. On the other hand, MPC takes 8.315 ms per invocation, which translates into a performance overhead of 0.8% given a control period of 1 s. For a 32-chip CMP with 2 cores per chip, NLP costs 2.21 ms per invocation, or equivalently, a performance overhead of 2.21%. On

Table 8.4: Performance comparison among "Random + Priority", "Similarity + MPC", and PerfMax without guardbanding

| Benchmarks | | Max. violation (W) | | | (Avg., max.) performance improvement (%) | |
|---|---|---|---|---|---|---|
| | | Ran + Pri | Sim + MPC | PerfMax | Ran + Pri | Sim + MPC |
| SPEC | 87.5% | 7.44 | 3.24 | 1.46 | (8.9, 26.57) | (8.38, 19.37) |
| | 75% | 7.23 | 2.96 | 0.92 | (2.81, 26.52) | (3.62, 9.91) |
| BioBench | 87.5% | 11.73 | 7.69 | 1.31 | (11.07, 31.88) | (11.88, 36.31) |
| | 75% | 9.88 | 6.53 | 0.97 | (5.49, 20.59) | (6.56, 21.11) |

the other hand, MPC takes 174 ms per invocation, or equivalently, a performance overhead of 17.4%. When considering many-core CMPs such as 1024-core CMPs with 2 cores per chip, the performance overhead of NLP and MPC are 11.6% and 452.1%, respectively. In summary, NLP has much better scalability than MPC.

### 8.6.5 Evaluation Results – PerfMax

In this section, we present the experimental results for PerfMax and its competitors, i.e., Random combined with Priority (referred to as "Ran + Pri") and Similarity combined with MPC (referred to as "Sim + MPC").

Although power guardband is required to ensure that the power constraint is always honored, Table 8.4 present the results for the three techniques when no power guardbanding is used to provide a conservative estimation of the performance improvement achievable by PerfMax with no regard to constraint violations. When the power constraint is set to 87.5% of the full CMP power, the maximum constraint violations for SPEC process combinations are 7.44 W, 3.24 W, and 1.46 W for "Random + Priority", "Similarity + MPC", and PerfMax. Under the same power constraint, the maximum constraint violations for BioBench process combinations are 11.73 W, 7.69 W, and 1.31 W for "Random + Priority", "Similarity + MPC", and PerfMax. BioBench has larger constraint violations than SPEC for

Table 8.5: Performance comparison among "Random + Priority", "Similarity + MPC", and PerfMax with power guardbanding.

| Benchmarks | | Max. violation (W) | | | (Avg., max.) performance improvement (%) | |
|---|---|---|---|---|---|---|
| | | Ran + Pri | Sim + MPC | PerfMax | Ran + Pri | Sim + MPC |
| SPEC | 87.5% | 0.0 | 0.0 | 0.0 | (19.65, 47.65) | (13.94, 23.87) |
| | 75% | 0.0 | 0.0 | 0.0 | (11.1, 25.89) | (9.69, 16.84) |
| BioBench | 87.5% | 0.0 | 0.0 | 0.0 | (22.93, 44.3) | (16.7, 34.03) |
| | 75% | 0.0 | 0.39 | 0.0 | (14.78, 29.99) | (12.32, 24.49) |

"Random + Priority" and "Similarity + MPC" because the power fluctuations in BioBench are more frequent and larger than those in SPEC on average, causing Priority and MPC to violate the constraint by a larger amount. In contrast, NLP accurately computes the exact frequency levels to keep the total power below the constraint and is thus immune to changes in workload behavior. Compared to "Random + Priority", PerfMax improves performance by 8.9% on average with a maximum improvement of 26.57% for SPEC process combinations and 11.07% with a maximum improvement of 31.88% for BioBench, given a constraint of 87.5% maximum CMP power. Under the same constraint, PerfMax improves performance by 8.38% with a maximum improvement of 19.37% for SPEC process combinations and 11.88% with a maximum improvement of 36.31% for BioBench process combinations, when compared to MPC. When the power constraint is set to 75% of the full CMP power, the results are similar.

Table 8.5 show the results when power guardbands are added. We note that the performance improvements achieved by PerfMax in Table 8.5 are the actual improvements when no constraint violations are allowed. As indicated in Table 8.5, both "Random + Priority" and PerfMax are able to meet the power constraints, while "Similarity + MPC" violates the constraint by 0.39 W when the power constraint is set to 75% of the full CMP power, indicating the necessity of a larger power margin. When the power constraint is set to 87.5% of the CMP power, PerfMax results in an average improvement of 19.65% with a maximum improvement of 47.65% for SPEC process combinations, and an average improvement of 22.93% with a maximum improvement of 44.3% for BioBench process combinations, when compared to “Random + Priority”. Under the same constraint, PerfMax leads to an average improvement of 13.94% with a maximum improvement of 23.87% for SPEC process combinations, and an average improvement of 16.7% with a maximum improvement of 34.03% for BioBench process combinations, when compared to “Similarity + MPC”. Hence, we conclude that PerfMax is able to achieve a much better control performance compared to the most closely related work. Compared with Table 8.3 where all three techniques use the same assignment scheme, CAMP further improves the performance for SPEC process combinations by 3.31% and 2.56% on average when compared to Random and Similarity. In addition, CAMP further improves performance for BioBench process combinations by 4.79% and 3.06% on average when compared to Random and Priority. Therefore, we conclude that combining assignments that lead to low contention levels and adaptive local power control techniques produces better results than those when optimizing performance within either of the two stages. The same argument holds when changing the power constraint to 75% of the CMP power.

**8.6.5.1 Deviation From Oracle Results**: We define an oracle technique as one that assigns processes and dynamically controls the power consumption in the best possible way to maximize system performance. Moreover, this technique has knowledge of the future power and performance implications of assignment decisions. Although PerfMax can significantly improve performance compared to prior work under the same power constraint, it is still inferior to the oracle technique. This is because when assigning processes, the assignment manager always chooses the assignment with the highest predicted throughput when no power control techniques are applied. However, this assignment scheme deviates from the oracle solution for two reasons: (1) CAMP may occasionally make wrong predictions and thus pick the wrong assignment and (2) when optimizing system throughput, the assignment layer and the power control layer are inter-dependent in that the local power control technique can affect the choice of best assignment in the assignment layer. Hence, an oracle technique accounts for the inter-dependency between the two layers when assigning processes. However, these two layers are decoupled in PerfMax to simplify its design. Consequently, this leads to degraded performance for cases where the oracle assignment, i.e., one that maximizes throughput, differs from the one selected by the assignment manager in PerfMax.

In order to determine how much the results generated by PerfMax deviate from the oracle solutions, we evaluate PerfMax on 100 SPEC process combinations and 50 BioBench process combinations, each of which is associated with three different assignments. For each process combination, we then compare the results produced by PerfMax with the highest throughput among all three assignments. The power constraint is set to 87.5% of the maximum CMP power. Experimental results indicate that PerfMax's results deviate from the optimal solutions by 0.39% on average and up to 3.63% for SPEC process combinations, and by 1.01% on average and up to 11.3% for BioBench process combinations. Hence, we conclude that PerfMax can achieve close-to-oracle results for different benchmarks under different power constraints.

## 8.7 Conclusions

This chapter has described PerfMax, a power control technique for maximizing performance in a power-constrained environment for multi-chip CMPs equipped with chip-wide DVFS. PerfMax consists of two major components: the assignment manager and the power manager. The assignment manager uses CAMP to accurately estimate the performance impact of each tentative assignment and selects the best process-to-core mapping, accounting for cache contention, process phases, and cache miss induced off-chip memory contention. The power manager formulates the power-constrained throughput maximization problem as a constrained nonlinear optimization problem based on accurate performance and power models. During runtime, the power manager solves the problem optimally and use the solution to guide frequency selection for different chips to control the CMP power while maximizing the system-level throughput. We evaluated PerfMax on a physical testbed with different benchmark suites. When the power constraint is set to 87.5% of the full CMP power, PerfMax achieves an average performance improvement of 13.94% with a maximum improvement of 23.87% for SPEC CPU2000 benchmarks, and an average improvement of 16.7% with a maximum improvement of 34.03% for BioBench benchmarks when compared to the most related work, while still honoring the power constraint.

CHAPTER 9

# Conclusions

In this dissertation, we have presented a comprehensive set of modeling techniques for design-time validation and run-time monitoring and optimization for high-performance computer systems such as workstations and servers. We also described both software and hardware optimization techniques that are motivated by the performance and power implications of such models.

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.

We presented FATA, a temporally-adaptive asynchronous time marching technique for fast and accurate dynamic thermal analysis. FATA improves performance by 38–138$\times$ compared to the fourth-order globally adaptive Runge-Kutta method while maintaining accuracy. We proved that step sizes of step doubling based globally adaptive fourth-order Runge-Kutta method and Runge-Kutta-Fehlberg methods regardless of initial power profile, thermal profile, and error threshold. We also analyzed the impact of temperature update functions and step size adaptation methods on accuracy and performance of dynamic thermal analysis. We concluded the combination used by FATA achieves the best performance among all candidates while maintaining accuracy.

As indicated by CAMP, last-level cache affects system performance significantly, thus making them the ideal target for optimization. We presented C-Pack, a lossless compression algorithm designed for fast, on-line cache compression using pattern matching and partial dictionary coding. C-Pack achieves a system-wide compression ratio of 61%, comparable to that achieved by most advanced related work. In addition, we reduced the proposed algorithm to a register transfer level hardware implementation, which is twice as fast as the best existing hardware implementations potentially suitable for cache compression.

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.

Finally, we propose PerfMax, a throughput optimization technique for power constrained multi-chip CMPs equipped with chip-wide DVFS. PerfMax takes advantage of accurate performance model and power model for throughput maximization across the boundary of process assignment and local power state control. In particular, it relies on CAMP to predict the performance implications of individual process-to-core mapping, considering cache contention, off-chip memory contention, and process phases. In addition, it formulates the power-constrained performance maximization as a constrained nonlinear optimization problem, solves it optimally, and uses the solutions to guide chip frequency selection. We evaluated PerfMax on a physical testbed and compared the results with those produced by the most related work. Experimental results indicate when the power constraint is set to 87.5% of the full CMP power, PerfMax achieves an average performance improvement of 13.94% with a maximum improvement of 23.87% for SPEC CPU2000 benchmarks, and an average improvement of 16.7% with a maximum improvement of 34.03% for BioBench benchmarks when compared to the most related work, while still honoring the power constraint.

# BIBLIOGRAPHY

[1] CACTI: An integrated cache access time, cycle time, area, leakage, and dynamic power model. http://quid.hpl.hp.com:9082/cacti/.

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

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

[4] Intel Core 2 Quad Process Q6600 data sheet. http://download.intel.com/design/processor/datashts/31559205.pdf.

[5] lpsolve 5.5. http://lpsolve.sourceforge.net/5.5/.

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

[7] PAPI 4.1.2.1. http://icl.cs.utk.edu/papi/.

[8] Steven g. johnson, the NLopt nonlinear-optimization package. http://ab-initio.mit.edu/nlopt.

[9] Alaa Alameldeen and David A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical Report 1500, Dept. of Computer Sciences, University of Wisconsin-Madison, April 2004.

[10] Alaa R. Alameldeen. Using compression to improve chip multiprocessor performance. Ph.d. dissertation, Computer Sciences Department, University of Wisconsin-Madison, March 2006.

[11] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proc. Int. Symp. Computer Architecture*, pages 212–223, June 2004.

[12] Alaa R. Alameldeen and David A. Wood. Interactions between compression and

prefetching in chip multiprocessors. In *Proc. Int. Symp. High-Performance Computer Architecture*, pages 228–239, February 2007.

[13] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proc. Int. Conf. Performance Analysis of Systems and Software*, pages 2–9, March 2005.

[14] Nicholas Allec, X. Chen, Robert P. Dick, Z. P. Gu, Zyad Hassan, Li Shang, Yonghong Yang, and Changyun Zhu. ISAC2: Incremental self-adaptive chip-package thermal analysis software, version 2. ISAC2 link at http://ziyang.eecs.umich.edu/projects/isac and http://eces.colorado.edu/~hassanz/ThermalScope.

[15] The ALPBench Benchmark Suite (Version 1.0). http://rsim.cs.illinois.edu/alp/alpbench.

[16] AMD athlon x2 processors. http://www.amd.com/us/products/desktop/processors/athlon/Pages/AMD-athlon-processor-for-desktop.aspx.

[17] ANSYS. http://www.ansys.com/.

[18] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive selective replication for cmp caches. In *Proceedings of Internationa Symposium on Microarchitecture*, December 2006.

[19] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon. A framework for modeling and estimating the energy dissipation of vliw-based embedded systems. *ACM Trans. Design Automation in Electronic Systems*, 7(3):183–203, October 2002.

[20] Dileep Bhandarkar and Jason Ding. Performance characterization of the pentium pro processor. In *Proc. Int. Symp. High-Performance Computer Architecture*, February 1997.

[21] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., second edition, December 2002.

[22] 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.

[23] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm.

Technical Report 124, Digital Equipment Corporation, 1994.

[24] Jeffrey Buzen. Analysis of system bottlenecks using a queueing network model. In *Proc. SIGOPS Wkshp. System Performance Evaluation*, 1971.

[25] 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.

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

[27] K. Mani Chandy, John H. Howard Jr., and Donald F. Towsley. Product form and local balance in queueing networks. *J. of the ACM*, 24(2):250–263, April 2005.

[28] 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.

[29] 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.

[30] 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.

[31] Clock modulation for Intel processors. http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/54118.htm?page=4.

[32] COMSOL Multiphysics. http://www.comsol.com/products/multiphysics/.

[33] 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.

[34] Ashutosh S. Dhodapkar and James E. Smith. Comparing program phase detection techniques. In *Proceedings of Internationa Symposium on Microarchitecture*, December 2003.

[35] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme.

*SIGARCH Comput. Archit. News*, pages 74–85, May 2005.

[36] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2006.

[37] 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.

[38] FLOMERICS. http://www.flomerics.com/.

[39] 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.

[40] P. Franaszek, J. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. In *Proc. Data Compression Conf.*, pages 200–209, April 1996.

[41] L. Yang, Haris Lekatsas, and Robert P. Dick. High-performance operating system controlled memory compression. In *Proc. Design Automation Conf.*, pages 701–704, July 2006.

[42] Erik G. Hallnor and Steven K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *Proc. Wkshp. Memory Performance Issues*, pages 9–15, 2004.

[43] Vinay Hanumaiah, Ravishankar Rao, Sarma Vrudhula, and Karam S. Chatha. Throughput optimal task allocation under thermal constraints for multi-core processors. In *Proc. Design Automation Conf.*, pages 776–781, July 2009.

[44] Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill Science Engineering, 2001.

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

[46] Sebastian Herbert and Diana Marculescu. Variation-aware dynamic voltage/frequency scaling. In *Proc. Int. Symp. High-Performance Computer Architecture*, November 2007.

[47] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002.

[48] 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.

[49] Intel core i7 processors. http://download.intel.com/products/processor/corei7/prod_brief.pdf.

[50] Intel xeon processor 5600 series. http://download.intel.com/products/workstation/323493.pdf.

[51] Intel xeon processor 7500 series. http://www.intel.com/Assets/en_US/PDF/prodbrief/323499.pdf.

[52] Intel xeon processor e5310. http://ark.intel.com/Product.aspx?id=28030.

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

[54] 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 *Proceedings of Internationa Symposium on Microarchitecture*, pages 78–88, December 2006.

[55] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of Internationa Symposium on Microarchitecture*, pages 359–370, November 2003.

[56] International Technology Roadmap for Semiconductors. http://public.itrs.net/.

[57] Ravi Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proc. Annual Int. Conf. on Supercomputing*, pages 257–266, June 2004.

[58] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of Internationa Symposium on Microarchitecture*, December 2006.

[59] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proc.*

*Int. Symp. Computer Architecture*, 2004.

[60] Nam Sung Kim, Todd Austin, and Trevor Mudge. Low-energy data cache using sign compression and cache line bisection. In *Proc. Wkshp. on Memory Performance Issues*, May 2002.

[61] 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.

[62] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proc. Int. Symp. High-Performance Computer Architecture*, February 2008.

[63] Chunho Lee, Miodrag Potkonjak, and William H. Mangione Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. http://cares.icsl.ucla.edu/MediaBench.

[64] Jang-Soo Lee, Won-Kee Hong, , and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *Proc. Int. Conf. Computer Design*, pages 184–191, October 1999.

[65] Sang-Jeong Lee, Hae-Kag Lee, and Pen-Chung Yew. Runtime performance projection model for dynamic power management. *Advances in Computer Systems Architecture*, 4697:186–197, August 2007.

[66] J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proc. Int. Symp. High-Performance Computer Architecture*, February 2006.

[67] Pu Liu, Zhenyu Qi, Hang Li, Lingling Jin, Wei Wu, Sheldon Tan, and Jun Yang. Fast thermal simulation for architecture level dynamic thermal management. In *Proc. Int. Conf. Computer-Aided Design*, October 2005.

[68] 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.

[69] Nihar R. Mahapatra, Jiangjiang Liu, Krishnan Sundaresan, Srinivas Dangeti, and Balakrishna V. Venkatrao. A limit study on the potential of compression for improving memory system performance, power consumption, and cost. *J. Instruction-Level Parallelism*, 7:1–37, July 2005.

[70] Cameron McNairy and Rohit Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[71] Andreas Merkel and Frank Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Proc. Wkshp. Power Aware Computing and Systems*, December 2008.

[72] Alistair Moffat. Implementing the PPM data compression scheme. In *IEEE Trans. on Communications*, volume 38, pages 1917–1921, November 1990.

[73] Ripal Nathuji and Karsten Schwan. VPM tokens: Virtual machine-aware power budgeting in datacenters. In *Proc. Int. Symp. High Performance Distributed Computing*, June 2008.

[74] J. L. Núñez and S. Jones. Gbit/s lossless data compression hardware. *IEEE Trans. VLSI Systems*, 11(3):499–510, June 2003.

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

[76] 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.

[77] W. H. Press, B. P. Flannery S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge University Press, 1992.

[78] Prateek Pujara and Aneesh Aggarwal. Restrictive compression techniques to increase level 1 cache capacity. In *Proc. Int. Conf. Computer Design*, pages 327–333, October 2005.

[79] 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.

[80] Quad-core AMD opteron processors. http://www.amd.com/us/products/server/processors/opteron/Pages/opteron-for-server.aspx.

[81] T. Quarles. The SPICE3 implementation guide. Technical report, EECS Department, University of California, Berkeley, 1989.

[82] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of Internationa Symposium on Microarchitecture*, December 2006.

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

[84] Jan M. Rabaey. *Digital Integrated Circuits*. Prentice-Hall, NJ, 1998.

[85] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A comparison of high-level full-system power models. In *Proc. Wkshp. Power Aware Computing and Systems*, December 2008.

[86] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Trans. Computers*, 27(3):17–29, March 1994.

[87] 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.

[88] Stephen Sherman, Forest Baskett, and J. C. Browne. Trace-driven modeling and analysis of CPU scheduling in a multiprogramming system. *Commun. ACM*, 15(2):1063–1069, December 1972.

[89] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, England, 2005.

[90] 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.

[91] Prabhakant Sinha. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.

[92] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proc. Int. Symp. High-Performance Computer Architecture*, February 2002.

[93] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proc. Int. Symp. Computer Architecture*, pages 2–13, June 2003.

[94] C. Xu, X. Chen, Robert P. Dick, and Zhuoqing Morley Mao. Cache contention and application performance prediction for multi-core systems. In *Proc. Int. Conf. Performance Analysis of Systems and Software*, pages 76–86, March 2010.

[95] X. Chen, Robert P. Dick, and Li Shang. Properties of and improvements to time-domain dynamic thermal analysis algorithms. In *Proc. Design, Automation & Test in Europe Conf.*, pages 1165–1170, March 2010.

[96] X. Chen, C. Xu, and R. P. Dick. Memory access aware on-line voltage control for performance and energy optimization. In *Proc. Int. Conf. Computer-Aided Design*, pages 365–372, November 2010.

[97] X. Chen, L. Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. C-Pack: a high-performance microprocessor cache compression algorithm. *IEEE Trans. VLSI Systems*, 18(8):1196–1208, August 2009.

[98] X. Chen, L. Yang, Haris Lekatsas, Robert P. Dick, and Li Shang. Design and implementation of a high-performance microprocessor cache compression algorithm. In *Proc. Data Compression Conf.*, pages 43–52, March 2008.

[99] X. Chen, C. Xu, Robert P. Dick, and Zhuoqing Morley Mao. Performance and power modeling in a multi-programmed multi-core environment. In *Proc. Design Automation Conf.*, June 2010.

[100] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proc. Annual Int. Conf. on Supercomputing*, pages 1–12, June 2001.

[101] 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.

[102] Ivan Sutherland, Robert F. Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, first edition, 1999.

[103] 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.

[104] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proc. Int. Symp. Computer Architecture*, June 2008.

[105] B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M.E. Wazlowski, and P. M. Bland. IBM memory expansion technology. *IBM J. Research and Development*, 45(2):271–285, March 2001.

[106] R. Brett Tremaine, T. Basil Smith, Mike Wazlowski, David Har, Kwok-Ken Mak, and Sujith Arramreddy. Pinnacle: IBM MXT in a memory controller chip. *IEEE Micro*, 21(2):56–68, April 2001.

[107] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[108] 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.

[109] Gregor von Laszewski, Andrew J. Younge Lizhe Wang, and Xi He. Power-aware scheduling of virtual machines in DVFS-enabled clusters. In *Proc. Int. Conf. Cluster Computing and Workshops*, September 2009.

[110] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. SHIP: Scalable hierarchical power control for large-scale data centers. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, pages 91–100, September 2009.

[111] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *Proc. Int. Symp. Computer Architecture*, pages 314–324, June 2009.

[112] Steven J. E. Wilton and Norman P. Jouppi. Cacti: an enhanced cache access and cycle time model. *IEEE J. Solid-State Circuits*, 31(5):677–688, May 1996.

[113] 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 *Proceedings of Internationa Symposium on Microarchitecture*, pages 271–282, November 2005.

[114] Yuan Xie and Wei-Lun Hung. Temperature-aware task allocation and scheduling for embedded multiprocessor systems-on-chip (MPSoC) design. *J. VLSI Signal Processing*, 45(3):177–189, December 2006.

[115] Yonghong Yang, Z. P. Gu, Changyun Zhu, Robert P. Dick, and Li Shang. ISAC: Integrated space and time adaptive chip-package thermal analysis. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, pages 86–99, January 2007.

[116] Keun Soo Yim, Jihong Kim, and Kern Koh. Performance analysis of on-chip cache and main memory compression systems for high-end parallel computers. In *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications*, pages 469–475, June 2004.

[117] Michael Zhang and Krste Asanovic. Victim migration: Dynamically adapting between private and shared CMP caches. Technical report, Massachusetts Institute of Technology, October 2006.

[118] Xiao Zhang, Kai Shen, Sandhya Dwarkadas, and Rongrong Zhong. An evaluation of per-chip nonuniform frequency scaling on multicores. In *Proc. USENIX Conf.*, June 2010.

[119] 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.

[120] 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.