OVER-PROVISIONED MULTICORE SYSTEMS

by

Koushik Chakraborty

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2008

Abstract

Technology scaling has provided system designers with an exploding transistor budget, far more than what was available when the core principles behind many existing commodity microprocessors were envisioned. With this tremendous growth, however, comes a whole new set of engineering challenges involving power density, thermal efficiency, and so on. In particular, the power constraint is rapidly becoming one of the first order design considerations in microprocessor designs. In the landscape of general purpose processors, such power limited designs designate a significant paradigm shift from the area limited designs of the past several decades.

This dissertation proposes a model to capture the first order impact of the power constraint in the architectural design. Denoted as the Simultaneously Active Fraction (SAF), this metric represents the fraction of the entire chip resources that can be active simultaneously, given a target power envelope. As the improvement in the energy efficiency of individual transistor devices lags behind the growth in their integration capacity, the dissertation finds that the SAF is monotonically decreasing in each successive technology generation.

In the context of such a fundamental technology trend where integrating processing cores on a single chip comes at a marginal cost compared to the cost of powering them up simultaneously, this dissertation investigates a novel multicore design paradigm: Over-provisioned Multicore System (OPMS). An OPMS is a class of multicores that *by design* provision more processing core resources than that can be kept active for their target Thermal Design Power (TDP). Since only a subset of the on-chip cores can be active at any given time, this design paradigm affords tremendous flexibility in assigning computation on processing cores, facilitating many novel techniques in this broad framework.

To demonstrate a concrete application of this framework, the dissertation proposes Computation Spreading (CSP): a new model for distributing the collective work from multithreaded appli-

cations. CSP aims to collocate similar computation fragments from different threads on the same core, while distributing dissimilar computation fragments from the same thread across multiple cores. Under CSP, on-chip cores in an OPMS are dynamically specialized via retaining mutually exclusive predictive states, and function like *hardware accelerators* for appropriately assigned computation on them, without requiring any structural modifications in the core micro-architecture. The dissertation demonstrates the effectiveness of CSP in an OPMS through rigorous evaluation of performance, energy efficiency, and several design trade-offs that are critical to understand the impact of the OPMS design paradigm in future microprocessor designs.

Acknowledgments

First and foremost, I thank my wife Sanghamitra for her immense love and support. I would not have come this far, without her unwavering zeal to inspire and motivate me. Being a graduate student herself, she has put her needs aside in numerous occasions, and guided me through the most difficult times in my graduate student career. Words cannot express my gratitude for her, and I dedicate this work to her.

I owe a great deal of what I am today to my late mother, and my father. They have instilled a sense of responsibility and dedication in my life, and this work would not have been possible without them. Even though far apart, my brother has supported me tremendously through out my life, and took care of my parents so that I can have peace of mind while pursuing my PhD. I am greatly indebted to all of them

My advisor Guri Sohi has been instrumental in the development of my research career. Guri gave me the freedom to choose problems on my own, while at the same time guiding me to make this choice. He has changed my engineering mindset of looking for specific solutions that are relevant today, to focus on a larger context, and realize the importance of intriguing research directions that can have broader impact in the future. His unique mentoring style has made a lasting impact on me, including my vision in pursuing research, and writing.

I thank several members of Guri's research group, both current and past: Sai, Jichuan, Matthew Allen, Philip Wells, Allison, and Vikas. I learned a great deal from their feedbacks on my research, writing and presentations. Sai was instrumental in the early phase of our infrastructure develop-

ment that many of us used as a starting point for our own developments. Special thanks to Philip, who not only withstood my presence in our shared office for long, but also collaborated with me in several research projects and stimulating discussions. I also thank other members of the Wisconsin Architecture group in general for providing a tremendous atmosphere for pursuing my PhD.

I thank members of my committee: Mark Hill, David Wood, Mike Swift, Mikko Lipasti and Remzi Arpaci-Dusseau. Through out my stay in Madison, I have learned a great deal from all of them. I have also enjoyed several discussions with Karu Sankaralingam, and his feedback on my presentations was invaluable.

I have had the good fortune of having many good friends in and outside of Madison. All of them have contributed in some way or other in my life, and I thank them all!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The design of a microprocessor must conjoin two fundamental technology trends: trends in semiconductor industry that underlies the basic building block of microprocessors, and trends in software that runs on these microprocessors, dictating their functional requirements. Behind many of these technology trends, the market economics plays a critical role and deeply impacts their evolution, including the shape of microprocessors.

The most profound trend in the semiconductor technology is the scaling of transistor devices, allowing to double the number of available transistors on a chip every two years [78]. In addition to this exponential growth in devices, the scaling also comes with an improvement in the performance of each device, enabling more powerful microprocessors in successive generations. Indeed, no other industry in the history has seen the phenomenal growth that the microprocessor industry has demonstrated in the past several decades [118]. This tremendous improvement from the hardware greatly influenced the software, dramatically increasing its diversity and complexity in the pursuit to exploit the performance boost from the hardware, and improve the end-user satisfaction.

While seeking performance improvement was the most dominant requirement from an economic standpoint in the past, more recently, the market economics is increasingly demanding en-

ergy efficiency, while prohibiting the rise of complexity and associated design costs, substantially altering the design considerations of microprocessors. Consequently, the traditional strategy of exploiting more transistor resources to build increasingly complex, and power hungry monolithic processor cores, is giving way to the trend of modular designs based on multicores where a single processing core is replicated multiple times on the same die.

The emergence of multicore chips brings forth many new challenges and opportunities. Exploiting the concurrency, and the computation bandwidth afforded by these multicores is a major design challenge for the future systems, encompassing multiple layers of system designers from the hardware architects to the application developers [9]. On the other hand, the superior communication bandwidth and latency between on-chip cores inspire novel techniques to manage the aggregate core resources, so as to satisfy the demand for performance and energy efficiency. Many of these techniques may have been outright infeasible in multiprocessors built from uniprocessor chips, and therefore failed to justify serious considerations before. Yet another challenge revolves around ensuring reliable execution, especially when the inherent reliability from the hardware components is expected to fall with the technology scaling [15, 17].

All these broad problem domains are important and need substantial research, but this dissertation focuses on the second problem: exploiting more resources to improve energy efficiency and performance, without putting additional burden on the power budget. Several current generation multiprocessor chips are already operating near the engineering threshold for power, based on air cooling [94, 101]. With the exponential growth of transistor devices, the ability to put more processing cores on a chip is now poised to easily outpace the corresponding ability to power up these cores simultaneously.

In such a context, this dissertation investigates a multicore design paradigm that, by design, incorporates more processing cores than the number of cores allowed to perform simultaneous

computation without exceeding the chip power budget. The dissertation calls this multicore design paradigm an *Over-provisioned Multicore System (OPMS)* [25], reflecting the unique over-provisioning of processing core resources with respect to the power envelope. Since only a subset of the on-chip cores are active at any given time, an OPMS affords tremendous flexibility in assigning computation on processing cores, facilitating many novel techniques in this broad framework.

After presenting the general framework and implementation of an OPMS, a classic application for this design is proposed and rigorously evaluated in this dissertation, which combines the scheduling flexibility in an OPMS with the superior inter-core communication in multicore systems. Unlike the traditional approach to distribute the collective computation based on threads in a multithreaded application, this dissertation proposes *Computation Spreading (CSP)* [24]. CSP aims to collocate similar computation fragments from different threads on the same core, while distributing dissimilar computation fragments from the same thread across multiple cores. Under CSP, on-chip cores in an OPMS are dynamically specialized via retaining mutually exclusive predictive states, and function like *hardware accelerators* for appropriately assigned computation on them, without requiring any structural modification in their core micro-architecture.

1.1 Motivation

This section discusses the two key motivations behind this dissertation work: challenges and opportunities in multicore systems, and the demand for energy efficiency.

1.1.1 Challenges and Opportunities in Multicore Systems

The emergence of multicore systems in the recent years, which has now become ubiquitous across the entire computer industry, has spurred several new challenges within the context of delivering

energy efficiency and performance improvement. The additional processing cores introduce new perspective in system design as efficient use of aggregate resources is becoming a major engineering challenge. Earlier, in a uniprocessor domain, all resources including caches and memory bandwidth were solely provisioned for the single computation executing on the core. However, in a multicore systems, multiple computations compete for the shared resources, and appropriately allocating resources among these computations has a large impact on overall performance and fairness of the system [27, 49]. Furthermore, the many fold increase in the pool of aggregate private resources like L1 caches and branch predictors, also imposes intriguing design challenges in terms of using them efficiently *as a whole*.

Despite these design challenges, multicore systems also offer new opportunities that compel a fresh look into the fundamental principles of the system design. Most importantly, multicore systems offer superior bandwidth and latency in communication between on-chip cores, as compared to equivalent multiprocessors built from uniprocessor chips [23]. Co-operation among different on-chip cores becomes more practical in these multicore platforms, enabling superior performance and reliability of a wide range of applications [24, 38, 121].

1.1.2 Energy Efficiency

The proliferation of digital systems have led to an ever increasing global energy cost attributed to them. A recent study indicates that the energy cost from servers in data centers have doubled from 2000 to 2006 in US, and now contributes 1.5% of the overall energy consumption, equivalent to the household energy consumption in US [112]. This large rise in energy cost has now crossed the threshold where the operating cost of computers are now poised to overshoot the cost of their ownership [31]. Although many aspects of an entire system play key roles behind this surge in the energy expenditure, a substantial fraction of this growth is attributed to the rising power consump-

tion of microprocessors [31].

The increasing power consumption in a microprocessor introduces a serious problem as the heat generated in a small form factor must be removed briskly, so as to remain within the operating temperature range, necessary for the reliable execution of the devices. However, air cooling used by commodity microprocessors—prevalent in both desktop and server platforms—has an engineering limit of the amount of heat it can remove [55], dictating the power budget. Moreover, economic considerations to save operating cost further restrict the power envelopes in current multicores [91].

With the doubling of transistor devices in microprocessor chips, their power consumptions also increased, but did not double in successive generations, as substantial benefits in energy efficiency was drawn purely from technology scaling [15]. As the size of the transistor scaled with successive technology generations, the energy efficiency also improved from voltage and capacitance scaling. But in recent years, voltage scaling has almost stopped due to various constraints from the device engineering [16, 29], limiting the energy efficiency gained from technology scaling. Consequently, system designers are now expected to focus on various techniques at the architectural level to control the power consumption of microprocessors [56].

An indication to fulfill this expectation is seen in several current generation microprocessors that show a trend in increasing functional diversification where a diverse pool of micro-architectures, performing specialized functions, are integrated [55]. Several research proposals have also appeared to advocate more of such specialized components [5, 28, 39]. Broadly termed as *hardware accelerators*, these special purpose circuit components excel in both power and performance metrics. Compared to general purpose components, such custom designed components often deliver orders of magnitude improvement in power-performance for the specific computation they were intended [56]. Several microprocessors already include such hardware accelerators (e.g., the cryptographic unit in Sun's Niagara [82]), and the ITRS roadmap envisions a continuation of

this trend [55].

1.2 Thesis Contributions

This dissertation makes several contributions in the context of energy efficient multicore designs. First, it defines an abstract model for the power constraint in microprocessors, and demonstrates its use in reasoning about different multicore configurations in the early phase of their designs. Second, in light of the increasing power constraint in successive technology generations, this dissertation proposes Over-provisioned Multicore System (OPMS), which by design provisions more core resources on a multicore than the maximum possible simultaneously active cores for its target power budget. Such a design is in a stark contrast with the conventional multicores that target full utilization of core resources. Nevertheless, this design paradigm captures the fundamental technology trend where integrating cores comes at a much lower marginal cost than powering them up. Third, this dissertation proposes Computation Spreading: a new model for distributing collective computation from multithreaded applications, and delves into a rigorous evaluation of its performance and energy efficiency.

1.2.1 Simultaneously Active Fraction

While the power constraint is rapidly growing in importance for microprocessor designs, estimating the extent of this constraint at an early phase of a microprocessor design cycle imposes a serious challenge [19]. To mitigate this challenge, this dissertation develops an abstract model for the power constraint by expressing it in a more tractable area metric: *Simultaneously Active Fraction (SAF)*. SAF of a chip is defined as the maximum fraction of on-chip devices that can be active at the same time, while still remaining within the chip's power budget, also known as the

Thermal Design Power (TDP).

Combining the expression derived for the SAF, and the expected transistor characteristics for several upcoming technology generations [55], the dissertation finds that the SAF is likely to shrink substantially in future microprocessors, limiting the fractional resources that can be active at any given time. Using analytic models, two concrete applications are then presented to illustrate the impact of shrinking SAF in multicore design considerations.

1.2.2 Over-provisioned Multicore System

In the light of various technology trends, this dissertation investigates a novel multicore design paradigm, which embodies a class of multicores that by design integrates more cores than allowed by its power budget. Throughout this dissertation, such a multicore is referred as an *Over-provisioned Multicore System (OPMS)* [25]. From a system design perspective, the OPMS design paradigm designates a substantial shift in design considerations from the past, where increasing the processing core utilization was of prime importance. Instead, this dissertation proposes a novel multicore paradigm where power is considered more critical, enabling a dynamic distribution of the available power budget among the on-chip cores. Since only a subset of the available cores is engaged in computation, the inherent flexibility of assigning computation on the cores facilitates various techniques to dynamically specialize the on-chip cores, yielding benefits in both runtime and energy efficiency.

1.2.3 Computation Spreading

Finally, this dissertation proposes *Computation Spreading* that combines the scheduling flexibility of an OPMS, and the superior communication in multicore systems, to build a co-operative framework among on-chip cores to perform the collective computation from multithreaded appli-

cations [24]. Conventional multicores distribute the computation on such applications based on threads, resulting in a poor utilization of predictive structures that exploit properties of instruction streams (e.g., instruction caches and branch predictors). In contrast, CSP localizes similar computation from multiple threads onto a single core, while distributing dissimilar computation from a single thread onto multiple cores. Consequently, on-chip cores in an OPMS retain mutually exclusive predictive states, and function as *hardware accelerators* when appropriate computations are assigned on them that best exploit those predictive states.

1.3 Dissertation Outline

Rest of the chapters in this dissertation are organized as follows. Chapter 2 presents the model for the Simultaneously Active Fraction, with a detailed discussion on the background concepts of area and power constraints, the expected trends of SAF, and its application in the early phase of a microprocessor design. Chapter 3 introduces the primary contribution of this dissertation — Overprovisioned Multicore Systems (OPMS), presents a broad discussion on its design aspects, and the specific implementation adopted for the rest of the dissertation. Chapter 4 illustrates Computation Spreading with its motivation in the context of multicore designs and OPMS, and the specific flavor of CSP adopted in this dissertation: separating OS and user computation. Chapter 5 describes the detailed methodology used throughout this endeavor. Subsequently, Chapter 6 presents a rigorous evaluation of performance and energy efficiency of the OPMS. Finally, Chapter 7 summarizes the main contributions detailed in the previous chapters, and outlines possible research directions stemming from this dissertation.

Chapter 2

Simultaneously Active Fraction

Driven by the increasing demand for energy efficiency at the system level, power consumption is rapidly emerging as one of the first order design considerations in modern microprocessors [80]. In the general-purpose microprocessor design space, power limited designs designate a significant paradigm shift from the area limited designs of past several decades [48]. While several architectural intuitions deal well with area, allowing a hierarchical and modular design strategy to be effective, considering power consumption at the early phase of a similar design flow imposes a major challenge.

Although one can accurately estimate transistor level power consumption, transcending that estimation at the chip level requires aggregation over many levels of abstraction, substantially complicating the assessment. For example, power consumption of a single transistor may vary widely depending on the micro-architectural circuit component it belongs to. Even within a given small circuit component, the power consumption may vary based on its topological placement, activity and physical characteristics (e.g., width). Consequently, it is practically impossible to accurately estimate the power consumption of a system at an early phase of its design cycle [19].

Given this technical difficulty, this chapter develops an abstract model for the engineering con-

straint imposed on a microprocessor design due to its power consumption. The key objective in this pursuit is *not* to come up with an accurate estimation of power consumption of a particular design, but to develop a first order model of chip level power constraint, and investigate the technology trend of this metric. This dissertation develops this model for power constraint by expressing it as an area constraint. This chapter defines this expression as a metric, referred as the *Simultaneously Active Fraction (SAF)*: the maximum fraction of on-chip devices that can be active at the same time, while still remaining within the chip's power budget, also known as the *Thermal Design Power (TDP)*.

The chapter begins with a background discussion of area and power constraint in the context of a microprocessor design in Section 2.1. Next it develops the first order model of SAF in Section 2.2, and presents quantitative trends of SAF based on prevailing technology characteristics. Next, it describes the role of SAF reduction in future microprocessors in Section 2.3. Then, the chapter presents two concrete examples to demonstrate the use of SAF in the early phase of power constrained multicore designs. First, it incorporates the power constraint in a recent analytic model for determining speedup from multicore chips in Section 2.4. Second, it shows the application of SAF in designing multicores for multithreaded server applications in Section 2.5. The chapter concludes with a summary in Section 2.6.

2.1 Area and Power Constraint

This section presents the background concepts of the two key engineering constraints in a microprocessor design: area and power. Fundamentally, the area constraint is well defined, and it is *statically* met by the collective presence of devices, each of them occupying a certain fraction of the area. In contrast, the power constraint is satisfied dynamically, as the mere presence of a device is not sufficient to determine its power consumption. Different ensemble of devices can

operate at a wide range of power levels at different times, lending a dynamic nature to the total power consumption. Even at a fixed total power, the selection of devices consuming power may differ. A detailed discussion of these constraints, and the design implications of satisfying them concurrently—a necessary requirement for microprocessor designs—is presented next.

2.1.1 Area Constraint

The area of a micro-processor chip is a function of the size of devices, and the total number of devices integrated onto it. The total area constraint can then be expressed as:

$$A_{die} = \sum_{i=1}^{N} D_i$$

where A_{die} represents the die area constraint, N is the total number of devices integrated on a die, and D_i is the size of an individual device i. The total available area for a given implementation is determined by various factors such as form factors, expected die yield, wafer size, and economic consideration for the target market place [55]. Despite the complex interaction of several factors, ITRS roadmap predicts a constant die-size for the foreseeable future, especially for high-performance microprocessor units [56].

After the total area constraint or die size is established, it must then be met by the aggregate sizes of integrated devices. As the equation indicates, one can either reduce the size of each device or simply remove the required number of devices to satisfy this constraint. However, within a given technology generation, there is a limit for the minimum device size. Once this minimum size is attained for all the integrated devices, violation of the area constraint can only be rectified by downsizing the device count. Of course, performance considerations may dictate larger device sizes in various critical micro-architecture components [118], thereby requiring further reduction

in the aggregate devices.

2.1.2 Power Constraint

Similar to the area, the total chip power is an aggregation of power from each device. The total power is constrained by the cost-effective cooling capacity of a high volume micro-processor designs [53]. More recently, the demand in energy efficiency is imposing severe restrictions on the chip power budget [91]. Collectively, these two trends indicate a constant chip power budget, even under optimistic assumptions. This power constraint can be expressed as:

$$Power_{die} = \sum_{i=1}^{N} P_i$$

where $Power_{die}$ is the chip-level power constraint, P_i is the power consumed by the device i, while N is the total number of devices as before. While the minimum area of a transistor device (D_i) is set by the technology generation, there is no such set minimum for device level power consumption: devices can exist on a die without consuming power. Within a given implementation, the power consumption of on-chip devices also tends to vary substantially [19, 20]. Thus, unlike the area constraint, where the presence of a device results in a reduction of total available area, the mere presence of on-chip devices may not reduce the available power budget. This key distinction allows a variety of design strategies to meet the chip level power constraint (e.g., half of the total devices consuming the entire power budget, while the rest are in zero power mode).

2.1.3 Satisfying Area and Power Constraints

A viable microprocessor design must concurrently satisfy both its area and power constraint as determined by the form factor, energy efficiency requirements, economic considerations and so

forth. At a high level, the area constraint limits the number of aggregate devices on a single chip, which in turn will limit micro-architecture components (e.g., processing cores, caches, special purpose accelerators) that can be realized within the die area. On the other hand, for each individual device power level, the chip power constraint, typically referred as its Thermal Design Power (TDP), limits the total number of devices consuming power at that level. As the device power level varies, the total number of devices allowed at that level will also change. Furthermore, the ability to maintain devices at (or near) the zero power level, allows different collection of on-chip devices, and the corresponding micro-architecture they constitute, to consume power at different times.

Therefore, the power constraint of a given implementation can be dynamically satisfied by different subsets of on chip devices. Any given subset of such devices also have a certain area, but their aggregate area is a fraction of the total available chip area. This dissertation defines such a fractional area as the *Simultaneously Active Fraction (SAF)* of a chip. Formally, the SAF of a chip designates the largest fraction of the on-chip devices, and their collective area thereof, which is allowed to remain simultaneously active and consume power without violating the chip TDP.

The SAF captures the chip TDP in a more tractable metric for the architects: area. The maximum value of an SAF is 1, where all the on-chip devices can actively consume power. In practice, typical implementations have lower values of SAF. Therefore, from the power perspective, the devices are *over-provisioned* as the power constraint bars concurrent active power consumption in all of them. A key contribution of this dissertation is to build a framework to understand the concept of over-provisioning in the context of multicore designs and their TDP. However, this chapter focuses on a thorough understanding of the SAF, its technology trends, impact on power conservation, and applications of SAF during the early phase of microprocessor designs.

2.2 SAF: First Order Model

This section develops the model of SAF, and inspects the technology trends of it. The key objective behind inspecting the SAF trends is to provide an area perspective of the power constraint in future microprocessor designs.

2.2.1 Analytic Model

To develop the SAF model, this section begins with a discussion on the fundamental transistor level power characteristics and their recent trends, which are crucial to understand the SAF trends. A basic CMOS transistor device has two different modes of power consumption: *dynamic power* and *static power*.

2.2.1.1 Trends in Dynamic Power

Dynamic power is consumed through the switching of a transistor, which results in charging or discharging of the device. Because of the charging and discharging, a transistor can be modeled as a capacitive load, driven by the supply voltage of the transistor [118]. Based on the switching frequency (f), the dynamic power of a device is given by the following equation:

$$Power_{dynamic} = C \cdot V_{DD}^2 \cdot f$$

where C is the capacitive load and V_{DD} is the supply voltage.

To understand the transistor level trends in dynamic power, one can simply investigate different parameters in the equation shown above. Clearly, the most dominant factor is scaling down the supply voltage V_{DD}, which yields a quadratic reduction in power. In the semi-conductor technology, *scaling* can have different implications in different contexts. For example, scaling of voltage

and capacitance implies a reduction of these parameters, while frequency scaling usually implies an *increase*. However, for the ease of comprehension, scaling will be used to explicitly indicate a reduction throughout this chapter.

Historically, the supply voltage was scaled, almost in tandem with the device feature size, after the semi-conductor industry adopted constant electric field with technology scaling [15]. However, more recently, the scaling of voltage has slowed down considerably due to the reliability concerns in sub-micron devices, and the inability to scale the threshold voltage [47]. Several leading fabrication vendors are expecting a flat supply voltage in the near future [16, 29]. This diminishing return in power reduction from voltage scaling is the most fundamental reason behind the increasing power consumption in microprocessor chips [15].

On the other hand, scaling the capacitance gives a linear reduction in dynamic power. Although, the capacitance was scaled almost linearly with the feature size, in recent years it is showing signs of slowing down [76]. To understand this slowdown, let us inspect the governing equation for the transistor capacitance:

$$C_{total} = C_{intrinsic} + C_{parasitic}$$

Fundamentally, the intrinsic capacitance ($C_{intrinsic}$) is dependent on the gate length, and scales down as the device size shrinks, but the parasitic component ($C_{parasitic}$) is independent of the gate length [104]. In past technology generations, the intrinsic capacitance made up the bulk of the total capacitance [104], thereby resulting in a near linear reduction of the total capacitance with the technology scaling. On the other hand, the parasitic capacitance is increasing with the technology scaling, and is rapidly becoming a significant fraction of the total capacitance [76]. Consequently, the overall capacitance scaling is now slowing down.

Unlike the supply voltage and load capacitance, the frequency has been increasing in successive

Characteristics	Ideal/Theoretical	Real
Supply Voltage	Linear scaling (0.7) [15]	Marginal scaling [16, 54]
Capacitance	Linear scaling (0.7) [15]	Scaling slowing down, sub-linear [76]
Frequency	43% (1/0.7) increase [15]	Target 17%, likely smaller in future multicores [56]
Dynamic Power	50% improvement [15]	Limited improvement [16]

Table 2.1: Technology scaling trends of Dynamic Power

technology generations. The scaling of transistor devices leads to a reduction in its intrinsic switching delay, allowing a commensurate increase in the clock frequency [118]. Increasing the clock frequency was also one of the most effective techniques for improving performance. Moreover, the increase in power consumption from the rise in frequency was already more than compensated by the scaling of supply voltage and capacitance, resulting in an overall improvement in device level energy efficiency [15]. However, as the performance improvement from increasing the processor clock frequency started to slowdown, with the concurrent trend in diminishing returns in power consumption improvement from the supply voltage, aggressively increasing the frequency fell out of practice in modern microprocessor implementations. The ITRS predicts only limited increase in frequency for logic components [56], even for high performance microprocessors.

Table 2.1 presents a summary of the above discussion. Combining all these trends, it appears that the improvement in dynamic power is likely to slowdown dramatically over the next several technology generations.

2.2.1.2 Static Power Trends

In contrast to the dynamic power, static power from a transistor is consumed due to a small continuous discharge mechanism, otherwise known as *leakage*. There are several modes of leakage, but they can be broadly classified as sub-threshold, gate, and reverse-biased drain-and-source substrate junction band-to-band-tunneling (BTBT) [1, 93]. These leakage mechanisms have strong depen-

dence with several second order device parameters (e.g., oxide thickness, doping concentration, temperature), in addition to the supply and threshold voltage of these devices. Because of these second order effects, it is more challenging to accurately estimate transistor level trends in static power, and its overall contribution to the TDP.

However, the overall static power due to leakage has been steadily growing over the past several technology generations [22, 93]. Sub-threshold leakage, which has a exponential dependence with the threshold voltage, increased rapidly with the scaling of threshold voltage, and typically dominated the leakage currents [47]. More recently, threshold voltage has almost stopped scaling to counter this sub-threshold leakage, but the scaling of other device parameters (e.g., oxide thickness) caused a rise in gate leakage and BTBT. Subsequently, both these two leakage mechanisms now constitute a substantial fraction of the total leakage [1]. Despite this steady rise, several techniques have shown tremendous potential in mitigating the leakage (e.g., high-κ metal oxides, double gate devices). Within a technology generation, circuit designers choose the optimum techniques for controlling the leakage energy, determined by the optimization of the overall power-performance of the system [47].

Collectively, the trends in dynamic power and static power imply that the total power from individual devices (referred as P_{Device} next) is likely to improve very slowly in the next several technology generations.

2.2.1.3 SAF

The total power of a given micro-processor is simply an aggregation of the power consumed by each device on the chip. As the physical characteristics, activity and various key parameters of transistors vary widely across a given integrated circuit, the first-order model adopts a uniform transistor characteristics. Consequently, the model assumes that the power consumed by each in-

dividual transistor is identical across the entire chip. Therefore, at the system level, total aggregate power consumption is obtained by the aggregation of individual device power over the total number of active devices. A more detailed discussion on the justification and limitations of this assumption is presented later in Section 2.2.3.

$$\begin{aligned} Power_{chip} &= Power_{Device} \cdot \textit{Number of Active Devices} \\ &= Power_{Device} \cdot (SAF \cdot \textit{Total Device Count}) \end{aligned} \tag{2.1}$$

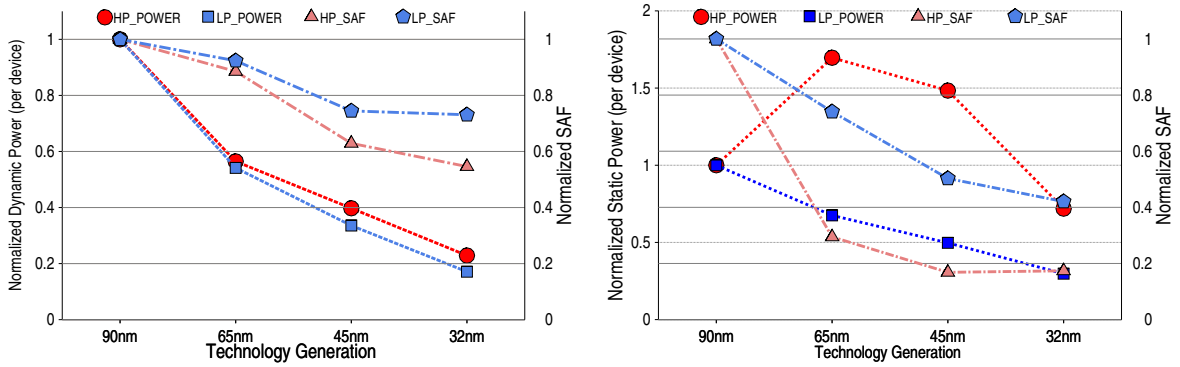
The *Total Device Count* is exponentially increasing with each new technology generation, but the *Individual Device Power* (when active) is not decreasing at the same rate. Meanwhile, the total $Power_{chip}$ remains relatively constant due to the limitations of cost-effective cooling and packaging techniques [53]. Thus, the SAF must decrease. Qualitatively, this reduction of SAF in successive technology generations designates the increasing power constraint at the early phase of a microprocessor design.

In order to provide a system-level perspective for architects, the next section quantitatively analyze these trends across different technology generations and shows the growing power constraint illustrated through the continuous reduction of SAF.

2.2.2 SAF Trends: A Quantitative Analysis

Several technology assumptions are necessary for a quantitative analysis of the SAF trend. For the sake of simplicity, let us consider the performance characteristics of two typical N-MOSFET devices, one designed for high-performance (HP), and the other for low power (LP) [54]. HP devices use a low threshold voltage and a nominal supply voltage, while the LP devices use a higher

threshold voltage and a scaled supply voltage. Together they encompass the range of devices, with diverse operating conditions, typically seen in high-performance microprocessors [54]. Furthermore, the estimation assumes that the entire chip is built with these devices, and that the chip size remains constant, resulting in a doubling of the number of devices every generation [53]. Based on the power consumption of a single device, determined using the MASTAR tool [57], the SAF is computed at each technology generation as the fraction of the total devices that account for a given TDP (which is assumed to remain constant across technology generations). The results, normalized to the 90nm generation, are shown in Figures 2.1(a) and 2.1(b).



(a) *Dynamic power trend and SAF due to dynamic power exclusively.*

(b) *Static power consumption trend per device and SAF due to static power exclusively.*

Figure 2.1: SAF trends

The trend in dynamic power, and the resulting SAF when considering dynamic power alone, is shown in Figure 2.1(a). The figure assumes a 17% improvement in switching frequency in successive generations based on the ITRS guidelines for logic components [54]. HP_POWER and LP_POWER show the power consumption of a single HP and a LP device, respectively, while HP_SAF and LP_SAF show the SAF for HP and LP devices, respectively. The improvement in dynamic power efficiency is less than the improvement in device integration capacity, in both types of devices. Consequently, Figure 2.1(a) shows a fall in the SAF in each successive generation. The

decline in the SAF is more pronounced in the HP devices than LP due to relatively smaller gains in the device level power consumption and energy efficiency. In summary, it is expected that the dynamic power, even when considered in isolation, will continue to limit ways to use aggregate resources due to the gradual reduction in the SAF.

Static power due to leakage is an additive power component. Figure 2.1(b) shows the trend in static power per device, along with the SAF when considering only static power. For high performance devices, a dramatic reduction in SAF is observed as, unlike dynamic power, the static power of individual devices is *rising* in successive technology generations, till the 32nm technology node. At the 32nm technology, the adoption of FINFET devices is expected to substantially reduce the sub-threshold leakage, leading to an estimated improvement in static power compared to previous technology generations [7]. Technology experts expect that leakage-induced static power will be mitigated by employing various circuit and process manufacturing techniques such as high κ-dielectric [54], FINFET devices [7], and so forth. However, it is still likely to pose additional constraints on aggregate resource usage.

While the analysis above provides a relative trend, the actual SAF of a real implementation depends on various other factors not considered in this analysis. These factors include several device-level parameters, circuit design techniques, purpose of the transistors (logic vs. memory), percentage of transistors on latency-critical paths, the temperature and activity distribution, among others. Regardless, in the absence of a dramatic alteration of device properties, this downward trend of the SAF will persist and thus processor designers will be forced to develop innovative techniques to reduce the SAF.

2.2.3 Limitations

The primary purpose of the SAF model is to provide a first order impact of power constraint in microprocessor designs across multiple technology generations. To this effect, the model assumes a uniform power consumption of transistor devices across the entire chip. In reality, on-chip transistors in a typical implementation operate at a wide range of power levels. Moreover, the power consumed in driving the wires was not considered explicitly. A more complicated model that assumes certain distribution of power consumption among on-chip devices may give a better absolute measure of SAF. However, the applicability of such a complex model becomes limited as a different microprocessor implementation may substantially alter the underlying assumption about the distribution of power consumption among on-chip devices.

Instead, the simplified model proposed in this work attempts to develop the intuition behind the impact of power constraint in future microprocessor designs. The quantitative trend illustrated in the previous section shows the necessary levels of SAF reduction required through architectural means, after accounting for the energy efficiency obtained from the technology scaling exclusively.

2.3 Targeting SAF for Power Reduction

Based on Equation 2.1, this dissertation broadly classifies power reduction techniques into two classes that target the individual device power or the SAF, respectively, and discuss each in turn.

2.3.1 Targeting Per-Device Power is Insufficient

Voltage Scaling The most effective technique for chip power reduction in past technology generations has been voltage and frequency scaling (VFS). Within a technology generation, voltage scaling, in combination with the necessary frequency scaling, yields a cubic reduction in individual

device power [118]. However, supply voltage reduction is reaching a plateau in current process technology and is unlikely to continue in future technology generations. For example, voltage scaling exponentially increases the *fault rate* due to particle strikes [122], which is already a serious reliability concern. In addition, with the limited flexibility in lowering the threshold voltage due to leakage concerns, maintaining a significant voltage over-drive is becoming mandatory for tolerating increasing process variation in scaled devices [18]. Therefore, this dissertation makes the fundamental assumption that voltage scaling will soon become infeasible—a conclusion shared by others as well [16, 29, 35].

Frequency Scaling Frequency scaling still remains a practical means for reducing dynamic power by simply reducing the clock rates of processing cores and caches[1]. The expectation is that frequency scaling can reduce power without incurring a significant performance impact, since several circuit components expend energy even when they are not doing useful work (for example, while waiting on a load). However, frequency scaling, in the absence of voltage scaling, is unable to fulfill this expectation for several reasons. First, widely employed techniques like aggressive clock gating already target the same power saving opportunities as frequency scaling, and save a significant amount of the overall power (up to 40% by one estimate [58]). Second, since frequency scaling by itself has no effect on static power, the impact on total power saving is further reduced. Thus, *the relative power reduction lags the relative frequency scaling*. Both compute bound and memory bound workloads are negatively impacted by frequency scaling due to this lag in power savings.

Compute bound workloads spend a major fraction of their execution cycles in generating new values and satisfying their inherent data dependencies, with limited stalls from their memory ref-

[1]Frequency scaling does not restrict the active computation area, but affects allowable SAF by reducing individual device power.

erences. Core pipeline logic such as functional units and register files remain busy and heavily utilized, and a high throughput from these micro-architecture components directly translates into a higher overall performance in these applications. Frequency scaling decreases the throughput of the core pipeline logic, thereby decreasing the pipeline performance. Therefore, the performance of compute bound workloads can degrade linearly with frequency scaling, defeating the purpose of improving energy efficiency metrics (e.g., energy-delay product).

Unlike compute bound workloads, the performance loss of memory-bound applications is more resilient to frequency scaling [68]. These workloads spend a majority of their execution on waiting for memory references [3], resulting in a low IPC (instruction per cycle), and observe lower activity in their core pipeline logic. Consequently, decreasing the pipeline throughput *does not* degrade the runtime linearly, potentially yielding a benefit in energy efficiency. However, because of lower utilization of core pipeline logic from their low IPC, substantial dynamic power from the core logic can be saved by aggressive clock-gating [58]. Savings in the dynamic power from the total power budget, increases the relative contribution from the static power component, which remains unaffected by frequency scaling. Thus, the effectiveness of frequency scaling in saving total chip power is dampened, leading to a reduction in energy-efficiency benefits in memory bound workloads as well. Collectively, this dissertation finds that frequency scaling alone is detrimental to power-performance characteristics of several workloads (details in Chapter 6).

In a multicore processor, frequency scaling can further be applied on a per-core basis. However, per-core frequency scaling has two additional pitfalls. First, it can lead to unpredictable performance characteristics because of the resulting asymmetric execution substrate [10]. Second, in multithreaded workloads, which often require thread synchronization, threads executing on a slow core can negatively impact the performance of *all* threads (e.g., by increasing the lock wait time), causing loss of overall performance and energy [87]. To avoid these problems in com-

plex multithreaded applications, this dissertation focuses on frequency scaling the entire multicore (Chapter 6).

2.3.2 Existing SAF Reduction Techniques

Given these practical challenges of reducing the individual device power without dramatic performance degradation, SAF reduction is likely to be increasingly important as a key power reduction technique in future technology generations. While it will be important to develop new forms of SAF reduction, it is instructive to briefly review the current techniques for SAF reduction.

Fine-grained Techniques The best example of fine-grain SAF reduction is clock gating within a processor core. Clock gating temporarily suspends the switching activity for transistors which are not involved in computation on a given cycle. Similar techniques can be applied to reduce leakage power as well (e.g., dynamic sleep transistors [110]). Such techniques are very effective for reducing SAF, but they have practical limits. In particular, the cost of additional hardware required to manage aggressive clock gating can often outweigh the benefits [58], encouraging designers to strive for additional SAF-reduction techniques, especially coarse-grain techniques that operate at higher levels of the design.

Coarse-grained Techniques Because of lower overheads, further SAF reduction is more effective when applied at a coarser granularity. Devoting a substantial fraction of the chip area to caches instead of cores is one method to achieve this objective since, by design, the second or third level (L2 or L3) caches have low activity (only a small fraction of cache lines are accessed in any given cycle). Additional SAF reduction can be achieved by putting large sections of the caches into a sleep state, reducing both leakage and dynamic power [94]. However, provisioning more resources

for caches yields diminishing returns in performance, even for workloads with large memory footprints, as demonstrated by recent studies [13, 59, 96].

2.4 Application of SAF: Speedup from Power Constrained Multicores

The previous sections have discussed the SAF and its trend based on the prevailing technology characteristics in details. The next two sections will demonstrates two concrete examples to demonstrate the application of SAF in reasoning about power limited multicore designs in their early design phase.

In their recent work on multicore chips, Hill and Marty [46] presented an analytic model to estimate the performance of various workloads on three different categories of multicore designs. Their cost model assumes that on-chip compute resources are limited (e.g., by area, power, or some combination thereof). However, with the increasing dominance of power considerations dictating the microprocessor design, whereas the available resources continue to grow exponentially, the area and power constraint must be decoupled. Decoupling these two constraints allows to model the active compute resources separately from the available resources, and offers a more flexible framework for modeling power and area constraint. Using SAF as a proxy for the power constraint, this section will discuss how to incorporate SAF in the analysis to achieve this decoupling, and its subsequent impact in the results and major conclusions.

2.4.1 Background

2.4.1.1 Cost Model

A brief discussion on the multicore cost model adopted by Hill and Marty is presented first. A multicore chip is divided into two components: *processor core components* that integrates n *base core equivalents* (n BCEs) and non-core components that include shared caches, interconnect, memory controllers and other essential microprocessor structures. Total area devoted to both these components are constant throughout the analysis. The processor core component can be configured in different ways, which will be determined by the specific multicore configuration this analysis inspects. In contrast, the non-core components remain identical across all the multicore configurations. While the configuration of the non-core components can clearly impact the performance, this section focuses on processor core area to understand the broader qualitative trend.

2.4.1.2 Performance Model

The performance and workload model for this analysis is driven by Amdahl's law [6]. Under Amdahl's law, the workload is modeled as two distinct phases: a sequential and an *infinitely* parallel phase. The performance obtained from the sequential phase is limited by the performance extracted from a single processor, while the parallel phase is only limited by the number of processing cores used to execute it. For a program that spends f fraction of its time in the parallel phase, on a multicore consisting of n processing cores, the speedup from this law is given by:

$$Multicore_{speedup} = \frac{1}{1 - f + \frac{f}{n}}$$

2.4.2 Hill-Marty Model Extension: Incorporating Power Constraint

The cost model described above does not separate the active compute resources from the available compute resources that are limited by the area constraint exclusively. Incorporating the power consumption within this framework, which is perhaps the more significant cost in modern multicore designs, is possible but requires it to closely tie into the area cost. In other words, a designer can limit the are devoted to the computation resources solely for a particular target power consumption. However, modern multicore chips already contain significantly more area than that can be powered up simultaneously, resulting in an $SAF < 1$. For example, 21% of the total area, implenting the processor cores in Intel Itanium 4, accounts for 59% of the overall power budget [101]. Therefore, for a more realistic modeling of power constraint in this analysis, one must *decouple* the area and power constraint.

As explained in Section 2.1.3, although the area and power constraints must be satisfied simultaneously for a viable implementation, particular mechanisms to satisfy each of them are fundamentally different. The area constraint is statically satisfied by the number of devices present on a chip, dictating the number of processor cores realized from them. In contrast, the power constraint of a given implementation can be dynamically satisfied, dictating which particular cores consume power. Decoupling these two constraints in such a manner allows to better model the current multicore systems that are primarily power limited with abundant area.

Thus, this analysis will include the SAF of this processor core area to represent the power constraint, in addition to the processor core area itself that satisfies the available area constraint. Following this basic modification, the analysis investigates the performance impact of different multicore chips with the variation of the SAF. For example, although the core area will contain n BCEs, depending upon the SAF, the number of BCEs that can be simultaneously employed for active computation will vary. An SAF of $1/2$ in a multicore will imply that only $n/2$ BCEs can be

employed to execute code at any given time. This section will use $\boldsymbol{\alpha}$ to denote the SAF of the core area, where $0 < \alpha \leq 1$.

2.4.3 Symmetric Multicore

A symmetric multicore consists of structurally identical processing cores. Thus, such a multicore with n BCE area, can be built with n cores (each with unit area), or n/r cores where each core contains r BCE area. The sequential performance of a core built with an area equivalent of r BCE is given by an arbitrary function $perf(r)$, although $perf(r)$ is strictly less than r [46]. Without considering the SAF, the performance of a symmetric multicore is driven by: the sequential performance of a single core (i.e., $perf(r)$), and the parallel performance of n/r cores (i.e., $perf(r).n/r$). This speedup is then given by the following equation:

$$Symmetric_{speedup} = \frac{1}{\frac{1-f}{perf(r)} + \frac{f \cdot r}{perf(r) \cdot n}}$$

Now the power constraint restricts the aggregate active area to $\alpha.n$ BCE area. Therefore, regardless of the amount of existing parallelism in the parallel phase of the code, one can employ $\alpha.n/r$ cores at most. The choice of r is now restricted in the sense that it cannot exceed $\alpha.n$. The speedup of the symmetric multicore considering its SAF is then given by:

$$Symmetric_{SAF speedup} = \frac{1}{\frac{1-f}{perf(r)} + \frac{f \cdot r}{perf(r) \cdot n \cdot \boldsymbol{\alpha}}}$$

2.4.4 Dynamic Multicore

A dynamic multicore is built with structurally similar processing core components, much in the same spirit of a symmetric multicore. However, unlike a true symmetric multicore, a dynamic

multicore can logically combine individual processing cores to boost the sequential performance. Thus, in the limit, during the sequential phase of an execution, it can fuse all the processing cores to boost the performance. Alternately during the parallel phase, it can employ all the cores in isolation to exploit maximum parallelism. Without considering the SAF, the speedup in this system will be:

$$Dynamic_{Speedup} = \frac{1}{\frac{1-f}{perf(n)} + \frac{f}{n}}$$

Similar to the case of symmetric multicore, incorporating SAF in this performance estimation boils down to realizing that only $\alpha.n$ BCE area equivalent can be in active state simultaneously. Therefore, both the sequential and parallel speedup boost in a dynamic multicore will be limited by the SAF factor α.

$$Dynamic_{SAFspeedup} = \frac{1}{\frac{1-f}{perf(\boldsymbol{\alpha}\cdot n)} + \frac{f}{\boldsymbol{\alpha}\cdot n}}$$

The basic assumption used here, which is derived from the original analysis by Hill and Marty [46], is that the performance of a dynamic multicore can continue to increase with additional processor cores, regardless of the implementation complexity for supporting such dynamic reconfiguration.

2.4.5 Asymmetric Multicore

The program analyzed here consists of two distinct phases: sequential and parallel. Therefore, although an asymmetric multicore allows many forms of structural asymmetry to co-exist, for the purpose of this analysis, the most optimal asymmetric configuration consists of one large core, comprising multiple BCE areas, for executing the sequential phase of the program, and the rest with unit BCE cores for executing the parallel phase. Thus, during the sequential phase, the large

core built with r BCE area is employed, while the parallel phase employs $n - r$ single BCE cores and the large core. Performance estimation without considering SAF will be:

$$Asymmetric_{speedup} = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}$$

Note that the above model presented by Hill and Marty [46] implicitly ignores any synchronization and wait time during execution of the parallel phase. Subsequent extension of this model in this analysis also conforms to this basic assumption.

While incorporating SAF in both symmetric and dynamic multicore performance estimation was simple, and qualitatively maintained the essence of the original model, asymmetric multicore is substantially different. The key contributing factor behind this fundamental difference is the ability to more effectively exploit the core area through the asymmetric design paradigm, and the ability to meet the power constraint by utilizing different fraction of the chip area. As the limitation of SAF restricts active computation to a fraction of the entire available area, in the limit, an asymmetric design can employ custom area component (e.g., a large core for sequential phase) to boost one phase without compromising the other phase. The computation then simply moves around on its designated best fit, without requiring the dynamic re-configuration as in dynamic multicore.

An expression is now developed for the general case of asymmetric multicore scenario. The sequential phase of the program is assigned to the large core with r BCE area. However, during the execution of the parallel phase, one may choose to exploit different sections of the chip, depending on the respective speedups offered. The collective area of those sections may exceed the power constraint when active simultaneously, but meeting the power constraint by selectively employing each fraction allows to exploit an over-provisioned area (see Section 2.1.3).

Different scenarios arise based on the sections of the area employed during the parallel phase

of the execution. First, if the area available after provisioning the large core, which is built with r single BCE cores, is as at least much as allowed by power constraint, then this section of the area can be used exclusively during the parallel phase. The performance in this case would be equivalent to the dynamic multicore, as both the execution phases are assigned to the best possible core area. Collectively, a total of $r + \alpha \cdot n$ area will be used, but the dynamic distribution of activity allows to satisfy the power constraint by restricting simultaneous activity to only $\alpha \cdot n$.

Otherwise, one can either use the large core during the parallel phase, or only use the single BCE cores. The speedup is estimated based on the maximum possible speedup attained among these two choices. Thus, the performance estimation of an asymmetric multicore will be [2]:

$$Asymmetric_{SAF\,speedup} = \begin{cases} \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{\boldsymbol{\alpha} \cdot n}} & : \quad r + \boldsymbol{\alpha} \cdot n \leq n \\ \frac{1}{\frac{1-f}{perf(r)} + \min(\frac{f}{perf(r) + n \cdot \boldsymbol{\alpha} - r}, \frac{f}{n-r})} & : \quad r + \boldsymbol{\alpha} \cdot n > n \end{cases}$$

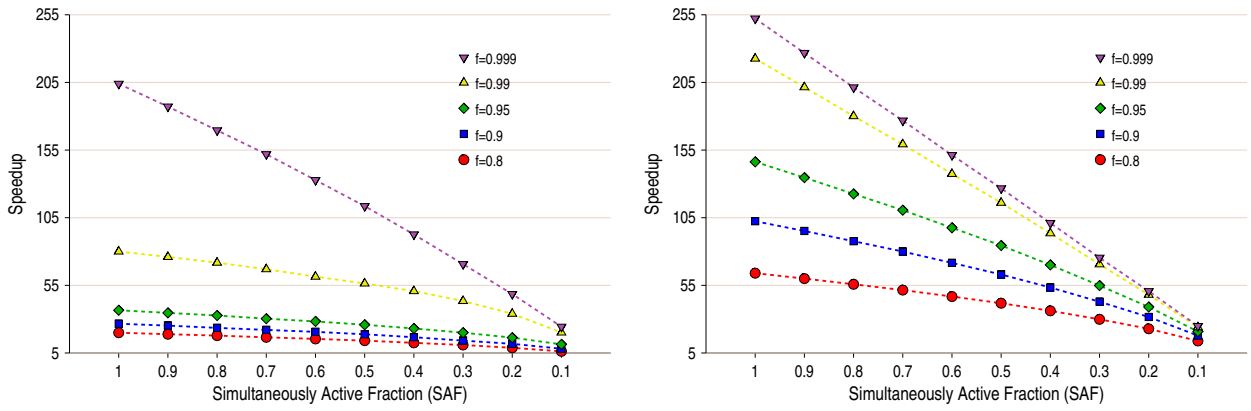
2.4.6 Results

This section now presents quantitative results based on the analytic models developed in the previous section. For each multicore configuration of a given size (i.e., n), there are three different parameters to vary:

- f: degree of parallelism,
- r: the number of single BCE area occupied by each core in symmetric multicore, and the size of the large core in asymmetric multicore,
- α: the Simultaneously Active Fraction or SAF.

For a given f and α, the performance of a particular type of multicore will vary depending on r, but will likely become optimal at a particular r (occasionally multiple values of r may also yield

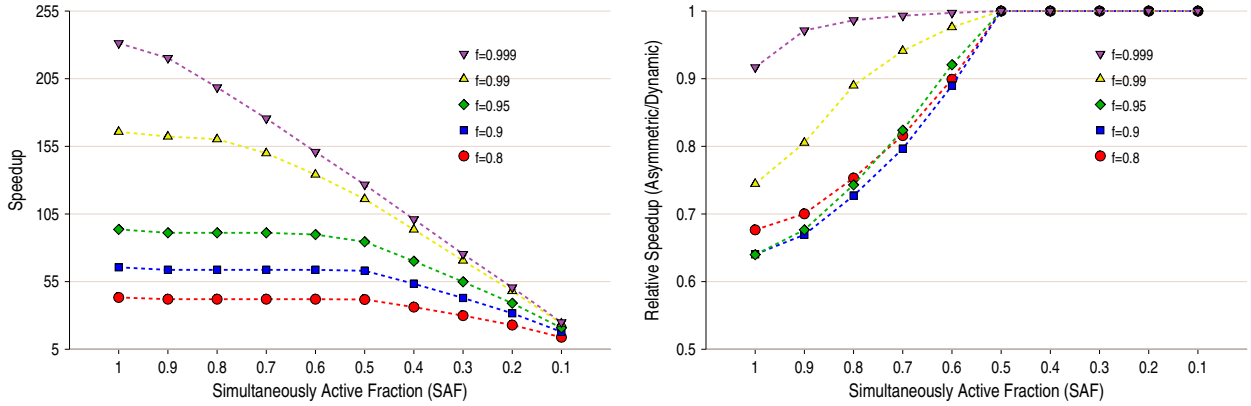
[2] Note that the minimum positive value in the denominator gives the maximum overall value.



(a) *Speedup from the Symmetric Multicore* ***(b)*** *Speedup from the Dynamic Multicore*

Figure 2.2: Speedup from the Symmetric and the Dynamic Multicore

the maximal performance). However, recall that the main objective of these analytic models is to estimate the maximum performance potential of a given multicore configuration. Consequently, for the benefit of readability, the figures will only demonstrate the maximum performance for each multicore as the experiment varies f and α, without specifically indicating the value of r that achieves this performance.



(a) *Speedup from the Asymmetric Multicore* ***(b)*** *Relative Speedup: Asymmetric vs. Dynamic Multicore*

Figure 2.3: Speedup from the Asymmetric Multicore and comparison with the Dynamic Multicore

Figures 2.2(a), 2.2(b), and 2.3(a) show the performance potentials of symmetric, dynamic, and asymmetric multicore configurations, respectively. For ease of comprehension, the relative speedup between asymmetric and dynamic multicore is shown in the Figure 2.3(b), where it plots the speedup of asymmetric multicore *normalized* to the speedup of dynamic multicore. Without any loss of generality, the figures show results for $n = 256$, and assume $perf(r) = \sqrt{r}$ for the purpose of comparison with the prior work. In all of these figures, results at $SAF = 1$ represent speedups obtained without considering SAF, and reflect the observation made in [46] (i.e., $Dynamic_{Speedup} > Asymmetric_{Speedup} > Symmetric_{Speedup}$).

However, with the reduction of SAF, the figures demonstrate several interesting trends in speedup. First, across all configurations, the reduction of SAF directly translates into the reduction in speedup obtained. This is an expected result since the fall of SAF logically implies a decrease in resource availability at any given time. Second, the analysis finds that among the three multicore configurations, the speedup reduction is lowest in the asymmetric configurations. Figure 2.3(b) shows this trend explicitly, where it presents the speedup of asymmetric multicore normalized to the dynamic multicore. At $SAF = 1$, the asymmetric multicore is substantially worse, yielding 8–36% lower speedup than the dynamic multicore (speedup loss being minimal where the sequential phase is comparatively smallest). However, with the reduction of SAF, the difference in speedup between these two configurations decreases, and at exactly $SAF = 1/2$, speedups from these multicores become identical. This result can be explained by the fact that when SAF is $1/2$, the asymmetric multicore can provision half of its core area for executing the sequential phase, and the rest for parallel phase. Since only half of the core area is active at any time, this provisioning does not compromise either of the phases, and therefore yield best possible performance with the allowable active area. Logically, this is exactly what happens in the dynamic multicore via re-configuration

With the reduction of SAF, another interesting trend observed within each of these configurations, is the decreasing gap in speedup between different degrees of available parallelism (or f). In fact, at $SAF = 0.1$, the speedups becomes significantly closer, regardless of the value of f—a significant change from the other extreme. For example, in asymmetric multicore, the maximum speedup (i.e., with $f = 0.999$) obtained at $SAF = 1$ is 5.34 times compared to that of the minimum (i.e., with $f = 0.8$). However, at $SAF = 0.1$, the same speedup ratio drops down to only 1.79. This result indicates that at lower values of SAF, the power constraint becomes more dominating than the degree of available parallelism: traditionally considered as the most important performance bottleneck.

Limitation Although an asymmetric multicore was able to cope with the increasing power constraint, much of its benefit came from exploiting an over-provisioned system (discussed next). Both symmetric and dynamic multicore had no such means to apply in this analytic model, and therefore with the reduction of SAF, employed a smaller core area over a larger time frame compared to the asymmetric multicore. One can argue that the additional area exploited in the asymmetric multicore could be added to the non-core area in dynamic and symmetric multicore. This additional non-core area may then lead to performance enhancement in those configurations, but such enhancement is impossible to estimate within this analytic framework–a fundamental limitation of this model. However, the addition of non-core area does not always lead to performance boost, and it is unlikely to alter the results in a way to impact the major conclusions.

For example, increasing the non-core resources to provision more chip area for caches yields diminishing returns in performance, even for workloads with large memory footprints. Despite the application trend in increasing memory footprints, many workloads do not show a proportionate increase in their working set behavior [45], leading to marginal improvements in miss rates with the increase of cache sizes after a certain point [13,59,96]. In addition, arbitrarily increasing cache

sizes leads to the degradation of hit latency, as well as the energy expended on every hit, causing a overall degradation in memory latency as the savings in the aggregate miss latency is offset by the increase in hit latency [41].

2.4.7 Over-provisioning

One of the major conclusions from the above study is the effectiveness of an asymmetric multicore in a power constrained design. Fundamentally, as the SAF is reduced, the asymmetric design starts to *over-provision* where it integrates more processing cores than that can be powered up simultaneously. Subsequently, this over-provisioning provides a choice in processing cores for assigning computation in different phases. With the reduction of SAF, the degree of over-provisioning increases, affording more choices that finally reach a point where both the computation phases can be assigned to a certain portion of the core area exclusively.

Figure 2.4 shows the direct impact of over-provisioning core resources in an asymmetric multicore. Unlike the results shown in the previous section, the allowable area for active cores is fixed to 256 BCEs in this graph. Instead, the figure shows the performance impact of increasing the total available core area. The degree of over-provisioning in the figure indicates this ratio of available core area to the active core area. The figure shows the speedups normalized to the speedup when the degree of over-provisioning equals one. Provisioning more core resources in an asymmetric multicore results in a dramatic improvement in performance, especially when the workload consists of a substantial sequential phase. For $f = 0.9$, the asymmetric multicore shows a 56% increase in performance when the available area is twice that of the active area, which allows the best assignment of the two computation phases.

Evidently, an over-provisioned multicore provides a straight-forward way to boost performance in an asymmetric design paradigm. However, whether it can provide any advantage for multi-

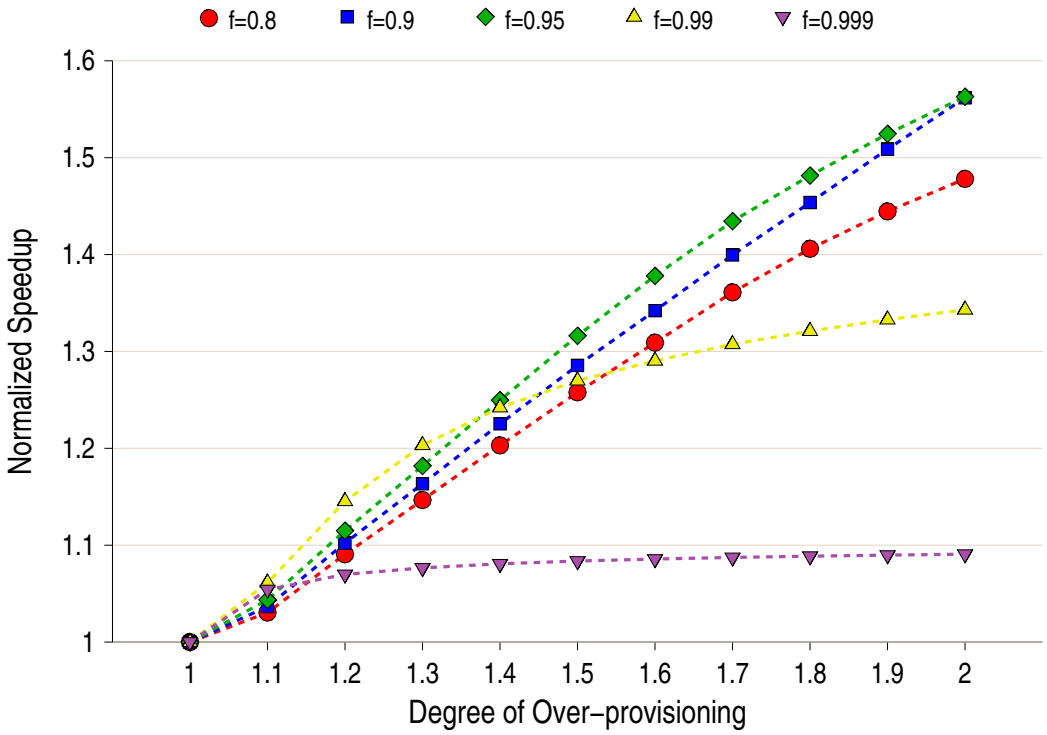


Figure 2.4: Impact of over-provisioning in asymmetric multicore speedup. Degree of over-provisioning indicates the ratio of available compute resources to the allowable active resources. The speedups are normalized to the speedup obtained when the degree of over-provisioning equals one.

threaded workloads in a power constrained symmetric or homogeneous design is an open research problem, and the main topic of this dissertation. The next section presents the conceptual idea of how over-provisioning can help in multithreaded workload via using an analytic model, while a more detailed analysis using execution driven simulation is presented throughout the rest of this dissertation.

2.5 Application of SAF: Designing Multicores for Multithreaded Applications

2.5.1 Background

While a large fraction of the computer industry is struggling to find ways to scale their applications to a multicore platform, one class of applications are already able to exploit current multicore systems: multithreaded programs. Multithreaded programs use a threading abstraction to express parallelism, which is then implemented using standard thread models such as POSIX threads or *pthreads* [21]. This section demonstrates another application of SAF and over-provisioning, but in the context of designing multicores for multithreaded applications.

This dissertation focuses on one important class of multithreaded applications: server class applications. Server class applications are widely deployed, and present a variety of system design challenges ranging from improving performance, energy efficiency and so forth. This section will illustrate how an SAF perspective can aid in early phase of designing such a system, particularly tailored to handle these applications.

2.5.2 Speedups from Multithreaded Server Applications: Analytic Model

Server class applications spend a significant portion of their execution time in the OS mode, essentially performing tasks in the kernel on behalf of the user [8, 36, 67, 71]. Mixing these two computation modes involving user and OS code often leads to conflicting demands on various micro-architecture components that exploit properties of instruction stream (e.g., branch predictors and caches), degrading the performance of these applications [36, 67].

For the purpose of this analysis, the multicore system is assumed to consist of two broad class

of processing cores that offers different speedups for the two computation modes. Although they offer different performance characteristics, these two class or processing cores can be structurally identical. The asymmetric speedup is achieved via the dynamic specialization of the on-chip cores, much in the spirit of the dynamic multicore in the previous section. This dissertation will illustrate how such specialization can be achieved later in Chapters 3 and 4. Effectively, this multicore design is logically equivalent to a system with two different types of hardware accelerators, each excelling in one of the two computation phases.

The workload is characterized by two distinct phases as mentioned above, but the portion of total execution time spent in a given phase can vary. In this section, the fraction of execution time in the user mode is denoted by β. To simplify this analysis, a given thread is assumed to be in one phase, independent of the phases of the other threads in the system. Thus, at any given time, the probability of a thread to be in the user mode is β, while that of in the OS mode is $1 - \beta$.

The effective speedup achieved during any given period of time depends on the actual configuration of the multicore that specifies the distribution of user cores and OS cores, and the phase distribution of the threads in that time period (i.e., the number of threads in OS and user computations, respectively). Assuming a relative speedup of η when a given phase is assigned to an appropriate core (e.g., user phase on user core), the expected overall speedup from a particular multicore configuration can be calculated by a weighted sum of the speedup from every possible distribution of phases, where the weight indicates the probability of the corresponding phase distribution.

Consider a N core system, where k cores are configured to accelerate user phases and the rest $N - k$ cores to accelerate the OS phases. The number of cores engaged in active computation is dictated by the chip TDP. In a conventional multicore, different threads from a multithreaded application are scheduled on all of these cores. As explained above, every possible phase distribution

has an associated probability, and an expected speedup from this multicore configuration. For example, a distribution with 2 user phases and $N-2$ OS phases has the probability of $\beta^2 \cdot (1-\beta)^{N-2}$, with $N!/(2! \cdot N-2!)$ ways such a distribution can be realized among N threads. The corresponding expected speedup will be $\eta \cdot 2 + (k-2) + \eta \cdot (N-k)$ (2 user cores speeds up the 2 user phases, the $N-k$ cores can speedup only that many OS phases, and the rest of the $k-2$ phases achieves no relative speedup). The overall speedup in this model is given by the following equations:

$$Speedup = \frac{\sum_{i=0}^{N} p_i \cdot s_i}{N}$$

$$p_i = \beta^i \cdot (1-\beta)^{N-i} \cdot (N!/i! \cdot N - i!)$$

$$s_i = \eta \cdot min(i,k) + \eta \cdot min(N-i, N-k) + max(i-k, k-i)$$

where p_i denotes the probability of a phase where i threads are in user mode, and s_i denotes the expected speedup for that phase. The additional speedup in the two phases is limited by either the number of corresponding cores or phases. For example, when the number of user phases i exceeds the number of user cores k, only k such user phases can achieve speedup from the user cores. Likewise, the minimum of the OS phase and cores determine the speedup from the OS phases. Rest of the phases, given by $max(i-k, k-i)$, must then be assigned to cores that are not the best possible fit, achieving no relative speedup.

2.5.3 Impact of Over-provisioning

Now consider the impact of over-provisioning in this analytic model for estimating the speedup in multithreaded server applications. The application still has N threads to schedule. Under over-provisioning, the chip can contain more than N cores, but have at most N active cores at any

given time as dictated by the chip TDP. This configuration still satisfies the SAF constraint of the conventional multicore, but the over-provisioning is done at a core granularity instead of devices, as explained previously in Section 2.1.3.

When there are more than N cores, more phases can be assigned to their appropriate cores that achieve the best speedup. In the limit, one may have as many as $2N$ cores in an over-provisioned multicore, with N user cores and N OS cores. This multicore always achieve the best possible speedup as all the threads get their best core assignments, regardless of the phase distribution among user and OS computation. The degree of over-provisioning in this case, given by $Cores_{available} \div Cores_{active}$, is 2. The degree of over-provisioning can vary with an impact on total speedup achieved, and the next section demonstrates this impact.

2.5.4 Results

There are four key parameters to consider for estimating the speedup in multithreaded server applications using the analytic model described above.

- k: the number of user cores in the system
- β: fraction of the execution time spent in user mode
- η: Speedups achieved when appropriate core assignment is made (e.g., user computation on user core)
- λ: degree of over-provisioning

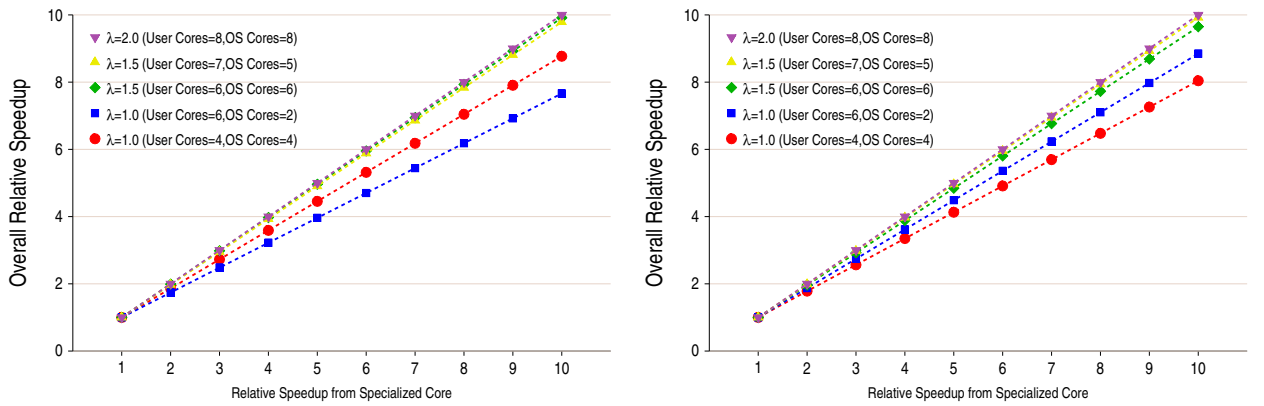
Without any loss of generality, Figures 2.5(a) and 2.5(b) show the results where the TDP of the target multicores allows a maximum of eight simultaneously active processing cores. Based on the degree of over-provisioning, the number of available cores in a multicore is $\lambda \cdot 8$. The system

is configured to schedule eight application threads to execute on the on-chip cores. Figure 2.5(a) shows the result for an application that spends equal amounts of time in both the user and OS, while Figure 2.5(b) shows a *bias* in the execution time spent between user and OS. In particular, Figure 2.5(b) shows an application which spends 70% of its time in the user computation. The result for an opposite bias (i.e., 70% time in OS) will be identical to Figure 2.5(b) due to the equivalent speedups in OS and user computation assumed in this analysis.

Both these figures show 5 different multicore configurations. Two configurations use no over-provisioning ($\lambda = 1$). Each of these two configurations represents the best performing multicore among all possible configurations in that class (i.e., $\lambda = 1$), for one of the two different applications shown in the two figures. For example, the multicore with 4 user cores, and 4 OS cores, performs best in the application that spends equal time in user and OS, while the multicore with 6 user cores and 2 OS cores performs best in the application with 70% time spent in the user computation. Similarly, two over-provisioned multicores are shown with $\lambda = 1.5$. Among these over-provisioned multicores, the one with 6 user cores and 6 OS cores performs best in the application with equal OS and user computation, while the multicore with 7 user cores and 5 OS cores performs best in the other. Finally, to show the speedup limit achievable in this model, a multicore is shown with $\lambda = 2$, with 8 user and OS cores, respectively.

Both these figures demonstrate that using appropriate core resources in the appropriate phases of computation increases the relative speedup seen in these workloads. Even without over-provisioning, the best possible speedup achieved in these two applications, albeit with different multicores, are significant. For example, Figure 2.5(a) shows that multicore with 4 user cores achieve a maximum speedup of 8.77, when $\eta = 10$.

Over-provisioning 50% more cores (i.e., $\lambda = 1.5$) further increases the speedup achieved in both these applications, approaching the best possible speedup in all cases. Figure 2.5(b) shows



(a) *Server application with uniform phase distribution (50% user, 50% OS)*

(b) *Server application with biased phase distribution (70% user, 30% OS)*

Figure 2.5: Impact of Over-provisioning on Speedups from Multithreaded Applications

that the multicore with 7 user cores achieve a speedup of 9.92, while the other one with 6 cores also achieving comparable speedup of 9.64.

In fact, these comparable speedups achieved by both the over-provisioned multicores, even when one of them is not the best for that particular application, is the most interesting aspect of this analysis. Over-provisioning allows these multicore configurations to *dynamically adapt* to the difference in application characteristics, yielding a performance comparable to the best possible configuration. For example, without over-provisioning, the multicore that showed best performance for the biased application only obtains a speedup of 7.66 in the other application. In comparison, the corresponding over-provisioned multicore with $\lambda = 1.5$ achieves an overall speedup of 9.79–an improvement of 27.8%

Figure 2.6 demonstrates this aspect of over-provisioned systems explicitly by showing the performance for five multicore configurations where the fraction of user computation (β) varies from 0 to 1. The results are shown assuming $\eta = 5$ only. Among the multicores without over-provisioning, the worst performance is seen when the application spends its entire time in the OS. The multicore with 6 user cores and 2 OS cores achieves a relative speedup of 2. The corresponding worst

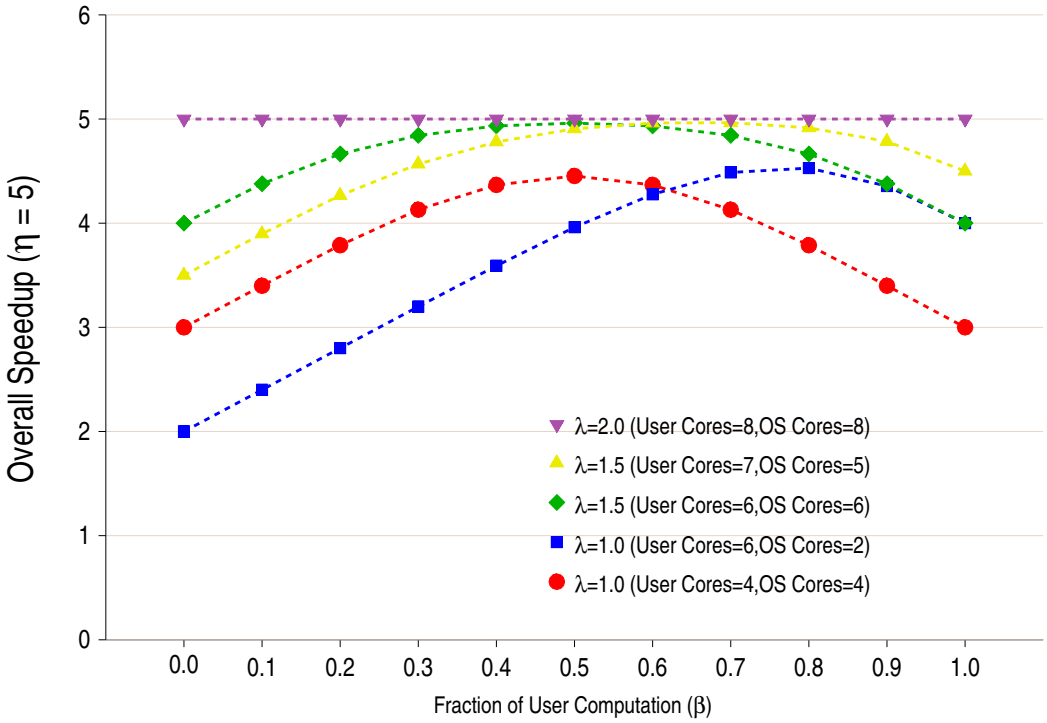


Figure 2.6: Speedups in server applications with varying user computations. The fraction of time spent in the user computation (β) is varied from 0 to 1.0. The speedups are shown for $\eta = 5$.

case for the over-provisioned multicores also happens in the same application, but relatively the over-provisioned multicore performs 75% better than the multicore without over-provisioning.

In summary, two different aspects of multicore designs led to the speedups seen for these server applications. First, the presence of core resources that can excel in one of the two major computation modes yields substantial speedup. Second, over-provisioning core resources yields even better speedup, compared to a multicore without any over-provisioned core resources–the typical case in current generation multicores. However, the most important insight gained through this analysis is an important feature of an over-provisioned system, where it could dynamically adapt to the change of application characteristics. Next several chapters in this dissertation will present a more in-depth characterization of the server workloads, demonstrate techniques to achieve dy-

namic specialization from structurally identical processing cores, and present a rigorous analysis of the impact of over-provisioning.

2.6 Summary

Simultaneously Active Fraction presents an abstract model of the power constraint in a microprocessor chip, and is defined by the *fraction* of the chip area that will account for the entire power budget when simultaneously active. Because this metric is expressed as a fractional area or chip resource, it can be very useful in the early phase of a power constrained microprocessor design.

This chapter developed a model for calculating the SAF in multicore chips by combining transistor level characteristics with the trends in system level device integration and power budget. Using published literature from the technology roadmap (ITRS documents) and tool set, the chapter demonstrated that the SAF of microprocessor chips is steadily decreasing over the next several technology generations.

The chapter then discussed the importance of SAF reduction as a means for controlling power consumption in future chip designs. Fundamentally, with the diminishing returns from transistor level power savings, the importance of SAF reduction is growing, and fast becoming the most dominant technique for reducing power consumption. For the ease of comprehension and developing new techniques, the chapter broadly classified the existing techniques for SAF reduction, and discussed their advantages and shortcomings.

Finally, the chapter illustrated two concrete examples of reasoning based on SAF to demonstrate its effectiveness in modeling power constraint at an early phase of multicore design. Based on a prior work on an analytic model for multicore speedup from Amdahl's law, it showed a technique to incorporate SAF in the model and extend it for current multicore systems that are primarily power limited. Similar to the original model, the chapter evaluated three different multicore

configurations: symmetric, asymmetric and dynamic. Interestingly, it found that different multicore configurations react differently when power becomes a constrained resource, even altering the qualitative difference between various configurations reported earlier. Comparatively, asymmetric multicore copes much better with the power constraint, and after a certain point, it performs as good as the dynamic multicore. The chapter then argued that *over-provisioning* is the fundamental underpinning of its relative performance enhancement.

Subsequently, another example is presented that investigate the combined effect of over-provisioning and using specialized cores in multithreaded server applications that show two distinct computation phases: OS and user. In this context, over-provisioning shows much better potential in adapting to the changing application characteristics, compared to a multicore without any over-provisioning.

Chapter 3

An Over-provisioned Multicore System

The pursuit of energy efficiency and higher performance in successive microprocessor designs have compelled the designers to effectively provision more functional components than those that are strictly necessary for correct execution. By definition, this strategy leads to a reduced resource utilization, with the extent of over-provisioning being inversely proportional to the utilization. Nonetheless, there are several examples of over-provisioning in the micro-architecture components of current implementations (e.g., the physical register file and instruction window entries).

A class of circuit components that epitomize such a design style, and is growing in prominence in recent times, is *hardware accelerators*. These special purpose circuit components are typically designed for a specific computation [5, 28, 39]. Such custom design components often deliver orders of magnitude improvement in power-performance for the specific computation they were intended [56]. The ITRS roadmap predicts a rapid proliferation of such accelerators [55], increasing the extent of over-provisioning in the multicore designs, since a specific computation utilizes only one of them.

But, unlike these examples of individual circuit components, this dissertation considers over-provisioning at an entirely different level of abstraction: a *processor core*. Fundamentally, over-

provisioning at the core granularity (and hence the term Over-provisioned Multicore System) makes this a unique system, and simultaneously leads to several challenges and opportunities.

An Over-provisioned Multicore System (OPMS) is a class of multicores that *by design* provisions more processing core resources than that can be kept active for its target Thermal Design Power (TDP). The key distinction between a conventional multicore and an OPMS stems from the intrinsic goals in their system designs, respectively. Canonical multicore considers a processing core as one of the most critical components of the system, and hence targets full utilization of its core resources. In accordance with the current technology trends, where the ability to integrate processing cores on a single die is outpacing the capability to power them up simultaneously, an OPMS considers power and thermal characteristics more critical than the processing cores. Consequently, it provisions more core resources to offer a *choice* in assigning computation on them, and improve system characteristics (e.g., energy efficiency, thermal profile) via judicious computation assignment policies.

Due to its unique system design perspective, an OPMS also offers a novel way for SAF reduction: a requirement that *all* future multicores must strive to meet. Inactive cores—processor cores that do not partake in active computation at a given time—do not constitute the SAF of the chip, thereby reducing its SAF significantly. So the obvious question then is: ***what is the purpose of an inactive core***? This dissertation will demonstrate however, an inactive core, while not performing any computation, can facilitate several desirable functions: retain predictive state (in a low-leakage mode) to function as a generic accelerator for a particular computation in the near future, allow thermal hotspots to cool off, or simply exist as a spare. A major part of this dissertation revolves around developing a broad framework to facilitate novel uses of inactive cores in future multicore systems. This chapter will discuss this broad framework in detail.

As an alternative to the OPMS designs, one may argue to simply reduce the die space with

technology scaling, effectively allowing lower cost micro-processors in each successive technology generations. However, as the past history suggests, architects have always sought similar, and often larger, die area to accommodate more resources, and improve performance by judicious use of such resources. While reducing the die size can save production cost, but even modest improvements in performance and energy efficiency can dramatically increase the market value of a processor, easily compensating the additional cost of larger die. Therefore, in accordance with the ITRS roadmap [55], this dissertation assumes that future designs will continue to adopt fairly similar die size, and delves into techniques that can use these additional resources for improving performance and energy efficiency, without violating the power constraints in the design.

This chapter is organized as follows. Section 3.1 describes the OPMS framework, illustrating the key concepts with an example. Section 3.2 presents the broad opportunities and challenges in this framework with a qualitative discussion of the cost and benefit of its major functional aspects. Implementation details of the OPMS framework is described in Section 3.3. Section 3.4 gives an outline of an application of the OPMS, which is rigorously evaluated in this dissertation. Finally, the chapter concludes with a summary in Section 3.5.

3.1 OPMS Framework

Figure 3.1 illustrates the fundamental working principles of an OPMS . A computation composed of three fragments A, B, and C is assigned to an OPMS with three cores $P1$, $P2$, and $P3$, respectively. In this example, and throughout this dissertation, a computation fragment will denote an arbitrary portion of the dynamic instruction stream. After the first fragment A completes execution on $P1$, a *Computation Transfer (CT)* takes places, which enables the next computation fragment to resume execution on a different on-chip core, $P2$. Since $P2$ was in an inactive state prior to the CT, a wakeup signal W is enabled to transition it to the active state. Computation B resumes on

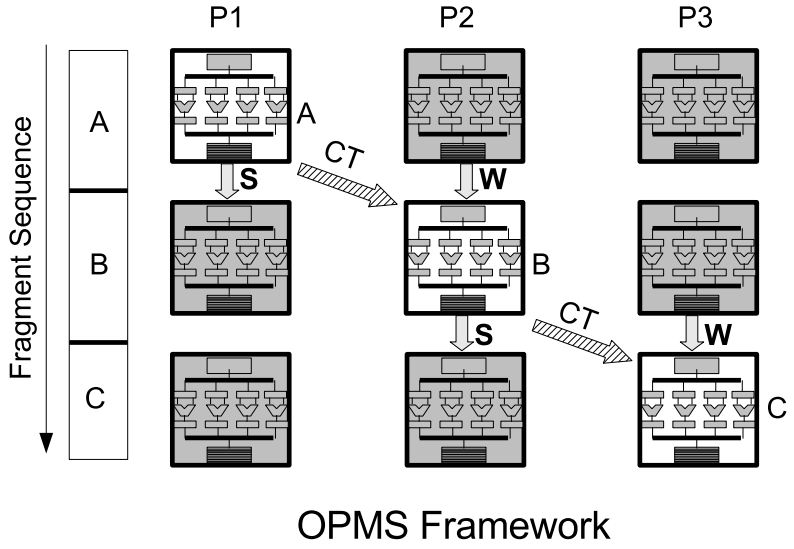


Figure 3.1: An example of OPMS execution. Inactive cores are marked by a shading.

$P2$ after the core transitions to the active state. Meanwhile, $P1$ transitions to the inactive state, via enabling a sleep signal S on it. Similarly, $P3$ executes the last fragment C. Collectively, three different processing cores compute the entire fragment, each of them transitioning between active and inactive state, whenever necessary.

3.1.1 Fundamental Technology Invariants

Before presenting an in-depth discussion on OPMS design considerations and its implementation, it is instructive to bring forth the fundamental technology invariants assumed in this design paradigm.

First, the OPMS design paradigm assumes that the relative area cost of integrating processing cores on a single die, is *marginal* compared to the cost of powering them up simultaneously. Con-

sequently, it considers the power budget as more critical than core utilization, and therefore aims to dynamically distribute the chip power budget by assigning computation on specific cores that are most likely to improve the performance and energy efficiency of the system.

Second, both the OPMS and CSP proposals rely on superior bandwidth of communication between on-chip cores, allowing to build a co-operative execution framework. The collective computation is multiplexed among the on-chip cores, enhancing the locality in their memory references and improve performance. Conventional multicores, which assign large portions of computation on individual cores, are not designed to exploit co-operation among on-chip cores, leading to a poor utilization of the aggregate resources.

Third, the work in this dissertation assumes a constant die size in future technology generations, also reflected in the ITRS roadmap [55]. Severe power constraint may cause a reduction in the die area, which can also bring down the production cost of microprocessor chips. However, judicious use of the transistor bonanza can improve the energy efficiency without requiring a larger power envelope, dramatically increasing the market value of such microprocessor designs. Thus from a research perspective, it is intriguing to investigate architectural means to deliver energy and power efficiency from the continuous growth of resources, rather than only realize benefits in production cost from technology scaling.

3.2 Challenges and Opportunities

An OPMS creates several broad implementation challenges in order to manage processing components in two disjoint pools: active and inactive cores. Furthermore, exploiting opportunities within this OPMS framework may require different processing cores to turn off and on frequently, creating a constantly-varying pool of processing cores. Given such a multicore framework, this section will first address its two major implementation challenges: (a) supporting a commodity general-

purpose software stack on top of an OPMS; and (b) understanding the cost-benefit trade-offs for supporting inactive cores. Subsequently, the next section will discuss the key opportunities offered in this radically different computation environment to improve thermal, energy and performance characteristics of the chip.

3.2.1 Interfacing with System Software

The system software, which can be a regular operating system or a hypervisor like VMware's ESX Server [113], manages three broad categories of system resources: processing cores, memory, and I/O devices. For each of these resources, this section discusses the special requirements for interfacing with an OPMS in turn. Because the applications are run by the system software itself, proper interfacing with the latter will guarantee correct execution of the former.

3.2.1.1 Processing Cores

Commodity system software uses a straight forward abstraction of the underlying complex micro-architecture: it assumes an entire processing core as the unit for assigning computation. Alteration in the core resources, and their management policies, as well as the variation in individual capabilities of cores, can directly impact the system software and its various performance metrics [10, 32]. Although assigning computation in a multicore adds complexity as compared to a uniprocessor system, the design paradigm of an OPMS, where cores frequently transition between active and inactive state, raises this level of complexity. In addition to managing multiple cores, the system software must now adapt to this intrinsic variability and availability of computation resources. However, this dissertation will demonstrate how to manage such a system with only modest increase in complexity, while delivering the benefits of an OPMS design paradigm.

There is a subtle distinction between the actual processing cores of the chip (the physical CPU

or PCPU), and the abstraction of processing core exposed to the system software (the virtual CPU or VCPU). Typically, there is a one-to-one correspondence between VCPUs and PCPUs, although a hypervisor hosting multiple virtual machines may break this unique mapping [99]. Broadly, computation resources in an OPMS can be exposed to the system software in two ways, essentially differing in the VCPU-PCPU correspondence, and the division of complexity between system software and the micro-architecture.

First, all the available PCPUs can be exposed to the system software, thereby maintaining the unique mapping of VCPU and PCPU. The system software will then be responsible for managing the computation resources entirely, and cannot simply assign computation on all VCPUs: the usual norm in a conventional multicore. Moreover, it must also support frequently transitioning on-chip cores from active and inactive, at time periods much lower than the typical scheduling quantum. Clearly, both these functional requirements will add significant complexity to the system software, but arguably will reduce hardware complexity. Second, only a subset of the available PCPUs are exposed to the system software (i.e., $VCPU < PCPU$). The system software will then schedule application threads to run on these VCPUs, while the micro-architecture will distribute the collective work among the on-chip cores. While avoiding any additional complexity at the system software, this solution is likely to introduce complexity in the hardware implementation.

These two broad solution strategies represent the two end points of a spectrum of possible solutions, which can balance the aggregate complexity between the hardware and software. The specific solution this dissertation evaluates employs virtualization techniques without requiring intrusive changes to system software, and it is discussed later in Section 3.3.

3.2.1.2 Memory

The OPMS design does not interact with the general memory management policies of system software, and therefore requires no change in that regard. However, the virtual-to-physical address translation mechanisms, which employ specialized structures like Translation Lookaside Buffer (TLB), can have correctness implications in an OPMS. A TLB is a small cache for the recently used physical address translations, and it is closely tied to the pipeline logic of a processing core. Transferring a computation from one core to another, requires appropriate handling of the TLB state, so as to preserve the correctness of the execution.

Two distinct courses of action may cause a correctness violation in a running thread. First, when a running thread (or VCPU) vacates a PCPU, the system must ensure that the translations present in the vacating PCPU's TLB are not incorrectly accessed by another thread. Although an implementation issue in an OPMS, the system software already deals with this particular problem during a regular context switch operation. Second, after transferring its computation onto a new physical core (or PCPU), the running thread must continue to execute correctly with the TLB associated with the new core. To support frequent transfer of computation, the performance penalty from a cold TLB must be avoided.

Much of the hardware support required to address these issues are already provided by modern microprocessors. Several current generation microprocessors support *address-space identifiers* or *ASIDs*, which allow a single TLB structure to cache entries from multiple processes running on their own virtual address space. Consequently, there is no need to flush the TLB entries when a computation moves between on-chip cores, saving much of the TLB miss penalty associated with starting execution on a cold TLB. Other design issues like TLB shootdown are similar in spirit to handling I/O interrupts (discussed next).

3.2.1.3 I/O

In a conventional system, I/O is interfaced with the system software through the interrupt handling mechanism. When a running thread, assigned to a VCPU, makes an I/O request, an interrupt is sent to the I/O subsystem. On completion of this request, the I/O subsystem informs the initiating PCPU—where the requesting VCPU was mapped—via another interrupt. The system software handles this second interrupt, and then resumes the computation of the original thread. A different flavor of this basic mechanism takes place during a software interrupt, typically used for inter-processor communication (e.g., TLB shootdown in Solaris 9). However, for the purpose of this discussion, the distinctions between them are insignificant.

A key aspect of an OPMS design, where the mapping between VCPU and PCPU alters dynamically, can have implications to this basic interrupt handling mechanism. For example, the mapping may change between the initiation and completion of an I/O request, creating a subtle challenge in delivering the interrupt to the appropriate PCPU. Furthermore, occasionally a VCPU may not even be assigned to a particular PCPU, exacerbating the problem of interrupt delivery.

For the purpose of this dissertation, the OPMS framework proposes a simple modification to the interrupt control logic so as to dynamically track the mapping of VCPUs to PCPUs. Occasionally, an interrupt may be held (e.g., in the absence of a VCPU mapping), and delivered after a delay.

3.2.2 Inactive Cores: Cost-Benefit Analysis

By design, an OPMS maintains a pool of processing cores in an inactive state, while the actual computation is assigned to the rest of the cores. While the inactive cores come with marginal area cost, especially in multicore chips with several billion devices, other costs to support them (such as power, interconnect) can be substantial. To understand the engineering trade-offs of inactive cores in an OPMS, this section presents a qualitative cost-benefit analysis, while more elaborate

discussions with the quantitative aspects will be presented in Chapter 6.

Benefit Components Inactive cores offer several broad utilities in a multicore system that can be harnessed through appropriate computation assignment policies. Three different mechanisms are outlined by which an inactive core can be effective in an OPMS framework: by allowing heterogeneous system design, retaining useful predictive state, and offering thermal slack.

First, provisioning a set of inactive cores allows the designer to build effective multicore systems that logically comprise of hardware accelerators, each of them excelling in a specific computation. As this dissertation demonstrated in the previous chapter, these heterogeneous or asymmetric systems offer tremendous potential for improving performance in a power constrained environment (see Section 2.4.6). In that example, either the large core provisioned for the sequential phase, or the group of small cores to facilitate the parallel phase, remained *inactive* during the execution of their complementary phase, respectively. But all these cores were useful by being efficient while active, thereby improving the performance of the system.

Second, inactive cores can retain *useful* predictive state in their private structures like branch predictors and caches. In a processor core, predictive structures play a key role in speeding up a computation, especially when the cached predictive state is effective for the assigned computation. By maintaining the relevant predictive state of a computation, inactive cores can logically function as a *generic accelerator* for a particular code sequence, and substantially speed up their executions as compared to other cores without the appropriate cached state. The key advantage of such execution is that the benefits are derived without any structural modification to the micro-architecture of the cores, unlike the hardware accelerators that require custom design to realize a power-performance advantage.

Finally, inactive cores offer a thermal slack—*at a core granularity*—for current multicore designs where the temperature management in critical structural components is a major design objec-

tive [97]. Periodically distributing computation between multiple cores reduces the thermal load on each core, allowing it to cool off during the periods of inactivity. Depending on the periods of activity, this work distribution can decrease both the peak and average temperature of major circuit components in the core.

Cost Components The major cost components of inactive cores come from their power consumption, and the interconnect support necessary for their seamless operation in a multicore.

Even without executing any instruction, inactive cores can still consume power, thereby reducing the available power budget for their active counterparts. The static power component in the transistor devices contribute bulk of this power consumption (see Section 2.2). Therefore, one major implementation goal for an OPMS is controlling the power consumption from inactive cores, so as to allow maximum power budget to the active cores. Other system goals, like the ability to quickly turn on and off cores, and maintaining predictive state in the core structures, add complexity in designing a solution to achieve this goal.

Fortunately, several techniques have been developed in the recent past that specifically target reducing leakage, while allowing fast startup for various micro-architecture components [30, 85, 93, 110]. This dissertation extensively adopts two of these techniques, and the major trade offs with implementation details are discussed in Section 3.3.2.

Another major cost component in supporting an OPMS is the additional cost for building an interconnect for OPMS cores. Compared to a conventional multicore, the incremental interconnect cost will depend on the degree of over-provisioning. For example, if a current multicore consists of 8 on-chip cores, supporting 12 cores in a similar interconnect fabric may not radically alter the interconnect cost, but expanding the same interconnect to support 32 cores may become prohibitive. Moreover, the primary runtime cost of an interconnect manifests as messaging delays and energy overheads. If an OPMS can maintain or even *improve* the bandwidth usage from the underlying

interconnect, these additional costs for the fabric may be worth the extra cost. Chapter 6 will present quantitative results to demonstrate how such improvement can be achieved via judicious assignment of computation on an OPMS.

3.2.3 Flexible Computation Assignment

In a uniprocessor environment, there is only one processor core to assign computation. Multicore systems introduce a choice in assigning computation by virtue of multiple available cores. However, current multicores deem these processing cores as critical resources, and thereby target full utilization of them. Fully utilizing processor cores limits the flexibility of scheduling computation on them, as there is a potential performance penalty for choosing one processing core at the cost of keeping some of the cores inactive.

By provisioning more computation resources, an OPMS relaxes the necessity of keeping all the processing cores busy at all time. As a result, for every computation, one can choose among several processing cores to assign it, without any negative impact on system performance due to the under utilization of the rest of the cores. Moreover, by being careful to choose processing cores (e.g., one with a cooler temperature or one that retained useful predictive state), various performance metrics can be improved substantially. OPMS designs enable this flexibility in assigning computation, and is the fundamental principle behind various novel application of OPMS designs. This dissertation will focus mainly on *Computation Spreading*, a locality enhancement technique for multithreaded server class applications.

3.3 Implementation

This section will now discuss the specific implementation details of an OPMS, including how this implementation handles the opportunities and challenges of the constantly varying pool of computation resources in an OPMS through virtualization support, and the techniques adopted to alleviate leakage in inactive cores.

3.3.1 Virtualization Support

While a software intrusive implementation is indeed possible as outlined in the section 3.2.1, without any loss of generality for OPMS designs, this dissertation implements a specific solution based on virtualization techniques. The main objective of this solution is to support an OPMS *transparent* to the system software, but limit the complexity for the hardware implementation.

At a first glance, modern multiprocessor operating systems already seem able to handle processors being turned on and off for reasons such as faults and power management: it is as simple as providing the OS scheduler with the new resource configuration. In an OPMS, however, as one attempts to make effective use of inactive cores (e.g., their ability to retain predictive state in a low power mode), the system will encounter frequent core transitions at time scales that are orders of magnitude smaller than a typical OS scheduling quantum. Effectively handling such frequent transitions would likely require intrusive modifications to existing operating systems, as current implementation are considerably slow in adapting to system re-configurations [115]. As an alternative, an OPMS can be supported using virtualization techniques, and thereby require no modification to the OS or the various application software layers.

Due to the increasing prevalence of a full fledged virtualization layer in server platforms, termed as System VM [99], the proposed OPMS implementation is designed to inter-operate transparently in virtualized platforms as well. System VMs are responsible for managing hardware re-

sources among its guest OS. Three major components of such hardware resources are: processors, memory and I/O [99]. Unlike such VMMs, designed to support multiple guest OSs, the functional requirements in an OPMS framework are limited, and only involve the virtualization of the processing cores. Other design components of the system VMs like memory, I/O, and security measures are not necessary for the OPMS framework, substantially reducing the complexity in the VMM implementation for an OPMS framework.

Due to this simplification, this dissertation implements this component of virtualization—managing cores resources of an OPMS—as a programmable firmware component. This component is specifically responsible for moving a computation from one core (e.g., a core that the system might want to transition to an inactive state) to another core (e.g., a core where the computation might be carried out more efficiently), and assigning computation based on some pre-determined policy. To accomplish this goal, an OPMS must provide specific mechanisms to communicate necessary processor state between on-chip cores, and implement a framework to support a variety of assignment policies, which assign a particular computation fragment to appropriate cores. Implementation details for these components are discussed in the next three sections.

3.3.1.1 Processor State

To manage the processor core resources via virtualization, the OPMS framework must also provide support for virtualizing the processor state, which is necessary for the correct execution on a processor core. In the evaluation, this dissertation uses functional models of the UltraSPARC IIICu architecture, based on SPARC V9 ISA (described in more detail in Chapter 5). The processor state for this CPU is composed of two major components: register state and the TLB state.

The UltraSPARC IIICu CPU has a large number of architected registers, including windowed general purpose registers (GPRs), four sets of global registers (normal, Alternate global or AG,

Interrupt global or IG, and MMU global or MG), floating-point, privileged, ASI-mapped, and TLB control registers [103]. The aggregate register volume comprises of 277 64-bit registers, or 2.2kB (nearly half of which is SPARC register windows). In other architectures without register windows (e.g., $x86$), register values are saved in the memory across function calls, thereby reducing the architected register state.

The processor also contains two classes of the TLB structures to maintain TLB state for data and instructions, respectively [103]. The data TLB arrays consist of two 512-entry, 2-way associative structures, and one 16-entry, fully associative structure. The fully associative structure is often used to hold locked OS translations, and can support multiple page sizes (8KB, 64KB, 512KB, and 4MB), while the 2-way arrays can only support one type of page size. Solaris 9 use these 2-way structures to hold only 8KB pages. The instruction TLB consists of a 2-way 128 entry structure, exclusively for 8KB pages, and another 16-entry fully associated entries that supports multiple page sizes.

These TLBs are accessed using several TLB control registers, including an ASID (address-space identifiers). Each of the TLB entry is tagged by an ASID, which is matched with every incoming TLB translation request. Because of this TLB tagging, the entire TLB state need not be communicated between on-chip cores with the other processor state, which includes all the TLB control registers. Next section will discuss the specific mechanisms implemented to transfer computation between on-chip cores.

3.3.1.2 Computation Transfer

A major function of the VMM component discussed above is *Computation Transfer* (CT): the movement of the state of a computation from one core to another. There are two specific types of state a thread may carry: *architected state* and *microarchitectural state*. Architected state consists

of a thread's memory and the processor state discussed above, and must be preserved. A thread's memory state can simply be communicated as needed via the on-chip coherence network already present to support shared memory multiprocessing. Processor state that includes the register state, must be saved and restored, similar to an OS saving the state of a process when it is context-switched. Because of the tagged TLB state in the SPARC V9 based architecture, it need not be communicated, as long as the TLB control registers are communicated correctly. Microarchitectural state, consisting of predictive state such as cached data and branch predictor entries, need not be preserved for correctness. Unlike thread migration, CT is performed under hardware/firmware directives, transparent to the system software. The entire process is carried out in two parts: saving and restoring.

Saving To initiate a CT, the register state in the processor must be saved. The OPMS framework sets aside a specific portion of the physical memory, for each VCPU, for holding the register state of a computation, much in the similar spirit of the Intel VT [111]. After a CT is triggered, the processor pipeline is flushed, and the control of execution is transferred to the firmware. The firmware issues a series of store operations, that save each register to its allocated memory location, for that VCPU. These memory locations are typically cached, since they were accessed during the start of this computation on the current processor. After saving the entire register state from the current core, the firmware initiates a restore of this state on the destination core for this computation, assigned by the specific assignment policy.

Restore Before beginning the restore operation, the firmware first ensures that the destination core is not executing any other computation. To restore the register state, the firmware issues a series of load instructions. Each load instruction access the specific memory location assigned for the destination register where the value is loaded, depending upon the particular VCPU. After

completing the restoration of the entire state, the computation is resumed, and the next instruction is fetched in the pipeline using the program counter (PC).

In a multicore, state saving is relatively a low latency operation because many of the memory locations accessed are already present in the local private caches (since these were used to load the values at the beginning of the computation). However, as the register state is restored on a separate on-chip core, it leads to a series of cache-to-cache transfers. Nevertheless, due to the superior bandwidth of the on-chip interconnect, these operations are easily pipelined, dramatically reducing the end-to-end latency of CT. The quantitative data on the CT latency, and relative contributions from saving and restoring, are presented later in Section 6.3.1.

Compared to a regular context switch or thread migration in the operating systems, CT is a substantially lightweight operation. In particular, it does not involve any modification to the respective run queues maintained by the system software, and necessary modifications to the kernel data structures that must accompany a thread migration [74]. Conceptually, CT is more similar to a fast context switch between user-level threads, which is shown to be orders of magnitude faster than a full fledged thread migration [42].

3.3.1.3 Assignment Policy

A variety of assignment policies in the OPMS framework is supported via two simple hardware structures for each core: a small RAM, and a small wait queue. The RAM structure maintains the PCPU mappings of the different computations from a VCPU, where the specific applications of the OPMS provides information about techniques to identify a computation fragment (e.g., starting and ending instruction address). This mapping is copied at each core to enable localized decisions, and is initialized by the VMM based on the workload profile. When a VCPU is assigned to an inactive core, the core transitions to the active state and starts executing the VCPU, after restoring

its state as described above. However, if the assigned PCPU is already busy, the execution of the VCPU is paused, and an entry for this VCPU is inserted in the wait queue of the assigned PCPU. Later, when the core becomes available, the control logic inspects the wait queue and resumes VCPU execution. The next chapter provides a concrete example with a detailed walk through that shows how these structures are able to implement the necessary support required for an OPMS application (see Section 4.5.2).

3.3.2 Leakage Control in Inactive Cores

Cores in an OPMS frequently transition between active and inactive states, and therefore an OPMS maintains power supply to the inactive cores for fast startup. By gating the clock signal to a core at the root, one can save most of the dynamic power of an inactive core [63]. However, maintaining the power supply, a necessity to enable fast wakeup, may still cause leakage from an inactive core, potentially defeating the purpose of SAF reduction.

With the increasing importance of static power from leakage, a large body of circuit research, both from academia and industry, has been targeting leakage control in the past few years [30, 85, 93, 110]. There are many leakage control techniques, broadly classified into two categories: device or channel engineering, and circuit techniques [93]. This dissertation focuses on circuit techniques only, as the device engineering techniques will affect all implementations equally.

The most important goal of leakage control techniques in an OPMS is to allow fast wakeup. Within an inactive core, there are two classes of structures that differ on their functions during inactivity. Predictive structures (such as caches and brand predictors) retain their state, while pipeline logic components, like register files and instruction window, do not need to do so. This distinction in state retention affects the choice of leakage control techniques on them.

Among various circuit techniques for controlling leakage, one of the most relevant techniques

for an OPMS implementation is Multithreshold-voltage CMOS (MTCMOS) based on sleep transistors. This technique is particularly promising when state retention is not a requirement [93]. Several current microprocessors are already aggressively incorporating MTCMOS [81]. When state retention is important, as for predictive structures, drowsy mode operation is more applicable [33]. This section will focus the discussion on these two techniques, and their engineering trade-offs in the OPMS context.

MTCMOS A MTCMOS circuit inserts a low leakage high-threshold (V_{th}) device in series to a low threshold circuitry (e.g., an ALU or a register file). This high V_{th} device is known as the *sleep transistor*, as it is controlled by a sleep signal. During normal operation, the sleep control is disabled, allowing the high V_{th} transistor to transmit and deliver the power supply to the low V_{th} circuitry. When the sleep signal is enabled, the sleep transistor turns off, cutting the power supply. As a result, the leakage becomes negligible in the low V_{th} circuitry during this period, which brings down the overall leakage dramatically, within 1-2% of active operation when the sleep signal is disabled [30].

In addition to its excellent leakage control, MTCMOS circuits can be enabled (or disabled) within a few processor cycles [30, 85, 110], making it very effective for an OPMS implementation. However, there are a few cost components that must be considered. First, there is an area overhead for inserting the sleep transistor, and delay overhead from its series insertion. Previous studies have shown that these overhead can be up to 4% for area, and 10% for delay [110]. With an ever increasing effective area, and delay improvement from technology scaling, both these overheads are well within tolerable limits in the context of growing importance of static power, and the increasing demand for energy efficiency [55, 93].

Second, there is an energy overhead to transition between active and sleep modes, due to the charging and discharging of the sleep transistors. As a result, there is a trade-off in the amount of

leakage energy saved by transitioning into the sleep mode, and the transition energy. Typically, the designers assign a *breakeven point* for a circuitry, to denote the amount of time it must be in sleep mode to realize overall energy savings [110]. The breakeven point can be up to a few hundreds cycles [120], although techniques to reduce transition energy overhead can substantially reduce it (by up to 50% [85]).

Drowsy Caches The basic idea behind drowsy cache design is to dynamically adjust the supply voltage depending on the frequency of access [33]. Drowsy mode reduces leakage substantially while retaining predictive state, but allow slower access to them. When the cache state is not accessed with regular frequency, as is the case in an inactive core, the predictive structures can be put in a drowsy mode. The state can be brought into normal active mode, but such transitions come with slight energy overhead.

Based on these considerations, the OPMS framework incorporates aggressive leakage control using MTCMOS technique in all the major functional blocks (e.g., register files, instruction window, functional units) of the cores in an OPMS. The sleep signal is enabled when a core transitions to inactive state. The core's L1 caches and predictors, which retain their state during inactive periods, are put in a drowsy mode [33]. This dissertation will characterize the periods of activity and inactivity during evaluation (see Chapter 6), so as to demonstrate the viability of MTCMOS techniques in an OPMS.

3.3.3 di/dt Impact

Apart from controlling the leakage, another challenge in implementing OPMS is the potential increase in supply voltage fluctuation due to the increased di/dt impact or current variation [64]. An OPMS design can negatively impact the variation in current drawn from the power supply net-

work, because of frequent core transitions. However, these transitions are a result of Computation Transfer, where one processing core becoming inactive is usually accompanied by another core becoming active. Therefore, while active to inactive transition causes a drop in current drawn from the power supply, the accompanying transition in another core, from inactive to active, results in a surge in the current. Collectively, these two events partially compensate each other, and later this dissertation will show the extent of di/dt variation in an OPMS, relative to the baseline (see Chapter 6).

3.4 OPMS: Application

The design paradigm of an OPMS is applicable in a variety of multicore designs, spanning both asymmetric and symmetric multicores. At a high level, the previous chapter demonstrated the effectiveness of this design paradigm for asymmetric multicores (see Section 2.4). In the context of the multithreaded applications, it assumed the presence of generic hardware accelerators that can speed up different computation modes. This dissertation will now focus on how to realize such an execution from structurally identical processing cores in a homogeneous multicore, using the OPMS design paradigm. The dissertation proposes a classic application that utilizes an OPMS by improving the computation and energy efficiency through locality-aware computation assignment. Although enhancing the locality of memory references is its primary target, implementing this application in an OPMS naturally leads to an intermittent use of on-chip cores, substantially improving the thermal profile of their micro-architectural components. The next section describes a basic outline of this application, while the rest of the dissertation will present more detailed design, with in-depth quantitative evaluations.

3.4.1 Computation and Energy Efficiency via Locality Enhancement

Traditionally, the OS schedules software threads onto a group of *Virtual CPUs (VCPUs)*, which map one-to-one with the cores, or *Physical CPUs (PCPUs)*, on the chip. However, in an OPMS, the aggregate PCPUs exceed the number of VCPUs, allowing a designer to consider a more selective assignment of computation to enhance the locality of memory references. In particular, this dissertation propose to provision a set of PCPUs for processing OS computation, and the rest for user computation, as these two modes are known to cause destructive interference in the core's predictive structures [36,67]. At any given time, only a subset of these cores are active, performing their assigned computation, while the rest retain their predictive state in a low leakage mode. This mode of distributing computation is a special application of *Computation Spreading* [24], and the next chapter discusses the general model with this application in details.

3.5 Summary

The design paradigm of an Over-provisioned Multicore System captures the shifting trends in the underlying semiconductor technology where integrating cores on a single die comes with a significantly lower marginal cost than the engineering and economic cost to power them up. At a high level, this trend indicates a major change in our comprehension about the critically of the system resources: *chip power* becoming more critical than individual processing cores on a single multicore die.

This chapter introduced the OPMS framework, discussed its major challenges and opportunities, and details of an implementation evaluated in the rest of this dissertation. An OPMS creates a constantly varying pool of computation resources, leading to implementation issues with interfacing with the system software, and power efficient management of inactive core resources.

To mitigate the first problem, the chapter described a solution based on virtualization technique that performs a part of the virtualization—processor core management—through a programmable hardware component. This solution enables supporting system software and application stacks transparently, while limiting the hardware implementation complexity. For the latter, the chapter proposed to incorporate aggressive techniques for leakage control, an increasingly popular trend even among current implementations as well.

The key to understanding the concept of an OPMS is its ability to enable flexible assignment of computation, and innovative use of inactive cores in future multicore systems. Inactive cores can add substantial value in the performance metrics, via their unique abilities to retain predictive state and logically function as a generic accelerator, offering thermal slack, and provide heterogeneity in execution capabilities.

Chapter 4

Computation Spreading

An Over-provisioned Multicore System offers an unique opportunity of dynamically specializing on-chip cores via retaining frequently accessed but mutually exclusive predictive state in each of them. To exploit this unique opportunity, a designer must inspect the prevailing policies of assigning computation on multiple cores, and the technology trends of applications.

In the canonical model of assigning computation from multiple threads to multiple processors, an entire software thread — including any operating system calls it makes — is assigned to a single processor for execution. This was, perhaps, the only practical approach for traditional multiprocessors built from multiple chips. But since there is commonality among the computation performed by different threads (i.e., commonality in the code executed), this distribution leads to inefficient use of the microarchitectural structures of the individual processing cores that are intended to exploit the properties of instruction streams (e.g., private instruction caches and branch predictors). Consequently, despite a large increase in aggregate private resources in microprocessor chips, modern applications are unable to efficiently exploit them.

On the other hand, current applications show a tremendous growth in their code and data footprints [12, 90], exacerbating the problem of inefficient resource usage. In addition, commercial

server class multithreaded applications make extensive use of the operating system services via system calls, and mixing these different instruction streams on the same micro-architecture causes destructive interferences [2, 8]. Collectively, these trends resulted in rapidly increasing memory stalls in these applications. Indeed, memory stalls are now widely regarded as the most important bottleneck for achieving high performance in server applications [3, 65].

To illustrate a classic application of the OPMS design paradigm, this dissertation proposes *Computation Spreading (CSP)*, a new model for distributing different fragments of a thread's computation across multiple processing cores. For the purpose of describing the CSP model, this dissertation defines a *computation fragment* as an arbitrary portion of a dynamic instruction stream. Conceptually, CSP aims to collocate similar computation fragments from different threads on the same core while distributing the dissimilar computation fragments from the same thread across multiple cores. Each core in the OPMS becomes dynamically and temporally specialized for executing a set of specific computation fragments by retaining the states (such as instruction cache contents and branch predictor entries) necessary to perform each computation efficiently. Furthermore, by reducing the activity on each core through inactive periods where it retain the predictive state, thermal characteristics of each core are substantially improved.

This chapter develops the model for CSP, and presents a concrete application with its implementation details, while the evaluation results will be presented in Chapter 6. Section 4.1 presents the code reuse characteristics of several multithreaded applications running on a canonical multicore. Next, Section 4.2 discusses the general model of Computation Spreading, highlighting the key design considerations. Guided by the design requirements, this dissertation demonstrates a specific application of Computation Spreading where it proposes to separate user and OS computation in Section 4.4. The implementation details is presented in Section 4.5, and finally this chapter concludes with a summary in Section 4.7.

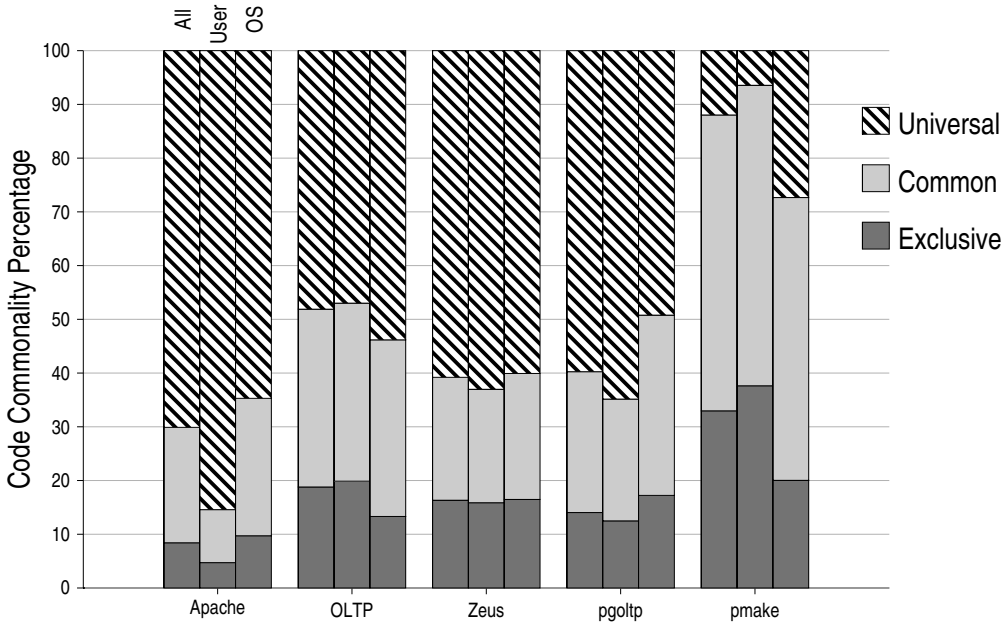


Figure 4.1: Code Commonality Characteristics for a multicore with 8 cores. *Each 64-byte static instruction block is marked* Universal *if accessed by all the cores,* Common *if accessed by more than one core, and* Exclusive *if accessed by only one. The left bar shows the commonality profile for the entire execution, the middle bar shows commonality for only user code, and the right bar shows the commonality for OS code.*

4.1 Multiprocessor Code Reuse

In the canonical software architecture for multithreaded server applications, a software thread is launched to process an input data item or request. Multiple threads are used to process multiple requests. A thread is scheduled for execution on a single processing core, and all the phases of execution for that thread are typically run on that core (in the absence of OS-induced thread migration). In a multiprocessor, concurrent threads execute on different cores, if possible. Since these separate threads are likely to act on many requests that are often similar, they traverse through similar code paths (possibly with different data) and end up executing the same instruction blocks.

To quantify this code commonality, instruction accesses are profiled for five multi-threaded workloads on a conventional multicore system with eight processing cores, and examine how many processing cores fetch each (static) instruction cache block.[1] Figure 4.1 shows this commonality for the overall execution (left bar), application-level, user code (center) and operating system code (right). Each bar shows the fraction of 64-byte instruction blocks fetched by a core that are fetched by all the processing cores (*Universal*), by more than one core (*Common*), and by only one core (*Exclusive*).

Several observations can be made from the figure. First, there is a significant amount of commonality in the code executed on the different cores, both for user code and system code. Second, there is little code that is exclusively executed on a single processing core. Third, the figure shows that the extent of universal sharing is more pronounced for OS code than for user code in three benchmarks (OLTP, pgoltp, and pmake) while the opposite is true for Apache. This extensive sharing of instruction blocks in database workloads running on simultaneous multi-threaded (SMT) processors was also observed by Lo, et al. [71].

Commonality in the code executed by the different processing cores results in the same instruction blocks being referenced by different cores. This leads to two distinct outcomes: (i) *reference spreading* [70], which refers to the multiple references to a single cache block across multiple processors, leading to a cache miss in each of those caches, and (ii) the replication of the code in multiple caches and subsequent reduction in aggregate cache space. This combined effect can drastically degrade the hit rate, especially for applications with large instruction footprints.

Table 4.1 presents the instruction footprint seen in these workloads. For each workload, the table presents the total instruction footprint, the user code footprint, and the OS code footprints, respectively. OLTP and pmake show large code footprint, each of them exceeding 1MB. However,

[1] The workloads and the target multicore system are described in more detail in the Chapter 5.

in pmake the entire code footprint is not as much spread across multiple cores as compared to OLTP (see Figure 4.1), allowing each core to see a smaller subset of the entire code, and observe better locality of references. The OS-intensive workloads, Apache and Zeus, exercise relatively larger regions of the OS code, while exhibiting significantly smaller footprint from their user code.

Benchmark	Overall	User	OS
Apache	603KB	157KB	445KB
OLTP	1291KB	1079KB	212KB
Zeus	560KB	137KB	423KB
pgoltp	724KB	487KB	237KB
pmake	1320KB	971KB	349KB

Table 4.1: Instruction Footprint

To mitigate the effect of code replication in multicore systems, this dissertation proposes *Computation Spreading* to distribute the collective work in a way that can improve the locality of references, while dynamically creating a cooperative execution framework to efficiently utilize on-chip resources.

4.2 CSP: An Overview

The idea behind *Computation Spreading* (CSP) is to spread out the dissimilar, and perhaps independent, computation fragments of a single thread across multiple processing cores, and concentrate similar fragments from multiple threads onto a single processing core. A depiction of CSP is illustrated in Figure 4.2. Consider three threads of execution: $T1$, $T2$ and $T3$. Each thread goes through three specific fragments of computation during a given period of time. In the canonical model for mapping computations, the entire execution of a thread is mapped onto one processing core (for example, $T1$ is mapped onto $P1$). In CSP however, the execution is mapped according to the specific computation performed. In this particular case, the similar computation fragments

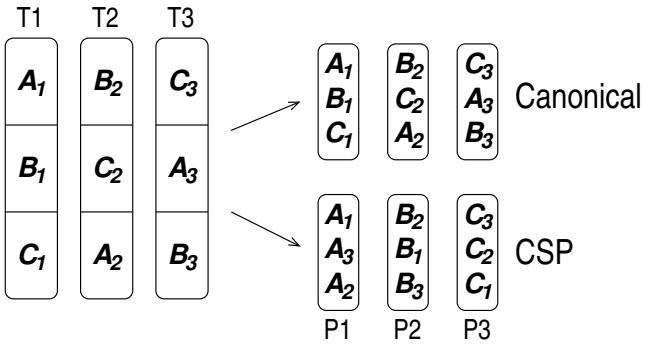


Figure 4.2: Computation Spreading vs. the Canonical Model for mapping computation fragments from a thread. A_1, A_2 *and* A_3 *denote similar fragments from different threads;* A_1, B_1 *and* C_1 *denote dissimilar fragments of the same thread.*

A_1, A_2 and A_3 are all mapped to $P1$, and when any of the threads execute these similar fragments, they are preferably executed on $P1$. Likewise, the other computation fragments are mapped to $P2$ and $P3$. Thus, dissimilar computation fragments (e.g., A_1 and B_1) are assigned to different processing cores, allowing each core to specialize its microarchitectural resources for a particular type of computation fragment.

In general, a well-written, modular program is composed of a set of fairly disjoint fragments (e.g., functions in high-level language). Two fragments can be considered similar when they carry out a related set of tasks, and thus traverse closely related code paths and/or have frequent data communication. Likewise, two fragments performing unrelated tasks, with little common code, will be inferred as dissimilar (e.g., page fault handling and process scheduling).

4.3 CSP: Design Issues

There are several key design aspects for CSP that dictate viable choices of computation fragments.

Dynamic Specialization By localizing specific computation fragments on a single core, CSP allows that core to retain many predictive states necessary for the efficient execution of those fragments (like instruction cache contents and branch predictor states). Collectively, different on-chip cores, with the same microarchitecture, become dynamically and temporally specialized for different computations, leading to more efficient overall execution.

Data Locality Since CSP attempts to exploit code affinity as opposed to data affinity [109], data locality can potentially suffer. However, different parts of a large computation do not communicate arbitrarily. Thus, it is possible to retain, and even enhance, the locality of data references through careful selection of the computation fragments.

Fragment Size The size of computation fragments plays a key role in determining the engineering trade-offs in CSP. One of the main features of CSP is to realize that often moving computation to a different core is beneficial because of the predictive state present in that core. However, this movement comes at the cost of moving the register state and potentially other predictive state. Since the movement of collective state, which includes both register state and predictive state, must be minimized to realize performance benefits, there is a design trade-off dictating the frequency of computation transfers. CT events must not be as frequent, so as to defeat the purpose of CSP by degrading performance. It is thus expected, computation fragments executed at each core should be of reasonable size, although the precise size will vary for different multicore implementations.

Core Contention The canonical system uniformly distributes the collective computation using a threading abstraction in the application program. CSP breaks this thread based work distribution, relying on similarity and dissimilarity in the actual computation performed. Different computation fragments are assigned to different cores. Since the software stack above still exposes computa-

tion based on threads, the utilization of these on-chip cores depends on the specific computation performed in those threads at runtime. Occasionally, multiple software threads may execute a particular computation fragment, creating resource contention on the core provisioned for that fragment.

Guided by these design considerations, next section presents a specific example of CSP for this dissertation. In particular, this example exploits the natural separation between user code and privileged, OS code, and assign separate processing cores for carrying out these distinct fragments of computation. It may be possible to identify other, more general, fragments in various ways, such as profile driven binary annotation or high-level directives from the application itself. However, such general classification of computation fragments is a separate research topic on its own, but the framework developed in this dissertation can exploit arbitrary work distribution based on it.

4.4 An Application of CSP: Separating User and OS Execution

4.4.1 Why Target User and OS?

Targeting CSP at the user/OS code boundary is appealing for several reasons. First, many commercial workloads spend a considerable amount of time executing both user and OS code, and frequently switch between the two (Table 4.2). Both the fraction of execution time, and the average length of computation phases in OS and user code, vary across these benchmarks. For example, Apache and Zeus spend 61% and 74% of their execution time in the OS, respectively. In contrast, OLTP and pgoltp spend only 16% and 14% of their execution time in the OS, respectively. The length of the user phases is substantially smaller in Apache and Zeus, compared to the rest of the benchmark. Compared to OLTP, pgoltp and pmake exhibit much longer computation phases in the user code.

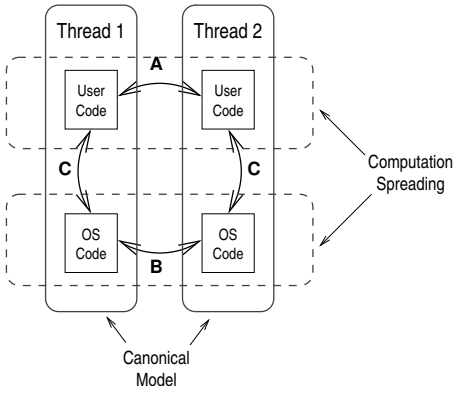
	Percent of Instr.		Instr. Before Switch	
Benchmark	**User**	**OS**	**User**	**OS**
Apache	39%	61%	2.3k	3.6k
OLTP	84%	16%	9.3k	1.8k
pgoltp	86%	14%	22.1k	2.9k
Zeus	26%	74%	1.7k	5.0k
pmake	83%	17%	24.2k	4.8k

Table 4.2: Breakdown of user and OS instructions, and the average number of instructions executed before switching from one mode to the other (excluding short TLB and register-window traps).

Second, there is a clearly defined separation between these two modes of computation that is easy to identify and exploit with minimal profiling requirements. Third, this clear separation leads to mutually exclusive instruction cache contents and branch predictor states of user and OS code. By collocating these OS and user fragments of computation, however, the canonical model causes extensive destructive interference in these two micro-architectural structures as observed by us and others [2, 67]. And finally, though seemingly counter-intuitive, OS and user fragments have limited data communications between each other, and therefore have a reasonably independent data footprints.

Figure 4.3(a) illustrates the nature of data communication between OS and user computation in the multicore system modeled in this dissertation. Consider two threads running on two different cores in the canonical model. Each thread performs both user and OS fragments of computation; in all there are four separate types of fragments (two from each thread). The canonical model groups the user and OS computation fragments of a given thread together on one core (shown by the solid rectangles), whereas CSP groups user fragments together, separate from OS fragments (shown by the dotted rectangles).

The arcs represent communication between the various computation fragments. Arc A, user-to-user communication, and arc B, OS-to-OS communication, are both a result of directly accessing



(a) *Comparing Models*

Benchmark	A	B	C	% of Refs.
Apache	1.78	5.58	1.00	4.89%
OLTP	7.48	4.40	1.00	3.95%
Zeus	0.81	12.8	1.00	7.43%
pgoltp	3.03	8.62	1.00	2.22%
pmake	4.42	1.04	1.00	1.69%

(b) *Normalized Relative Communication*

Figure 4.3: Communication Profile between OS and User. *Labels, in both the figures, denote different types of communication: between user computation fragments of two threads (A), OS fragments of two threads (B) and between user and OS computation fragments (C), both in the same and different threads. The relative communication between these three groups is shown in (b).*

locks and data structures that are shared among threads. OS-induced thread migration also contributes slightly to A and B. Arc C is a result of the OS reading and writing data structures passed as pointers to or from a system call.

Figure 4.3(b) presents the amount of communication between various computation fragments, relative to the communication of user code with OS code (the arcs labeled C). The figure defines the communication as any data access (load or store) to a cache line whose previous access was from a different type of computation fragment. For comparison, the last column shows the percent of *all* data references represented by the sum of these three types of communication. While this fraction is small, note that communication between fragments on different cores *always* leads to a cache miss, and is a major contributor to overall memory latency.

In every case except Zeus, there is more communication between user code on different threads

than there is between user and OS code (regardless of which thread). Similarly, in every case, the figure shows more communication between the OS code of different threads than between user and OS code.

For an I/O intensive, data streaming application such as Apache, the fact that there is little communication between user and OS code may seem counter-intuitive. But consider that Apache predominantly uses the *sendfile()* system call to tell the OS to copy data from one file descriptor to another: the user code simply directs the data copying, but the OS performs the copy entirely within the OS computation fragment.

When choosing a boundary for spreading execution among cores (which is required in any model to achieve parallelism), it appears logical, given the data in Figure 4.3(b), to divide user and OS execution among cores rather than keep user code or OS code from the same thread on a particular core and arbitrarily distribute threads among cores.

4.5 CSP Implementation

This section now discusses the specific details of CSP implementation. First, it gives the details of scheduling policies employed in the evaluation for this dissertation, and then it illustrates how these policies are implemented in the OPMS framework.

4.5.1 CSP Policies

Targeting CSP at OS and user executions provides a simple means to determine when the VMM should consider moving a thread (as the thread enters or leaves the OS). User code invokes the OS for several reasons, such as system calls, interrupts or exceptions. SPARC V9 uses software handling of TLB misses and register window fills and spills, and these short-running traps dominate

user-to-OS transitions. For the purpose of this dissertation, the implementation considers these traps as user computations (unless a page fault occurs), since they are short running and many other architectures handle them in the hardware.

System calls, interrupts, and page faults — the events which trigger computation transfer in CSP — are relatively frequent and long-running. Table 4.2 shows, on average, how many user instructions pass between these events, and how many instructions are spent processing these events. When these events occur, the core notifies the VMM, which then directs a CT to another on-chip core as dictated by the specific assignment policy. At the end of the OS computation, the VMM initiates another CT to resume computation on the original core.

In all of the experimental evaluations, OPMS cores are statically partitioned between the user and OS code based on their relative execution time and cache behavior. For this study, the evaluations have used an OPMS configuration with 12 on-chip cores, which allows simultaneous computation on only eight of them. The multi-threaded workloads are configured in a way to have eight running threads (or VCPUs), identical to the baseline. Methodological details including the experimental setup and core provisioning for each workload is presented in Chapter 5, while the results from evaluation is presented in Chapter 6.

The dissertation evaluates two different assignment policies in this study: Thread Assignment Policy and Syscall Assignment Policy.

Thread Assignment Policy (TAP): The most straight-forward way for splitting user and OS execution from a thread is to maintain a static mapping of OS and user cores for every thread. At any given time, there are eight computation fragments, from both OS and user, to map onto a set of four or eight cores, as the case may be. For example, Apache is provisioned with four user cores, so two of the eight threads execute their user fragments on each core. The OS portion of the eight threads needs to be spread across eight cores uniformly.

Syscall Assignment Policy (SAP): Though TAP is straight-forward, further dynamic core specialization is possible by grouping similar OS computation fragments onto a subset of the OS cores, and spreading dissimilar OS fragments to the other OS cores. The similarity of different OS fragments is simply detected by recognizing the specific system call (based on register `%g1` in Solaris), or interrupt handling routine, being invoked. For this dissertation, the implementation uses a static mapping of particular system calls (plus interrupts and page faults) to cores. This assignment policy allows a particular system call to be mapped to one or more cores, and will only execute that call on those cores. The policy also allows other system calls to be mapped to the same cores. The dissertation refers to this policy as *Syscall Assignment Policy (SAP)*.

During the startup, most common OS computation fragments are provisioned onto different cores based on their L1 data cache miss component in the baseline (this matches well with the time spent in those computation fragments). Other system calls that are detected at runtime, are assigned to a single core in a round-robin fashion. This provisioning scheme aims to distribute the resource demands across different OS cores evenly. Again, online profiling can be used to perform this dynamic assignment.

4.5.2 Implementation on an OPMS Framework

Much of the required support for CSP is already provided by the OPMS framework, including computation transfer on computation mode change (i.e., OS to user or user to OS), and maintaining dynamic mapping between VCPUs and PCPUs. However, some additional support is required to ensure correctness during contention in CSP. Contention can happen in CSP due to simultaneous assignment of multiple computation fragments to the same core. For example, when the VMM decides to move the computation from a thread onto a core that is already executing another thread, the former must be paused till the core is vacated.

The scheduling employed in CSP is implemented using the hardware support provided by the OPMS framework (see Section 3.3.1.3). For CSP, the RAMs maintain the two PCPUs (one for user, one for OS) that are currently assigned to each VCPU. The exact mapping is initialized by the VMM component, based on the workload profile. When a VCPU is assigned to an inactive core, the core transitions to active state and starts executing the next computation from this VCPU. However, if the assigned core is already busy, the execution of the VCPU is paused, and an entry for this VCPU is inserted in the wait queue. Later, when the core becomes available, the control logic inspects the wait queue and resumes VCPU execution. This implementation is explained with an example next.

4.5.2.1 Handling Contention

Figure 4.4 shows how CSP handles execution during contention. For simplicity, the figure illustrates computation from two VCPUs: $VC1$ and $VC2$. Collective computation from these two VCPUs are mapped to three physical cores or PCPUs: $PC1$, $PC2$, and $PC3$. In this example, the user computation from $VC1$ and $VC2$ are assigned to $PC1$ and $PC2$, respectively. But only $PC3$ is provisioned for the OS computation from both the virtual cores. $PC1$ and $PC2$ do not experience contention as only one VCPU's computation is provisioned on them. However, $PC3$ can experience occasional contention, and the figure shows the wait queue, $VQ3$, for this core.

Different numbers mark the sequence of important events. At the beginning, both the virtual cores are executing user code on $PC1$ and $PC2$, respectively. The wait queue $VQ3$ is empty (marked by E), as none of the computation is waiting to execute on $PC3$. At event 1, $VC1$ makes a system call, which results in a computation transfer. The register state of $PC1$ is saved, and later, marked with event 2, restored at $PC3$, before resuming the OS computation. At 3, $VC2$ makes a system call, and as per the current assignment, seeks to run its OS code on $PC3$. This event

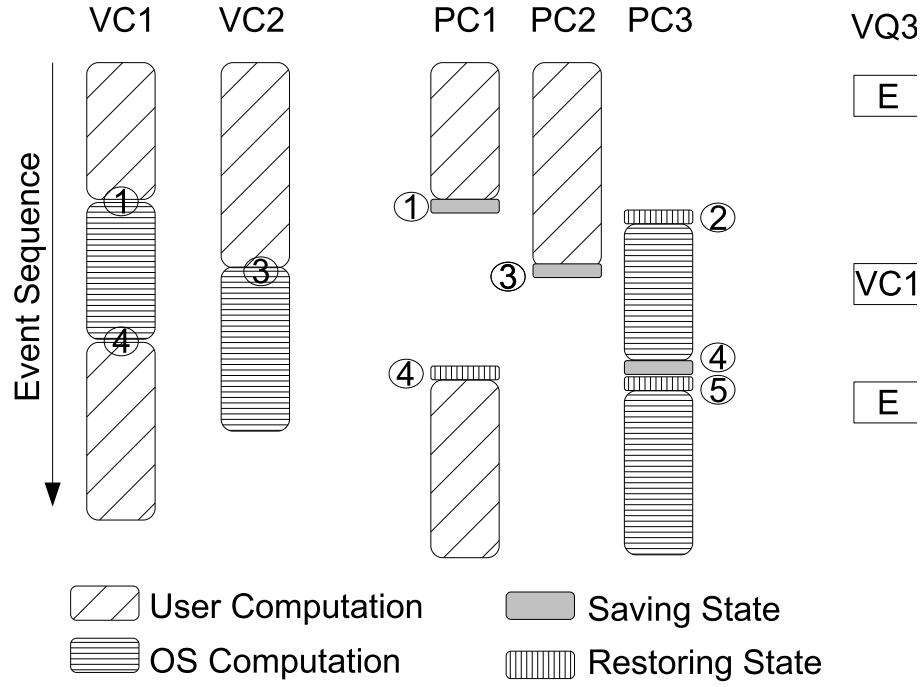


Figure 4.4: CSP Implementation: Handling Contention. Two virtual cores, $VC1$ and $VC2$, are assigned to an OPMS with three physical cores: $PC1$, $PC2$, and $PC3$. The user computation from $VC1$ and $VC2$ are assigned to $PC1$ and $PC2$, respectively, while the OS computation from both the virtual cores are assigned to $PC3$.

creates a contention for $PC3$, as it is already executing the OS code from $VC1$. As a result, the wait queue $VQ3$ is populated with an entry for $VC2$, after saving its register state at $PC2$.

At a later point, marked with 4, the OS computation from $VC1$ completes at $PC3$, triggering a computation transfer from $PC3$ to $PC1$ where the user computation resumes. When $PC3$ is vacated, the core logic inspects the VCPU wait queue $VQ3$, and finds an entry for $VC2$. Subsequently, it restores the register state of $VC2$, and resume its OS computation on $PC3$, and the wait queue again becomes empty (marked with 5).

This contention overhead is significant when CSP is used in a conventional multicore where

all the cores in the system are exposed to the OS for scheduling threads. Since CSP recognizes multiple computation phases from the same thread, the number of possible computation phases exceeds the aggregate on-chip cores substantially, causing a large increase in the contention overhead. However, OPMS alleviates the contention overhead by provisioning more on-chip cores, and enabling a much superior flexibility in assigning computation. Chapter 6 will present detailed quantitative results on the contention overhead, and demonstrate how OPMS dramatically reduces this overhead in all the benchmarks by using 50% additional cores and application aware core provisioning, without exceeding the power envelope of the baseline system.

4.6 Related Work

Several recent studies propose to redesign a software application to enforce staged execution, instead of monolithic execution. Such an optimization can significantly improve the performance of server applications, much of which is dominated by memory stall times (with significant contribution from instruction stalls) [3, 40]. Many of these proposals aim to achieve similar performance benefits as CSP via reducing memory stalls through rewriting the software, but CSP obviates the need for such software modifications.

One previous work which shares its key motivation with CSP in targeting these stalls is Cohort Scheduling proposed by Larus and Parkes [65]. They argued that the performance problems of server workloads stem from the canonical software architecture of server applications, which results in the execution of non-looping code segments with little instruction and data locality. They propose a new programming model to identify and group related computations and then subsequently schedule them to exploit the similarity between successive computations. SEDA, proposed by Welsh, et. al., decomposes an event-driven server application into multiple stages and schedules them on different processors to enable code reuse and modularity while improving performance

and fairness of the system [117]. Harizopoulos and Ailamaki [42] exploit the recurrence of common code paths in transaction processing by identifying regions of code that fit in a L1 I-cache, and frequently context switching among threads to reuse the cache contents. TAP and SAP simply distinguish between user and OS computation and employ computation transfer to localize these computations at the hardware level, without specific information about, or modifications to, the software architecture of server applications.

Computation spreading is also quite similar in spirit to LARD [84], which uses locality information at a front-end server to distribute web requests among back-end servers. Load balance is also a major issue with LARD, however it is easier to address because the number of requests is much greater than the number of servers.

Separating OS computation from the user computation is reminiscent of the CDC 6600, which had one high performance Central Processing Unit (CPU), and several lower performance, multithreaded Peripheral Processors (PPs) [107]. The intent was to run threaded, I/O bound, OS activities on the PPs, and save the CPU for heavy computation. When technology eventually allowed an entire processor to fit on one chip, however, the communication costs prohibited executing OS and user code on separate chips.

There have been many studies of OS behavior, and its interaction with the user code on various micro-architectural features [2, 8, 36, 67, 92, 98, 108]. The salient points of most of this research are that 1) many cycles are spent executing operating system code for server applications, 2) OS code has different behavior than user code, and 3) the OS adversely affects many micro-architectural structures through capacity and conflict interference. The study presented in this dissertation corroborates most of these claims, and it advocates CSP as a potential unified solution to these interference problems.

Several recent proposals aim to improve memory value communication in multicore systems.

Most rely on coherence protocol optimizations and/or alternate cache organizations [14, 26, 100]. This dissertation proposed an orthogonal technique to achieve similar goals.

Recent proposals also advocate using heterogeneous CMP cores to achieve performance and power benefits by scheduling the low ILP phases on the slower cores [63]. CSP is a promising match for such hardware, since different computation fragments (such as user and OS) typically have different resource requirements.

4.7 Summary

This chapter examined the code reuse characteristics of several multithreaded commercial workloads, and showed that most instruction blocks are accessed by many, if not all, on-chip cores. Such extensive code commonality indicated that different cores went through similar phases of computation (albeit at different times), and that the canonical model of work distribution led to inefficient use of the aggregate cache space.

The chapter presented *Computation Spreading* (CSP), a new model for distributing a thread's computation across multiple cores in an OPMS. CSP employs hardware thread migration to spread dissimilar phases of computation from a thread to multiple CMP cores, while localizing similar phases of computation from different threads onto a single core. This technique is applied to separate user computation from the OS computation performed on behalf of the user, and present two assignment policies for this application: TAP and SAP. TAP separated user and OS computation, and subsequently assigned them on physical cores, while maintaining a unique mapping between the OS and user computation from a given VCPU onto a particular core, respectively. SAP aimed to further specialize the cores responsible for OS computation, by assigning OS computations based on system calls, rather than the VCPUs they belong to.

This chapter then presented implementation details, especially focusing mechanisms to ensure

correct execution during core contentions, which arise due to the selective choice of on-chip cores for a given computation, enforced by CSP. While conventional multicores can degrade the performance benefit from CSP due to this contention overhead, OPMS dramatically reduces these overheads, allowing multicores to exploit CSP. Quantitative results for CSP is presented later in Chapter 6.

Several recent proposals aim to achieve locality enhancement for memory references through a complete re-write of the software [65, 84, 117, 117], but CSP is the first of its kind that exploited the superior communication in multicore systems to dynamically specialize on-chip cores, and realized these benefits without requiring any software modification.

Chapter 5

Methodology

All the experiments in this dissertation are carried out using a full system, cycle accurate, execution driven simulator that models multicore systems with 8 and 12 processors. The workloads evaluated consist of server class commercial multithreaded applications running on Solaris 9 Operating System for SPARC V9 platform. This chapter presents the methodological details used for the experiments in this dissertation.

The chapter is organized as follows. Section 5.1 presents a detailed description of the workloads used in this study, along with their respective configurations. Section 5.2 presents the details of the simulation infrastructure used in this dissertation. Section 5.3 describes the micro-architectural details of the multicores used in this dissertation. Section 5.4 presents the methodological details of the energy estimation used for this study. Finally, Section 5.5 presents the details of the OPMS configuration, and the specific core provisioning employed by the CSP policies.

5.1 Workloads

Table 5.1 describes the workloads used in this dissertation. Other than pmake, rest of the workloads represent multithreaded server class applications. The benchmarks are setup for a baseline multi-core system with eight processing cores (described later in Section 5.3), where the OS schedules eight concurrent threads from the application. Although the OPMS evaluated in this dissertation contains 12 on-chip cores, only eight of those core are exposed to the OS, which allows the evaluation to use of the same checkpoint as the baseline even for the OPMS. Only the results in Section 6.6.2 uses a slight modification of this general setup, and the differences are described in the next section.

All the benchmarks are set up on Solaris 9 and initially run for a long duration (up to several simulated minutes) to warm up the applications and OS disk caches. Subsequently, memory traces are collected for several simulated seconds. These traces are used to warm up the L2 caches in respective multicores before the start of the timing runs.

Detailed timing simulations are conducted for a specific number of workload transactions or a corresponding measure of application work. For Apache and Zeus, evaluations use 1300 workload transactions. Since user committed instructions are shown to be a good proxy for actual work done in these application [119], the rest of the benchmarks are run for 100 million committed user instructions. Due to the inherent variability from full-system execution [4], all evaluations add a small random variation to the main memory latency, and run several trials of each benchmark per experiment. Generally the average results are presented, but appropriate results also show the 95% confidence interval.

Apache	The Apache web server version 2.0.48 is used to deliver static file contents to a number client threads from Surge [11]. No dynamic processing (e.g., running CGI scripts) is done at the back end. To reduce the CPU idle time in the benchmark, the clients do not use any think time. Both the server and clients are running on the same multicore system modeled.
OLTP	OLTP represents a database back end running online transaction processing queries modeled after TPC-C. IBM DB-2, which is a commercial back end database engine, is used as the back end. The database is scaled down from TPC-C specification to about 800MB and runs 192 concurrent user threads. To reduce CPU idle time the user threads are configured to run without any think time.
pgoltp	pgoltp also runs queries from TPC-C, but uses the PostgreSQL 8.1.3 database [88] driven by OSDL's DBT-2 [83]. Unlike IBM's DB2, PostgreSQL performs I/O through the OS's standard interfaces and utilizes the OS's disk cache.
Zeus	Zeus is a commercial web server, and the clients for Zeus is configured identical to that in Apache.
pmake	Parallel compile of PostgreSQL using GNU make with the `-j 64` flag using the Sun Forte Developer 7 C compiler. Unlike the other four workloads representing multi-threaded server applications, pmake illustrates a multi-programmed workload consisting of multiple processes running in separate virtual address spaces.

Table 5.1: Workloads

5.1.1 Comparing Different System Configurations

For the purpose of comparison, Section 6.6.2 models a multicore system that use the same OPMS micro-architecture, but exposes all its 12 core to the OS for scheduling threads. This change in the system configuration (8-cores versus 12-cores) requires a different workload checkpoints than the baseline and OPMS. Consequently, comparison between this new scheme with the other two schemes (baseline and OPMS) must use different workload checkpoints, imposing a significant methodological challenge in architectural simulation.

To mitigate this challenge, this dissertation adopted a best effort methodology, and presents detailed results on application scaling in Section 6.6.2. There are two key aspects of the strategy

adopted here. First, to build new workload checkpoints, all the applications are setup in an identical manner on a 12-core multicore. The application configurations, including the number of threads, and system level parameters (e.g., buffer size specifications in the database workloads) are kept unchanged. Second, all experiments that include this 12-core configuration are run 5X longer than those reported in the previous section, so as to mitigate any transient differences with 8-core workload checkpoints used in the other multicore systems. In particular, Apache and Zeus are run for 6500 transactions, while the other benchmarks are run for 500 million user instructions.

5.2 Simulation Infrastructure

The simulation infrastructure is built on top of SIMICS, which provides an interface for building detailed full-system simulation models of computers [72]. SIMICS provides the functional component of this simulation infrastructure by modeling the impact of instruction executions, both from the application and the operating system. As a result, system level impacts like interrupts, interfacing with devices are realized in the simulation model through SIMICS. On its own, SIMICS can boot an unmodified Solaris operating system, and execute unmodified commercial applications on a simulated machine modeling a server class platform (e.g., Sun Enterprise 8000 with UltraSPARC IIIcu processors).

The simulation infrastructure augments this functional component with a timing component, greatly increasing the range of microarchitectures that can be modeled in sufficient details. The timing component consists of $82K$ lines of C++ code, built exclusively in the research group led by Dr. Gurindar S. Sohi.

The role of this timing component is to impart the timing characteristics of a detailed core micro-architecture one may wish to model. By interfacing with the functional model provided by SIMICS, the timing model dictates the precise timing of instruction execution, while SIMICS

handles the functional aspects of such execution. In addition to the core micro-architecture, the timing model also builds a detailed model for the cache hierarchy, coherence and consistency mechanisms required to support multiprocessor execution. To provide cycle accurate latency from memory references, several aspects of the on-chip cache hierarchy are modeled in great details including cache bank bandwidth, interconnect latency and resource contention, and main memory bandwidth.

To gain maximum leverage out of the functional component of SIMICS, the timing component uses the Micro-architectural Interface (MAI) of SIMICS. MAI interface provides useful information from the functional components (e.g., register dependency that aids in the scheduling of instructions), while obviating the need to duplicate implementation efforts in the timing components. Using this interface allows seamless integration of register data dependencies and computations, allowing the timing component to only focus on the timing aspects of execution.

Using the MAI interface also offers methodological advantages in the simulation fidelity over the timing first simulation [73], which completely decouples the timing and functional aspect of the simulation. Timing first simulators build an infrastructure that combines both the timing and the functional aspects of instruction execution, while the SIMICS interface is used simply to check the validation of the functional execution, and model system-level impacts. Consequently, the timing characteristics observed by the system in this simulation methodology is distinct from that modeled in simulator. MAI interface alleviates this problem by seamlessly combining the timing and functional components, thereby allowing the system to observe identical timing characteristics.

Implementations based on the SPARC V9 use an architected TLB where TLB misses are handled in the software. Several workloads evaluated in this dissertation exhibit large memory footprint, causing frequent control transfers to the OS for TLB miss handing, and leads to severe performance degradation, often masking other impediments to performance seen across different

platforms. To alleviate this problem, the simulation infrastructure models a hardware filled TLB by caching a large number of previously seen TLB translations, and saves control transfer to the OS by supplying them on-demand when TLB misses happen in the regular TLB structures. When such translations are not present among those seen previously, a regular TLB miss is generated.

5.3 Multicore Systems

The dissertation models a multicore system with eight or 12 on-chip cores, 2-level cache hierarchy, and the interconnect to support memory communication between different caches in the system. Individual processing cores functionally model UltraSPARC IIICu CPUs, but using the timing component in the simulation infrastructure, their core micro-architecture logically enforces timing characteristics of an out-of-order superscalar processor. The baseline multicore system contains 8 such processing cores. Various parameters of this core micro-architecture, along with the baseline multicore is presented in Table 5.2.

Technology Generation: 32nm at 3.00 GHZ and 0.9V	
Fetch/issue/commit	4 instructions / cycle
Integer pipeline	12 stages
I-Window & ROB	128 entries, OOO issue
Load & store queues	32 entries each, w/ bypassing
Store buffer	32 entries, processor consistency
YAGS branch pred.	4k choice, 1k except, 6 tag bits
Priv. L1 instr. cache	64kB, 8-way, 2-cycle, coherent
Priv. L1 data cache	64kB, 8-way, 2-cycle, write-back
On-chip shared L2 cache	16MB, 16 banks, 32-way, 60-cycle load to use, pipelined, inclusive
Coherence Protocol	Directory based MESI
On-chip interconnect	Point-to-point, 25 cycle
Main Memory	255 cycles load-to-use, 40GB/sec

Table 5.2: Baseline processor parameters

The OPMS multicore use a similar multicore system compared to the baseline, with two key

distinctions: (a) a total of 12 cores are provisioned on the multicore instead of 8 in the baseline, and (b) the shared L2 is reduced to a smaller 12MB. The core micro-architecture remains unchanged across the baseline and the OPMS.

The evaluation models a point-to-point logical interconnect where each node of the network is either a processor core or an L2 cache bank of 1MB. In all, there are 24 nodes in the network (fixed between the two multicore configurations). In the baseline system, these nodes are occupied by eight processor cores and 16 banks of the L2 cache. In the OPMS configurations, consisting of 12 cores, four banks of the L2 cache are removed, and subsequently replaced with cores. Therefore, the aggregate L2 cache size in the OPMS is reduced to accommodate additional processor cores. The area overhead from the additional cores and interconnect is compensated by the cache banks they replace [51]. The evaluation further assumes that the TDP of the chips allows an SAF that can contain up to eight simultaneously active processing cores.

5.3.1 In-order Core

For a sensitivity analysis, Section 6.6.3 presents experimental results where the multicore systems consist of in-order cores, instead of the out-of-order cores described in the previous section. The micro-architectural parameters for this study are given in Table 5.3. The cache and memory hierarchy parameters remain unchanged, and hence omitted from the table.

Fetch/issue/commit	2 instructions / cycle
Integer pipeline	7 stages
I-Window & ROB	32 entries, in-order issue
Store buffer	8 entries
YAGS branch pred.	4k choice, 1k except, 6 tag bits

Table 5.3: Micro-architectural Parameters of In-order Cores

5.4 Energy Estimation

To estimate power and energy consumption, multiple tools are integrated in the simulation infrastructure described in Section 5.2. Processor core activity and power is largely derived using the circuit models from Wattch [20]. The memory hierarchy uses CACTI 4.2 to model dynamic energy dissipation and leakage in the caches [105]. Since a large L2 cache can account for a significant amount of the static energy consumption [77], the evaluation assumes that it is implemented with stacked devices, effectively reducing leakage by 3X, and providing a much more energy efficient design point, more recently appearing in the current implementations [95].

Leakage is modeled as a temperature dependent fraction of the dynamic power [97]. This temperature dependence of leakage is derived from SPICE simulation of an inverter for the 32nm technology node using planar CMOS, combined with the typical leakage contributions in the total power reported in current implementations [34, 94, 95]. Temperature dependence is extracted for every 0.5 C interval between 30C to 115C. Figure 5.1 shows this temperature dependence of the leakage factor used in this evaluation. Leakage rises more rapidly with temperature, as the subthreshold component is exponentially dependent with temperature [118].

5.4.1 Calibrating Wattch

Power modeling tools such as *Wattch* rely on tracking the cycle-by-cycle access history of various functional blocks in a core, and on certain assumptions about circuit design styles. While such tools are considered reasonable at estimating the power/energy consumption *trends*, previous studies indicate their inaccuracy in estimating the *absolute* power [62, 66]. To overcome this drawback, this dissertation uses a combination of other tools to calibrate the energy consumption reported by the power model.

First, the combined tool chain derives the total power consumption (both static and dynamic)

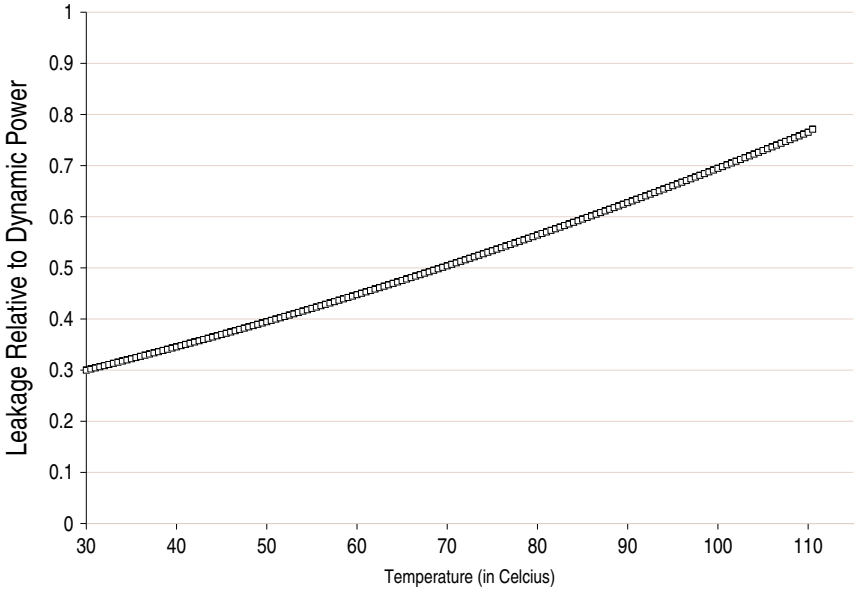


Figure 5.1: Temperature dependence of leakage power expressed as a fraction of the dynamic power.

of the major functional blocks of a core when they are operating close to their TDP using HotSpot [97]. Using the *hotfloorplan* tool (a part of the HotSpot tool set), the floorplan of a single processing core is obtained. The methodology adopted focuses on a single core because: a) accurate floor planning of a chip with eight or more cores is significantly complex and requires numerous assumptions about the topological organization of caches, interconnect, cores, I/O controllers and so forth and (b) the effect of lateral heat diffusion is mostly localized [97], and therefore the temperature of each functional block is primarily dependent on the nearby functional blocks of the same processing core. Because of this second factor, the methodology based on a single core is still able to capture the major effects of modeling temperature in the micro-architecture. In the second step after obtaining the total power, static power component is estimated as a temperature-dependent fraction of dynamic power using Figure 5.1, subsequently estimating the dynamic power component from the total power derived in the previous step. Finally, this dynamic power is used to

calibrate the dynamic energy consumption reported by the Wattch-based power module, when running a high IPC workload at the nominal voltage-frequency. For this evaluation, pmake is used as a proxy for a high IPC workload. A similar methodology was adopted by Li and Martinez [66], and an elaborate study on power model validation through thermal sensing also validated the soundness of this approach [75].

For the timing runs, the infrastructure initializes the temperature of the heat sink, and various functional blocks of the core, by several iterative runs of 500 ms, until there is little variance between two successive runs. During the timing runs, the temperature of various functional blocks is updated every 32K cycles, and the static power component of each block is adjusted based on its temperature by interpolation from the leakage factors shown in the Figure 5.1. Intermittent update of the temperature substantially improves the simulation speed, with negligible impact on preciseness of the micro-architectural impact of temperature [69].

Table 5.4 shows the power consumption of a single core in the baseline multicore across different workloads. The table shows the average, maximum and the minimum power consumed during several 32K cycle intervals. To avoid the startup effects, the initial part of the simulation run is not considered while calculating these values. The average power consumptions in the on-chip cores are substantially higher in pmake and pgoltp due to their high IPCs (shown later in Section 6.1.1, which significantly raise the pipeline activity.

Benchmark	Average (Watts)	Maximum (Watts)	Minimum (Watts)
Apache	5.01	8.54	2.53
OLTP	4.92	9.7	2.66
Zeus	5.12	8.52	2.21
pgoltp	6.75	11.95	3.14
pmake	7.35	12.0	3.25

Table 5.4: Core power consumptions in the baseline multicore

5.5 OPMS Methodology

This section presents the details of methodology adopted for the purpose of OPMS evaluation. Section 5.5.1 discusses the details of core provisioning adopted in the OPMS schemes. Section 5.5.2 presents additional methodological details on leakage estimation in OPMS.

5.5.1 Core-Provisioning

The OPMS employed by CSP use the same core microarchitecture, but the multicore integrates 12 such processing cores, and a reduced L2. For a given workload, CSP partitions the physical CPUs into two sets based on the fraction of time that a particular workload spends in OS or user mode. In this dissertation, the evaluation ignored short-lived register window traps for computation transfer. Other OS computation (e.g., system calls, page faults, interrupt handling) trigger a computation transfer, and are carried out in the cores provisioned for OS computation.

Among the five workloads evaluated in this dissertation, Apache and Zeus spend a majority of their execution in the OS mode (see Table 4.2). Consequently, CSP provisions 8 physical CPUs to execute the OS computation. Other benchmarks spend most of their execution in the user mode, and therefore, 8 PCPUs are provisioned for the user mode. The specific core assignment varies across TAP and SAP, and are described in Table 5.5.

5.5.2 Leakage in an OPMS

The OPMS schemes used in this evaluation maintain several cores in an inactive state. However, each of these cores implement aggressive leakage control techniques when inactive (see Section 3.3.2). Since state transitions happen at a larger timescale (one to two orders of magnitude higher) than the typical *break-even points* reported in the sleep transistor designs [86, 110] and analytic

Assignment Policy	Benchmark	User Computation	OS Computation
TAP	Apache, Zeus	4 cores, each core is responsible for user computations from two VCPUs.	8 cores, each core responsible for the OS computations from one VCPU.
	OLTP, pgoltp, pmake	8 cores, each is core responsible for the user computation from one VCPU.	4 cores, each core responsible for OS computations from two VCPUs.
SAP	Apache	4 cores, each core is responsible user computations from two VCPUs.	8 cores, *read*, *sendfile*, and *s/w interrupts*: 2 cores each, rest to one core.
	OLTP	8 cores, each core is responsible for the user computation from one VCPU.	4 cores, *semsys*: 4 cores, rest to a single core.
	Zeus	4 cores, each core is responsible for user computations from two VCPUs.	8 cores, *write*, *stat*, *readv*: 2 cores, rest to a single core.
	pgoltp	8 cores, each core handles the user computation from one VCPU.	4 cores, *send*: 3 cores, *recv*: 2 cores, rest to a single core.
	pmake	8 cores, each core handles the user computation from one VCPU.	4 cores, *page fault handling*: 3 cores, *read*, *s/w interrupts*: 2 cores, rest to one core.

Table 5.5: Core provisioning in TAP and SAP

models [30, 50], the transition energy is amortized over the duration of the idle period [30]. For the purpose of this dissertation, the experimental results assume the major functional blocks in the core expend 2% of their active leakage while inactive. Furthermore, the core's L1 caches and predictors are put to drowsy mode when the core is inactive, expending 20% of their active leakage energy mode [33].

Chapter 6

Evaluation

The goal of this chapter is to understand the performance characteristics of CSP and OPMS, and quantify their relative strengths and drawbacks compared to the baseline system. This dissertation focuses on one important class of applications, particularly relevant for multicore architectures: multithreaded server class applications. Results are presented for four different applications from this class, while another application (*pmake*), belonging to a different class, is also presented to highlight the distinguishing features of the server class workloads.

Several important aspects of OPMS and CSP require detailed quantitative study and validation. First, CSP aims to improve the properties based on the instruction stream, but the extent of this improvement plays a key role in its impact on the workload execution time. The specific application of CSP that this dissertation targets is unlikely to hamper the data reference locality, based on the preliminary characterization presented in Chapter 4. However, this chapter takes a more in depth look at data cache performance, quantifying the impact of both load and store operations.

Unlike the expected improvement in memory reference locality, CSP introduces new overhead from frequent computation transfers (CT) that are necessary to realize CSP. The runtime overhead from CT conspires to obviate the benefits from the reduction in memory stalls. However, the

possibility of superior communication between on-chip processing cores in a multicore system can dramatically reduce this overhead. This chapter presents quantitative data on the overhead from CT, including the average latency overhead from a single CT, collective overhead of CT in the runtime, and the aggregate interconnect bandwidth required for communicating register state between on-chip cores.

After discussing the overhead from computation transfer, overall runtime performance is presented. The performance impact of CSP is an outcome of dynamically specializing core microarchitectures that exploit properties of instruction stream. In addition to the overhead from CT, the assignment policies evaluated in this dissertation can cause contention on core resources (see Section 4.5.2). On the other hand, the OPMS design paradigm offers flexibility in assigning computation, facilitating schemes like CSP. This chapter quantifies these design trade-offs, and illustrates their impact on overall execution.

Since achieving energy efficiency is a major goal of the OPMS design paradigm, this chapter shows the results from a detailed study on the energy efficiency characteristics of an OPMS, compared to the baseline system. Although an OPMS provisions more cores than the baseline system, it restricts the core activity at or lower than the baseline multicore. However, the leakage power from inactive cores may increase the overall power consumption in an OPMS. This chapter quantifies this leakage overhead and compares core utilization, overall power consumption, thermal characteristics, and energy efficiency of the system using the energy-delay product metric.

Finally, this chapter presents several sensitivity analyses to validate and understand the results in different contexts. While demonstrating the robustness of the quantitative results, this analysis attempts to identify the critical sources of benefit or drawback of an OPMS as compared to the baseline. For example, the OPMS evaluated in this dissertation employs a different microarchitecture than the baseline, and use CSP that alters the inherent memory reference patterns

exercised on this micro-architecture. In the first part of this analysis, this chapter separately inspects the individual effects of the specific OPMS micro-architecture employed and CSP, respectively. Comparing results from their individual effects on the results, allows to ascertain the relative contribution from the OPMS design paradigm, CSP, and the particular micro-architecture OPMS employs. In the second part of this analysis, results using different processor core models (i.e., in-order), and different cache sizes are presented.

The chapter organization is summarized as follows. Section 6.1 presents the workload characterization with a detailed look in their memory stalls. Section 6.2 demonstrates the locality impact of CSP on various caches and branch predictors in the multicore system. Section 6.3 presents the overhead from Computation Transfer in detail. Overall performance, and the impact of various design trade-offs is presented next in Section 6.4. Section 6.5 illustrates the power and energy efficiency characteristics of the OPMS. An elaborate sensitivity analysis is presented in Section 6.6, followed by a summary related work for the OPMS design in Section 6.7. Finally, the chapter concludes with a summary in Section 6.8.

6.1 Workload Characterization

This section presents the performance characteristics of the multi-threaded workloads on the baseline system. The characterization will be crucial to understand various other performance results presented later in the chapter.

6.1.1 IPC

Figure 6.1 presents the IPCs of various workloads this dissertation examines. For each workload, the figures shows three bars representing the overall IPC, the user code IPC, and the OS code

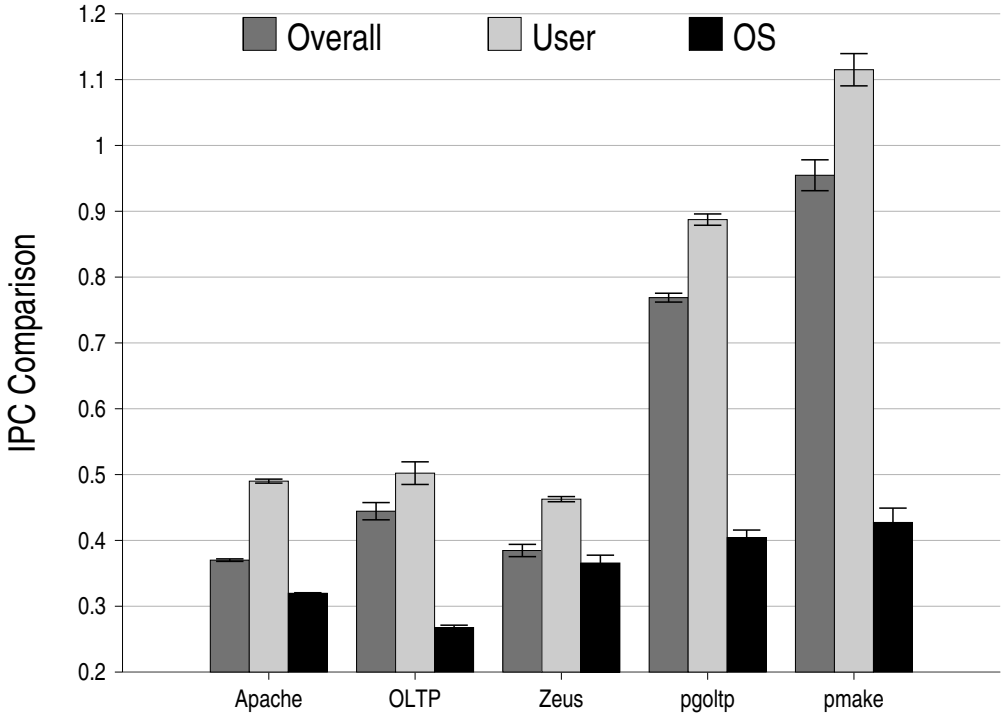


Figure 6.1: IPC Characteristics for the evaluated workloads in the baseline multicore. For each benchmark, the three bars represent the overall IPC (left), the user code IPC (middle), and the OS code IPC (right), respectively.

IPC, respectively. The errors bars show the 95% confidence interval in these data set. Across different workloads, the figure shows a substantial variation in the overall IPC. *pmake* achieves the maximum IPC of 0.95, while Apache achieves the minimum IPC of 0.37 among these workloads.

The figure also shows a significantly lower OS code IPC (i.e., IPC observed during the execution of OS code), compared to the user code IPC. For example, *pmake* achieves an IPC of 1.16 when executing its user code, while the OS code can only achieve 0.42. Many factors combine to restrict the IPC achieved in OS code: serialization instructions [116], large memory stalls [3], and user code interference [2], among others. The user code also sees a significant variation among different workloads due to the inherent disparity in their computation, as well as the extent of OS

execution impact (such as cache and predictor interference) on it. The variation in OS code IPC is much smaller among these workloads: 0.32 in Apache to 0.42 in pmake. This OS code IPC variation is due to the difference in OS services sought by these workloads, despite running on the same operating systems software.

For Apache and Zeus, the overall IPC is largely determined by the IPC in OS code, as major part of their executions are composed of OS code (see Table 4.2). Similarly, pgoltp and pmake demonstrate much higher overall IPC due to the IPC contribution from their respective user code, where they spend most of their runtime. OLTP also spends most of its execution in user code, but achieves significantly lower overall IPC, as the user IPC is fairly low in this workload.

The low IPC in these server class workloads is due to significant memory stall times, which results from poor cache performance in these workloads [3]. Server class workloads have substantially lower temporal and spatial locality in their memory references leading to the poor performances of caches. The next section presents a quantitative analysis of their memory latency profile.

6.1.2 Memory Latency

Figure 6.2 illustrates the average memory latency observed in various workloads evaluated in this dissertation. Three bars for each benchmark, from left to right, present the overall average memory latency from all references, average latency from data references, and the average latency from instruction references, respectively. Except pmake, all workloads exhibit substantially large delay in their average memory request times. Zeus and pgoltp show maximum delay of 11.3 and 10.9 cycles, respectively.

The average data reference latency is larger than the average instruction reference latency in all the workloads. However, for Apache and OLTP, the instruction reference latency is much closer

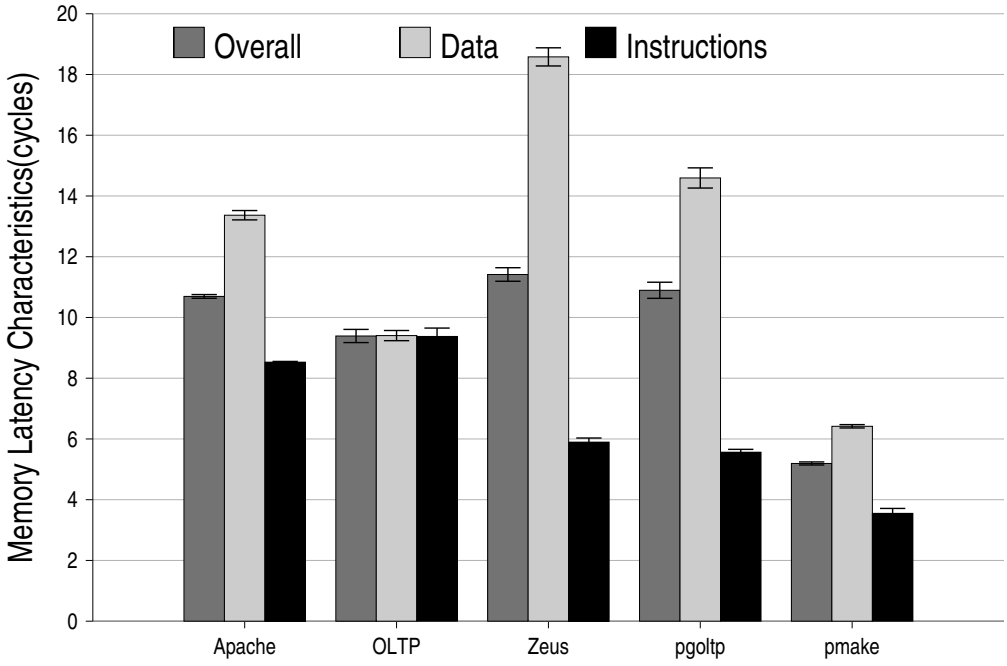


Figure 6.2: Average memory latency for the evaluated workloads in the baseline multicore. From left to right, the three bars for each benchmark present overall average memory reference latency, average data reference latency, and the average instruction reference latency, respectively.

to the data reference latency. In contrast, the instruction latency in other three applications is substantially lower—a behavior typically seen in many other applications (e.g., a vast majority of the single-threaded SPEC CPU2006 benchmarks [44]).

This large memory latency results in the low IPC seen in these workloads before (see Figure 6.1). However, the comparative results of pgoltp and OLTP shows an apparent anomaly. OLTP exhibits lower average memory latency than pgoltp, but the IPC of former is *less* than the latter. This phenomenon can be explained by the comparing the instruction and data reference latencies in these workloads. Although the overall memory latency in OLTP is lower, the instruction reference latency is much higher in OLTP than pgoltp: 9.37 in OLPT, while only 5.56 in pgoltp. The out-of-order execution cores modeled in this dissertation can overlap data latencies much better than

instruction latencies [3, 42]. As a result, an increase in instruction latency more severely impedes overall performance metrics such as IPC, compared to a similar increase in data latency.

Next subsection presents the performance of the cache hierarchy, which explains the source of the memory latency illustrated in Figure 6.2.

6.1.3 Cache Miss Rates

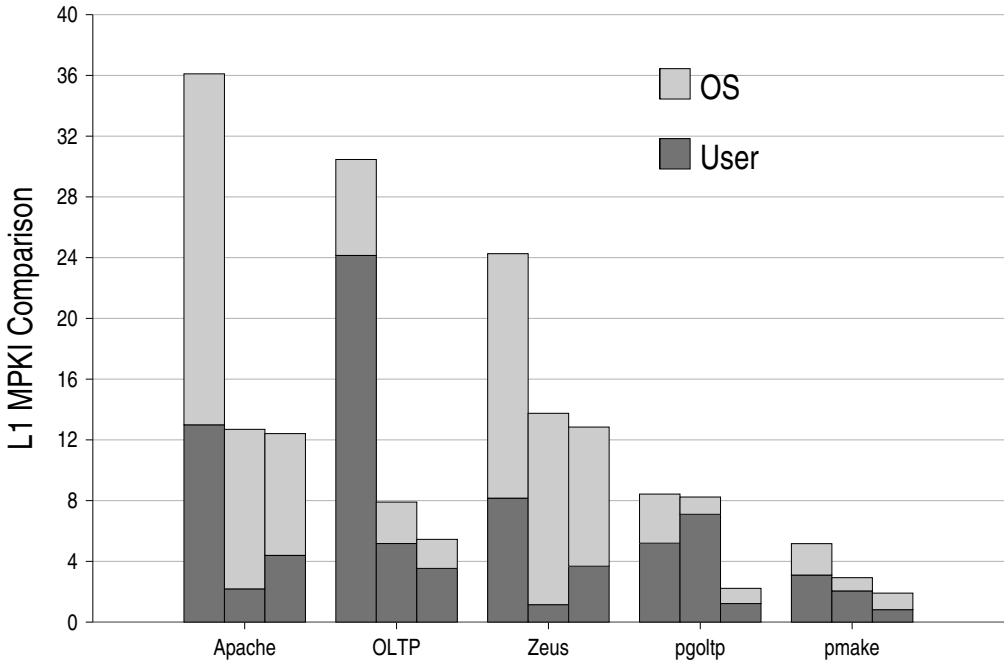


Figure 6.3: L1 misses presented as misses per kilo instructions (MPKI). From left to right, the three bars represent instruction fetch miss, data cache miss from load operations, and data cache miss from store operation, respectively. Each bar is also broken into two parts showing the miss component from the user code and the OS code.

Figure 6.3 presents the L1 cache miss components from user and OS code in these workloads. The figure shows the number for cache misses for every thousand committed instructions or miss per kilo instructions (MPKI). Three bars for each benchmark show the instruction cache miss rate,

the data cache miss rate from load instructions, and the data cache miss rate from store instructions, respectively (left to right). Each bar also shows the miss components from OS code and user code, where a cache miss generated while executing user or OS code is regarded as user miss component and OS miss component, respectively.

Apache, OLTP, and Zeus exhibit large miss rates at the L1 caches. For all these three benchmarks, the instruction MPKI is larger than either of the data cache miss rates from load or store instructions. In fact, the instruction MPKI of Apache and OLTP is larger than the combined miss rates from load and store operations in the data cache. For example, Apache shows an aggregate instruction MPKI of 36.1, while the data cache miss rates for loads and stores are 12.7 and 12.4, respectively. For the other two benchmarks, the miss rates are substantially lower. For example, pmake has an instruction MPKI of 5.2, while the total data cache MPKI is 4.8. Although, it also has higher instruction MPKI than the data cache MPKI, at these low absolute values of miss rates, none of these memory references impose significant performance bottlenecks for pmake.

For each of the miss rates reported, the graph also shows the miss component from OS and user code. For Apache and Zeus, the OS miss component is substantially higher. This is an expected result, as most of the execution in these workloads are performed in the OS mode. Likewise, the miss component from the user code is more dominant in OLTP, pgoltp and pmake.

Despite comparable instruction miss latencies reported for Zeus and pgoltp in Figure 6.2, the instruction MPKI is substantially higher in Zeus than pgoltp in this figure. Compared to pgoltp, Zeus generates significantly larger number of fetch accesses into the instruction caches due to more instruction accesses across cache line boundaries. Accesses across cache line boundaries cannot be satisfied by the instruction buffer modeled in this dissertation, and thus appear as fetch accesses. Subsequently, Zeus observes many more fetch misses, even while executing for an identical instruction count, resulting in a larger MPKI. Unlike MPKI, the local cache miss rate (i.e.,

misses/accesses in each cache) is quite similar in these two workloads.

Finally, Figure 6.4 presents the miss rates observed at the shared L2 cache. Similar to the previous figure, three bars represent misses from instruction accesses, load accesses and store accesses, respectively. Instead of separating OS and user miss components, each bar shows the classification based on where a particular miss sources the data from: main memory or another L1 cache (coherence). The L2 cache see negligible instruction misses, as its large capacity is able to retain bulk of the dynamic code footprint. Zeus exhibits the worst data miss characteristics, showing MPKIs of 7.4 and 9.6, from load and store operations, respectively. Relatively larger L2 miss rates in Zeus cause the increased data reference latency seen in Figure 6.2.

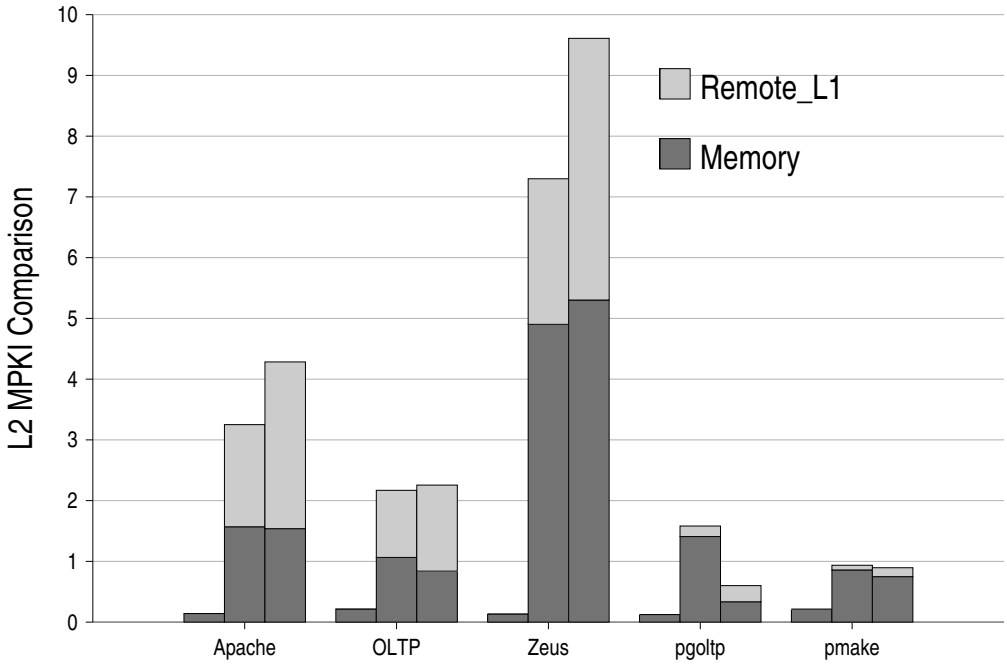


Figure 6.4: L2 misses presented as misses per kilo instructions (MPKI). From left to right, the three bars represent instruction fetch misses, data cache misses from load operations, and data cache misses from store operations, respectively. Each bar is also broken into two parts showing the misses that source data from the main memory and remote L1, respectively.

6.2 Locality Impact

By improving the locality of references, CSP aims to achieve higher efficiency in exploiting instruction stream properties, boosting the performance of caches and branch predictors. This section presents quantitative data on cache and branch predictor performance of an OPMS that employs CSP schemes. A major part of this section is devoted to illustrate the impact of memory references, and its impact on cache performances for both instruction and data. The last part will present results on the impact on branch predictors.

6.2.1 Instruction Locality

Figure 6.5 shows the LI Instruction cache misses due to instruction references, normalized to the baseline. The three sets of bars for each benchmark represent the baseline, TAP, and SAP respectively. Each bar is broken down into two components showing the contribution from the user code and the OS code, respectively.

L1 instruction misses are reduced for every benchmark using TAP, as it spreads the user and OS references that are completely independent across different cores. The locality of instruction references on each core is improved, allowing the corresponding instruction caches to retain predictive state for a longer duration, and satisfy accesses from the cores. This reduction is greater than 34% for every benchmark except OLTP, in both the CSP schemes.

Using TAP, the two web servers perform particularly well, achieving a reduction of 38% and 44% in Apache and Zeus, respectively. The relative reduction achieved in the miss component from user code is substantially more: Apache achieves nearly $2X$ reduction, while Zeus achieves more than $3X$ reduction. The long execution phases in the OS code, typically seen in these two workloads, evicts a majority of the user code fragments in the caches. Subsequently, when the execution returns to user code, a large number of cache misses follow. Thus, separating the relatively

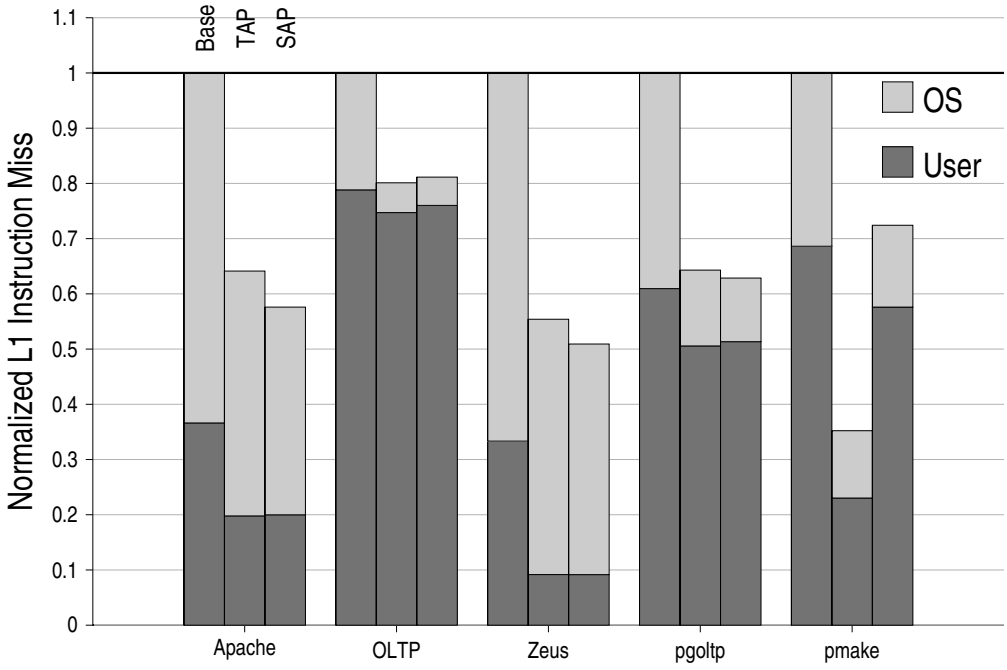


Figure 6.5: LI Instruction Miss Comparison. The three bars for each benchmark represent results for the baseline system, TAP and SAP, respectively (left to right). Each bar is further broken down to show the miss components from the user code and OS code, respectively.

smaller user code footprint from the OS code dramatically reduces misses from them.

In contrast, other three benchmarks see significantly more benefit in their OS miss components. For example, OLTP sees a reduction of over $4X$. However, the overall reduction is only 19% as the user code miss component, which comprise nearly 80% of all the misses, shows only a marginal reduction. The OS miss reduction is nearly $3X$ in pgoltp, but the overall miss reduction is also significant (34%). This result is due to a larger miss component of OS code in pgoltp (40%), as well as more relative reduction from the user misses.

SAP achieves further reduction in instruction misses for Apache and Zeus, although the extent of additional reduction over TAP is marginal: 4% and 3%, respectively. This lack of further improvement indicates that the distribution of computation based on system calls is not as effective

as expected. A more detail discussion on this topic is presented later in Section 6.4.3.

The benchmark pmake shows an unexpected reduction of the user miss component in TAP compared to SAP. Since both these CSP schemes assign user computation from the VCPUs in an identical manner, this reduction is unexpected. Unlike other benchmarks, pmake does not exhibit a steady phase, and the results can vary substantially during its execution depending upon the exact files it chose to compile (see Section 5.1 for the detailed description of this benchmark). The difference in specific files compiled during the simulation runs causes substantial alteration in the code path exercised, changing the reference locality, both for instructions and data. One indication of this behavior can be derived from the number of page faults and TLB shootdowns seen during these runs. While SAP and the baseline exhibit fairly comparable statistics for these events, all the simulation runs for TAP show substantially different statistics for them. Even after running more than hundred trials, and simulating for $10X$ longer compared to the other workloads, this difference persists, causing the discrepancy in results seen here.

A vast majority of the L1 instruction cache misses is serviced from the shared L2, as demonstrated by the small L2 instruction miss rates in Figure 6.4. This data is omitted due to its lack of importance in the entire discussion. Instead, this section presents the impact of reducing instruction reference latency on the performance of the processor core front end, which is responsible for maintaining a steady flow of instructions. First, a breakdown of the front end stalls is presented to show the role of instruction misses. Then, the total number of front end stalls are compared.

Figure 6.6 shows the comparison of front end stalls across different multicore configurations. A front end can stall fetching instructions due to a variety of reasons: instruction cache miss, lack of available space in the instruction window, serializing instructions (SYNC), and others. The other category collectively includes various events such as Load Queue full, Store Queue full, pipeline exceptions from various traps (e.g., register window trap). The aggregate contribution of the Other

category is small, and hence no further separation is shown. In this figure, each bar is normalized to itself, so as to indicate the relative contributions from the front end bottlenecks.

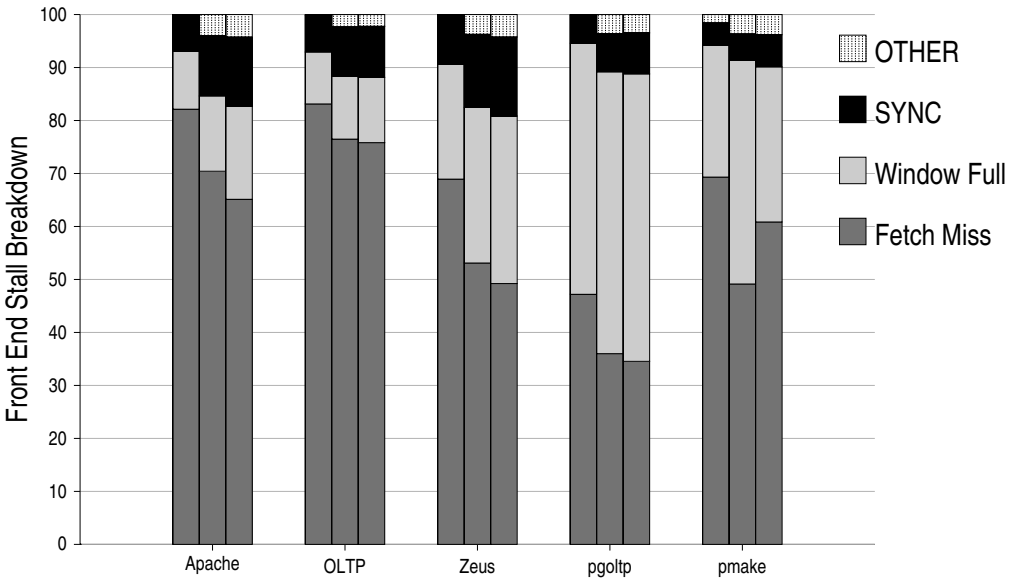


Figure 6.6: Front end stall breakdown comparison. The three bars for each benchmark represent results for the baseline system, TAP and SAP, respectively (left to right). Each bar shows the relative contribution of several events that lead to stall in the front end. From the bottom, these are: miss in the L1 Instruction Cache (Fetch Miss), instruction window full, serializing instructions (SYNC), and other events.

The bars for the baseline demonstrates that instruction misses are the most dominant bottleneck in the front end for all these workloads except pgoltp, contributing more than 60% of the front stalls. For Apache and OLTP, instruction misses contribute 82% of the front end stalls, severely limiting the IPC achieved in them despite aggressive out-of-order execution cores. For pgoltp, the lack of space in the instruction window is the most dominant reason for front end stalls. This high occupancy of instruction window in pgoltp stems from the large data stall component in pgoltp (see Figure 6.2), where dependent instructions behind a long latency load fills up the instruction window.

In TAP, the profile for front end bottleneck shows substantially lower contribution from instruction misses, and higher relative contribution from the rest. For example, in Zeus the relative contribution from instruction misses drops from 69% in the baseline to 52% in TAP, while the contribution of instruction window increases from 21% to 29%. SAP lowers the instruction miss bottleneck further, with a corresponding increase in the contribution from the rest. Except pmake, the aggregate contribution from the other categories (top most section in each bar) is negligible in the baseline because of the large instruction miss component. After substantial reduction in front stalls from instruction misses in TAP and SAP, the other category becomes visible, accounting for up to 4% of the front end stalls.

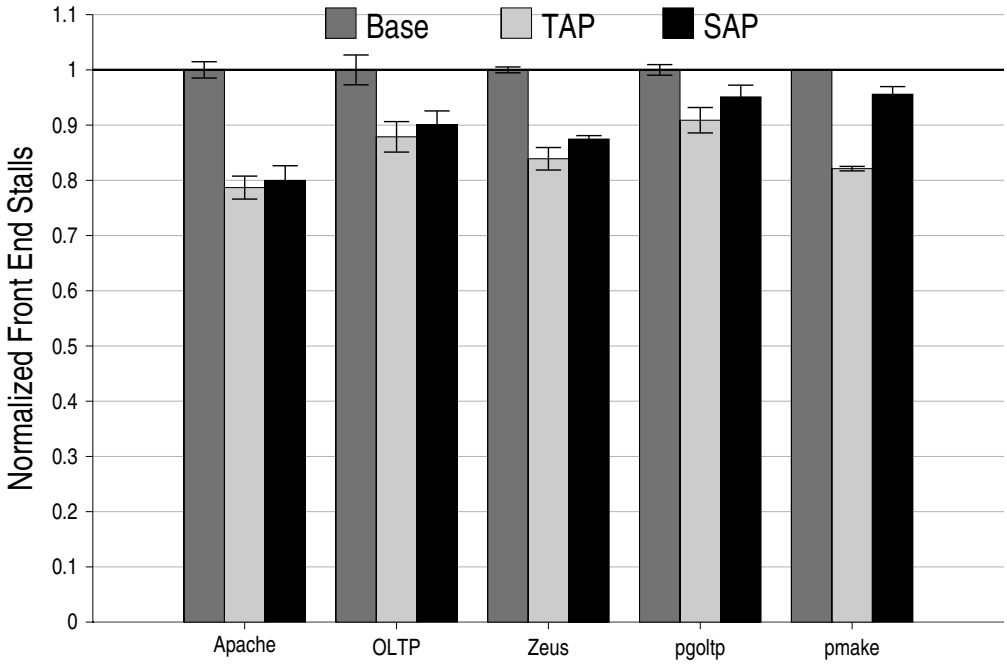


Figure 6.7: Front end stall events comparison. The three bars for each benchmark represent results for the baseline system, TAP and SAP, respectively (left to right).

Figure 6.7 presents the comparison of the number of front end stalls seen across the baseline

system, TAP, and SAP, respectively. Results are presented normalized to the baseline system. Both TAP and SAP shows a substantial reduction in the total number of front end stall events, demonstrating a superior instruction delivery at the front end. For Apache, TAP and SAP achieves 21% and 20% reduction, respectively. In pgoltp and pmake, much smaller reduction is seen as this benchmark does not exhibit large instruction stalls. Note that although TAP achieves larger reduction in pmake, this result is primarily due to the variability in this particular workload, as explained above (see the discussion on Figure 6.5).

6.2.2 Data Locality

In contrast to the L1 instruction cache, which sees only one type of accesses, L1 data cache services two different types of accesses due to load and store instructions, respectively. References from these instructions trigger different mechanisms in the coherence protocol implemented by the general purpose multicore systems. Servicing load operations requires a valid copy of the data, while the store operation can only be serviced after acquiring exclusive permission to modify the data location. The cost associated in these operations can vary depending on the precise coherence mechanism triggered. This subsection will present several aspects of data locality for an in-depth understanding of CSP in multicore designs.

6.2.2.1 Load

Figure 6.8 presents the comparison of L1 data cache misses from load operations. For each benchmark, the figure shows the result for the baseline system (Base), and the two CSP schemes TAP and SAP, respectively. Each bar in the figure also shows the miss components from user and OS, respectively.

TAP achieves lower data cache misses from load operations for two of the benchmarks evalu-

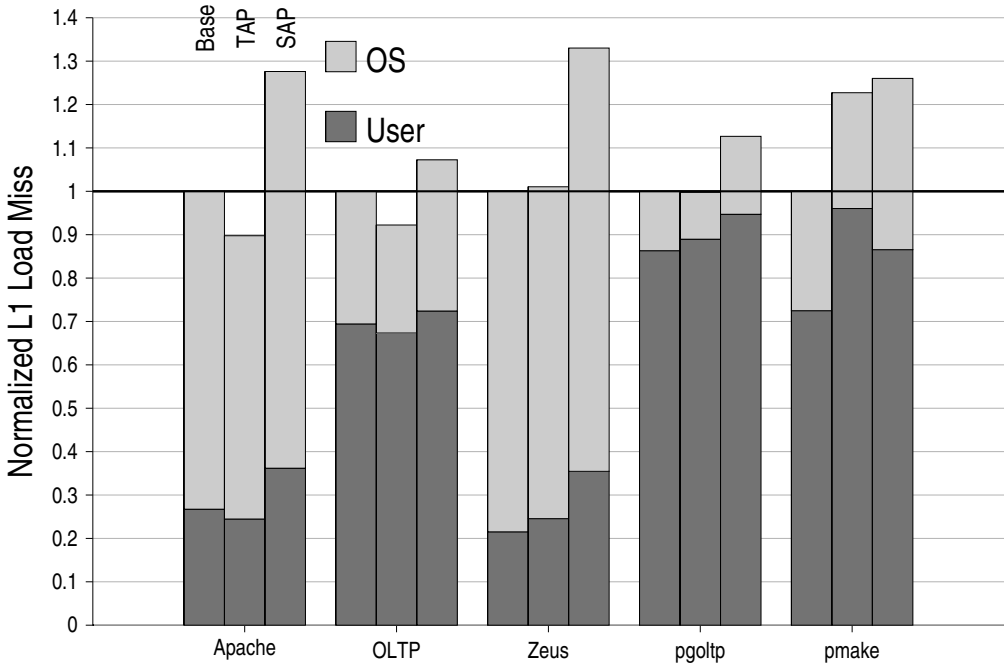


Figure 6.8: Normalized L1 Load Miss. From left to right, three bars for each benchmark show the results for the baseline, TAP and SAP, respectively. Each bar is broken down to show the contribution from User Code (bottom) and OS code (top), respectively.

ated in this dissertation. In Apache, it achieves a reduction of 12%, while the reduction is smaller in OLTP (6%). In pgoltp, the L1 load misses are comparable. In Apache, this improvement is a result of reducing both the user misses and OS misses. The relative reduction in user misses is more in Apache, unlike the general pattern seen for L1 instruction misses in Figure 6.5. However, TAP performs poorly in pmake, resulting in a 22% increase in the data cache misses from loads. Because pmake has lower misses in the baseline, the additional misses caused from OS to user communication in TAP, leads to a larger relative increase in overall misses. Apart from pmake, the load misses remain comparable to the baseline system despite frequent computation transfers in TAP, validating the preliminary results presented to motivate the separation of user and OS code fragments in the previous chapter (see Section 4.4).

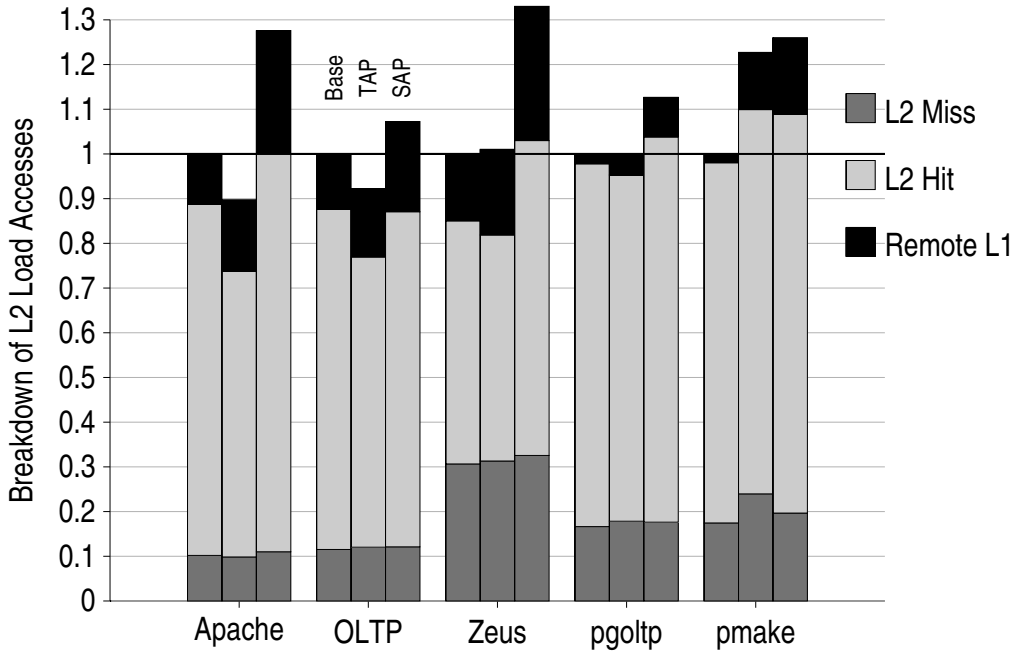


Figure 6.9: Breakdown of L2 Load Accesses due to L1 misses. From left to right, three bars for each benchmark show the results for the baseline, TAP and SAP, respectively. Each bar is broken down to show the individual fractions of L2 accesses that cause an L2 miss (bottom), L2 hit (middle), and remote L1 access (top).

On the other hand, SAP shows a degradation of data cache performances for loads across all benchmarks. Apache and Zeus see maximum degradation, observing an overall 28% and 32% rise in data cache misses, respectively. Both the user component and the OS component observe increases, but the increase is relatively more in the OS component. By provisioning a given system call on a particular core, different system calls from a single VCPU are assigned to different cores. This assignment negatively hurts the data locality, as the kernel often use the same section of the data memory for a given software thread that is scheduled on a VCPU, even while executing different system calls [74].

To understand the coherence mechanisms triggered by the L1 data cache misses due to loads,

Figure 6.9 presents a breakdown of L2 accesses due to L1 load misses. Each bar is broken down into three parts to show the fraction of L2 accesses that results in a L2 miss (bottom) and subsequent access to the main memory, L2 hit (middle), and remote L1 access (top), respectively. Among these three categories, main memory misses are most costly, followed by remote L1 accesses, and L2 hits. For each benchmark, the figure shows this result for the baseline (left), TAP (middle), and SAP (right), respectively.

Despite a smaller L2 size (12MB compared to the 16MB in the baseline), TAP shows only marginal increase in L2 misses. The slight reduction of L2 misses in Apache is counter intuitive. Because TAP accesses cache blocks differently (e.g., more L2 access is seen for the data locations shared by user code and OS code), it alters the LRU information maintained within each set. This alteration in LRU occasionally results in a difference in L2 misses. However, as the figure demonstrates, this marginal reduction in main memory access is not an expected pattern across different workloads.

In all cases, TAP also shows a reduction in L2 hits, accompanied by a corresponding increase in remote L1 accesses. Remote L1 accesses results in cache-to-cache transfers, and these events are triggered when one core attempts to read a data location that has been modified by another core. Due to an extra hop through the interconnect, misses in this category are more costly (both in terms of latency and interconnect bandwidth) than the L2 hits.

For SAP, every benchmark sees an increase in remote L1 accesses. The relative increase in remote L1 accesses is nearly $9X$ in pmake, which shows little sharing of data among on-chip cores in the baseline. In Apache and Zeus, this increases are more than $2X$ and $3X$, respectively. Similarly, there is also an increase in L2 hits, but like TAP, SAP does not show any significant rise in L2 misses that results in the main memory access, despite a smaller L2 cache. The rise in both L2 hit and remote L1 access is explained by the loss of data locality due to spreading systems calls

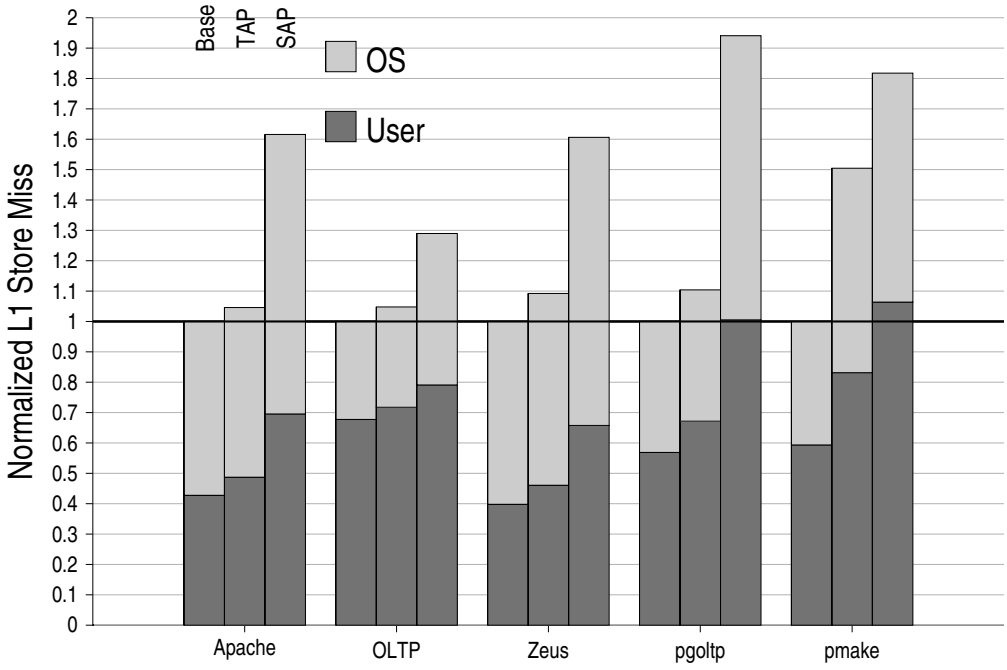


Figure 6.10: Normalized L1 Store Miss. From left to right, three bars for each benchmark show the results for the baseline, TAP and SAP, respectively. Each bar is broken down to show the contribution from user Code (bottom) and OS code (top), respectively.

from the same VCPU across different cores, as explained in Section 6.2.1.

6.2.2.2 Store

Figure 6.10 shows the comparison of data cache misses due to store operations. For each benchmark, the figure shows the results for the baseline (left), TAP (middle), and SAP (right), respectively. Each bar is further broken down to show the contribution from the user code (bottom) and OS code (top).

Unlike data cache misses from loads, TAP shows an increase in data cache misses from stores for all the benchmarks. This increase ranges from 4% in Apache to 50% in pmake. SAP exhibits even worse performance for these stores, showing an increase from 29% in OLTP to 92% in pgoltp.

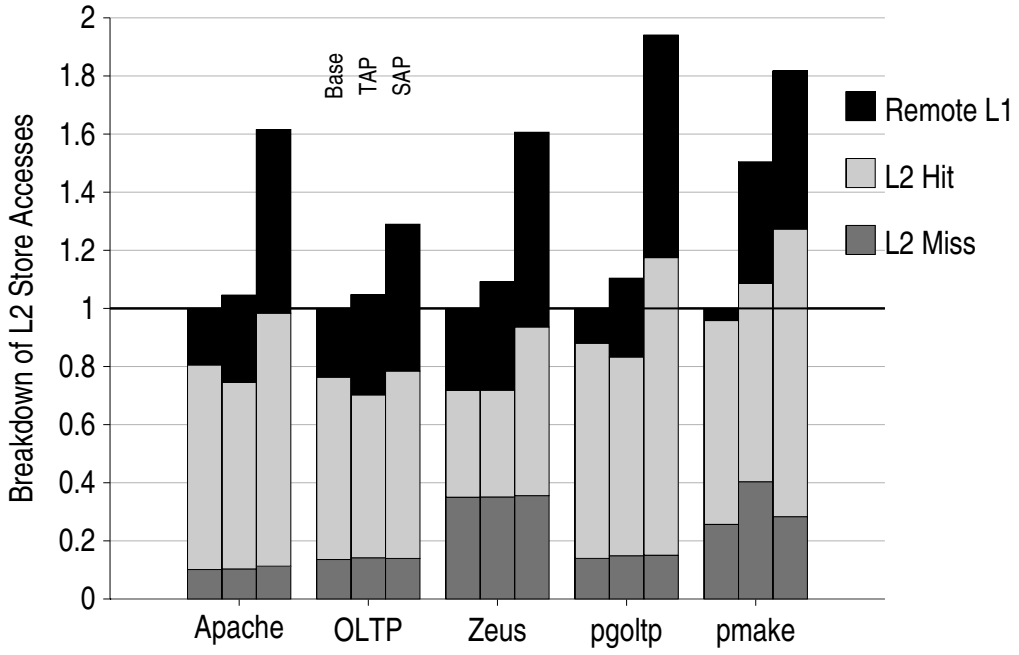


Figure 6.11: Breakdown of L2 Store Accesses due to L1 misses. From left to right, three bars for each benchmark show the results for the baseline, TAP and SAP, respectively. Each bar is broken down to show the individual fractions of L2 accesses that cause an L2 miss (bottom), L2 hit (middle), and remote L1 access (top).

To understand the source of large increase in L1 data cache misses from stores, Figure 6.11 shows a similar breakdown for stores, as shown previously for loads in Figure 6.9. Again, similar to load accesses into L2 from L1 data cache misses, a large increase is seen in store cache to cache transfers from accessing a remote L1.

To analyze this increase in store cache-to-cache transfers, these misses are classified into three categories in Figure 6.12: *false communication*, *explicit communication* between OS and user, and *other* cache-to-cache transfers. The total height of the bars in Figure 6.12 is the same as the upper portion of the bars in Figure 6.11 (labeled *Remote L1*). The figure classifies an explicit communication between the OS and user code when the OS reads or writes user data structures with

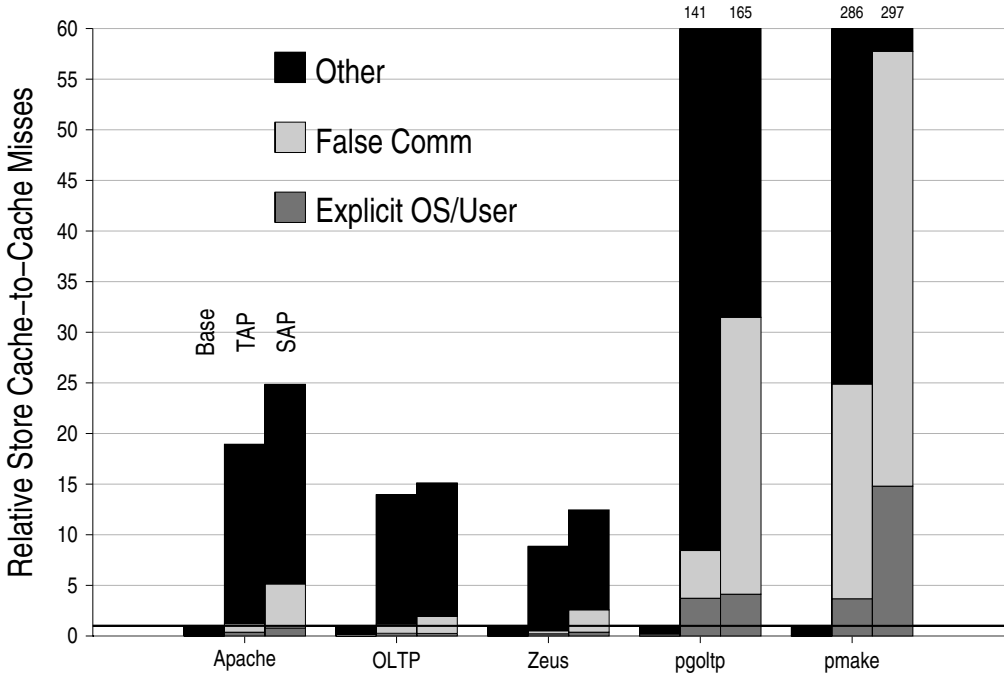


Figure 6.12: L2 Store Cache-to-Cache Miss Breakdown. Three bars for each benchmark show the results for the baseline (left), TAP (middle), and SAP (right), respectively. Each bar is further broken down to show different categories of these transfers. From the bottom there are explicit OS/user communication, false communication, and other, respectively.

Sparc V9's AS_USER ASIs. As expected, explicit user/OS communication results in negligible misses for the baseline, but even using TAP and SAP this explicit communication is a small fraction of store cache-to-cache transfers.

For TAP, more than 80% of the *false communication* is from the OS making a function call, saving a register window, and incurring a spill trap that saves a *user* register set. Since this spill accesses the user's stack space, which likely resides in a user core's cache, it results in a store cache-to-cache transfer (and later, a load cache-to-cache transfer when the user code incurs a fill trap for this window). Thus, the spill results in needless back-and-forth movements of register window state (since computation transfer already moves the register windows to the OS core)

leading to additional overheads. The additional false communication for SAP compared to TAP arises from accesses to the OS execution stack (including spill traps) that use the same address region for a given thread regardless of which system call is executing (no data is actually shared on the stack among different system calls).

The *Other* category represents communication that cannot be categorized more specifically. Much of it belongs to the inherent program communication, which is local in the canonical model, but not in TAP and SAP. Though both pgoltp and pmake show a large relative increase in store cache-to-cache transfers, the baseline misses in this category for these workloads are much smaller than for the other workloads. Thus, the absolute increase in store cache-to-cache misses, and its relative impact on average memory stall times, are quite small.

Although the store misses see a large increase in CSP, their impact on performance is lesser than that of loads in an out-of-order execution with a store buffer. A store buffer allows store instructions to retire before obtaining coherence permission, while also bypassing the new value to the dependent loads. Consequently, much of the store latency does not directly impact the pipeline throughput.

6.2.3 Impact on Average Memory Latency

The reduction in misses, especially for instructions, leads to a reduction in average memory latency. Figure 6.13 shows the improvement in average memory latency achieved through the CSP schemes. TAP improves the average memory latency in Apache, OLTP and Zeus, shows similar latency in pgoltp, and degrades the average latency in pmake. Apache and OLTP see an improvement of 12% and 11%, respectively. Both these benchmarks see superior cache performances for instructions and loads, while marginally degrading stores, resulting in an overall improvement in average memory latency. On the other hand, large increase in high latency cache-to-cache transfers

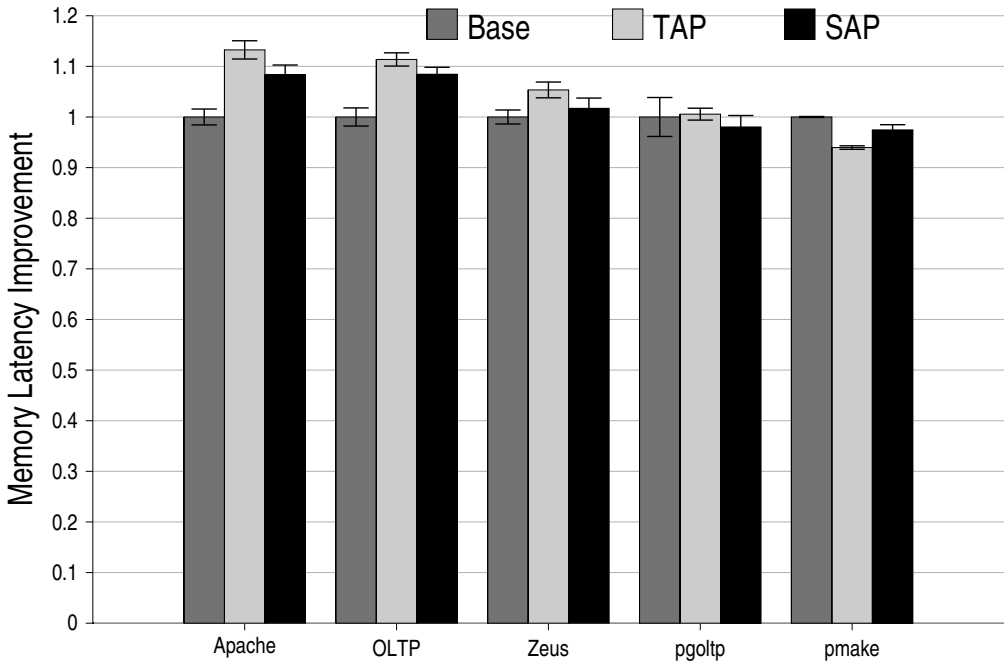


Figure 6.13: Improvement in Average Memory Latency. From left to right, the three bars for each benchmark present the baseline system, TAP, and SAP, respectively.

in pgoltp and pmake, results in a worse average memory latency in these benchmarks.

SAP also observes improvement in the average memory latency for Apache and OLTP, both by 9%. The instruction reference latency in these benchmarks is comparable to that of the data reference latency (Figure 6.3). Therefore, the large reduction in instruction misses leads to an improvement in the average memory latency. In other benchmarks, SAP performs poorly for data cache references, along with a large rise in cache-to-cache transfers. Collectively, these trends negatively impact the average memory references. Despite a large increase in store misses, the average memory latency in pgoltp sees a marginal degradation of 3%. This result is due to the much lower baseline store misses in this benchmark, compared to the loads (Figure 6.3), diminishing the impact of cache performance degradation for stores.

To understand the relative contributions from instruction and data latencies, Figure 6.14 and Figure 6.15 show the latency improvements from instruction and data references, respectively. Both TAP and SAP show dramatic improvements in the instruction latency across several benchmarks, as expected from the large reduction of instruction misses. For Apache, TAP and SAP demonstrate improvements of 39% and 58%, respectively. In contrast, data reference latency does not show any improvement in TAP and SAP across all the benchmarks (except marginal improvement in TAP for OLTP). Although TAP reduces the load misses in Apache and OLTP, the rise in higher latency cache-to-cache transfers for both load and stores, offset any latency improvements for data references. However, for TAP, the data latency degradation is less than 5% in all cases, except pmake. SAP shows much higher degradation, up to 14% in Apache.

6.2.4 Branch Predictors

Apart from caches, another key structure in the processor core that exploits properties of the instruction stream is the branch predictor. Branch predictors predict the outcome of a branch by discerning specific patterns in the previous outcomes. By spreading dissimilar computations across multiple CMP cores, CSP avoids destructive interference of multiple instruction streams (OS and user code) in the branch predictor. Consequently, the branch predictors in the OPMS cores employing CSP, repeatedly operate on a smaller range of instructions, naturally improving their performance.

Figure 6.16 shows the baseline branch mis-prediction rates (percent) from the YAGS predictor used in this dissertation. Three bars for each benchmark show the results for the overall miss rate (left), the miss rate from the user code (middle), and the miss rate from the OS code (right), respectively.

Similar to the variation in IPC seen across these workloads, the figure demonstrates a substan-

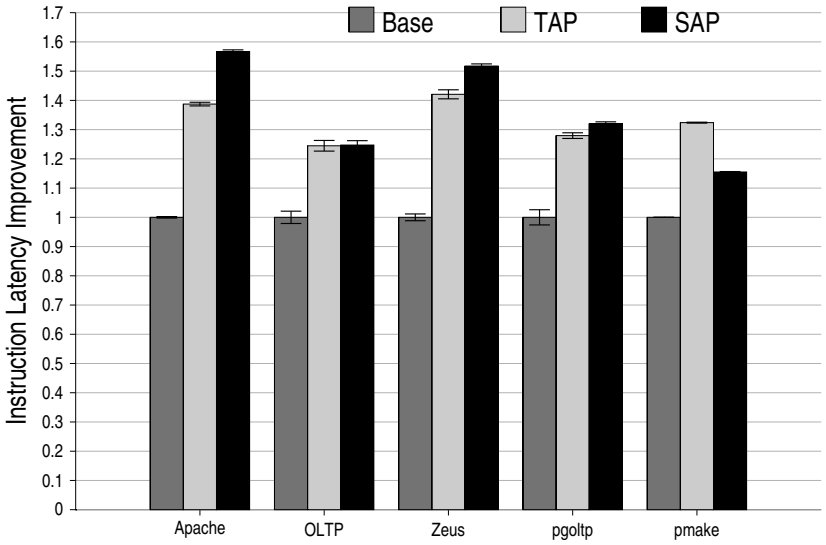


Figure 6.14: Improvement in instruction reference latency. From left to right, the three bars for each benchmark present the baseline system, TAP, and SAP, respectively.

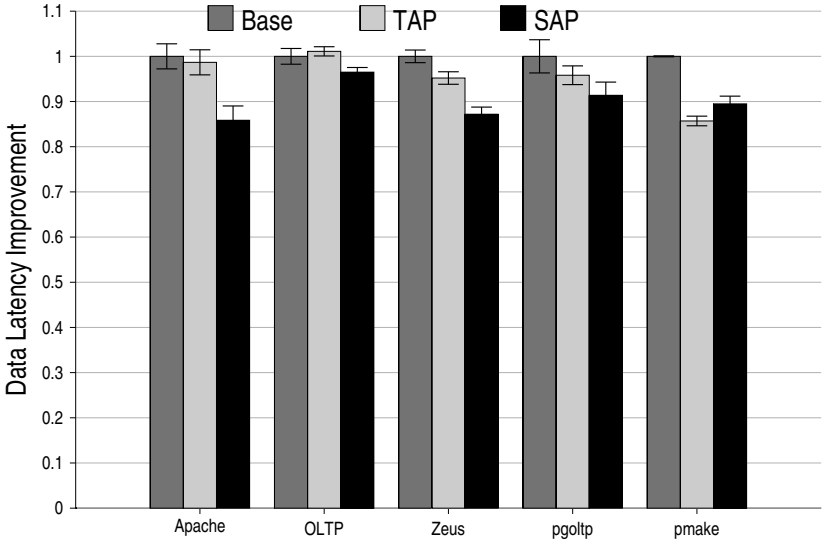


Figure 6.15: Improvement in data reference latency. From left to right, the three bars for each benchmark present the baseline system, TAP, and SAP, respectively.

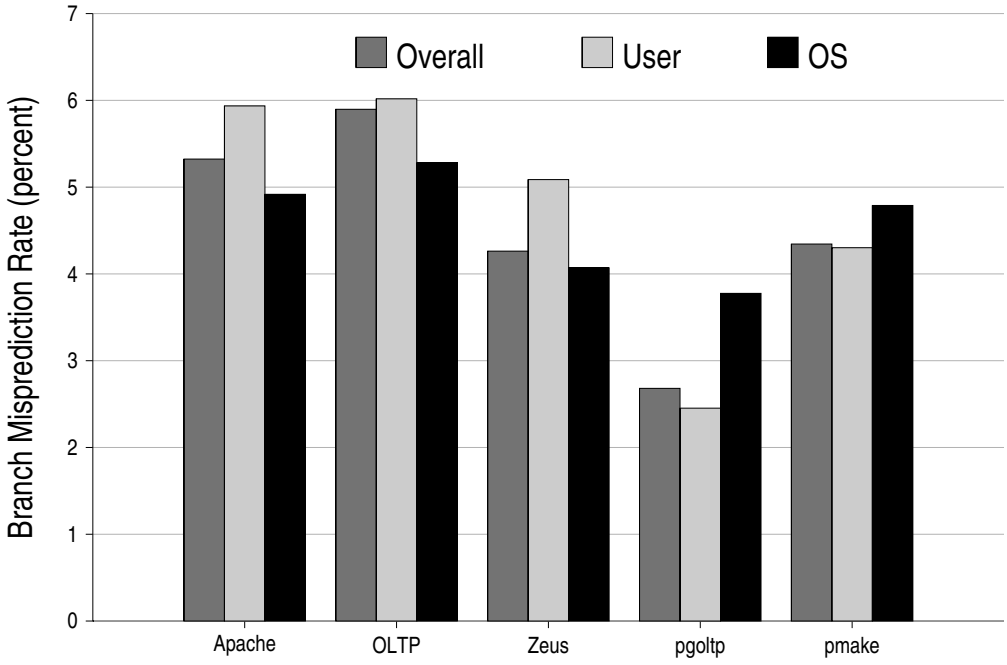


Figure 6.16: Branch Mis-prediction Rates (percent) in the baseline system. From left to right, three bars for each benchmark shows the result for the overall miss rate, miss rates in user code, and the miss rate in OS code, respectively.

tial range in the branch mispredictions. OLTP exhibits the highest mis-prediction rate of nearly 6%, while pgoltp shows the lowest rate of 2.7%. However, unlike IPC that shows worse behavior for OS code, the branch mis-prediction rates are lower in OS code than the user code in several benchmarks. For example, the user code in Apache sees a misprediction rate of 5.9%, while that in the OS code is 4.9%. In Apache, the larger miss rate for user code is primarily due to destructive interference from the OS code, where the benchmark spends most of its time. But in OLTP, the higher miss rate in the user code is from the inherent nature of its control flow, since most of its time is spent in the user computation. For pmake and pgoltp, the branch mis-prediction rates in OS code is more, due to the much shorter phases of OS computations, and the interferences from the relatively longer user computation phases.

Benchmark	TAP			SAP		
	Overall	OS	User	Overall	OS	User
Apache	29.7%	23.27%	38.47%	24.78%	15.50%	38.67%
OLTP	15.37%	87.10%	7.65%	15.45%	89.11%	7.06%
Zeus	12.64%	5.85%	44.36%	8.09%	0.86%	44.06%
pgoltp	10.53%	24.89%	5.93%	7.53%	16.58%	4.39%
pmake	2.52%	22.92%	0.74%	2.16%	10.23%	1.42%

Table 6.1: Percentage Improvements in YAGS Mis-prediction Rates

Table 6.1 presents the improvements in the YAGS mis-prediction rates achieved through CSP policies on an OPMS. The table presents the overall improvement (in percent), as well as improvements from the user code and OS code, exclusively.

TAP performs well across several benchmarks, except pmake. In Apache, TAP is able to improve the overall mis-prediction rate by 29.7%, while OLTP shows a 15.37% improvement. The improvement in Apache is derived from significant improvements in both user code and OS code. However, in OLTP, this improvement is primarily derived from the OS code, where TAP shows 87.10% improvement. Due to the destructive interference from the large user code footprint

in OLTP, the branch mis-prediction goes up in the OS code for OLTP. By separating user code and OS code execution, TAP avoids this destructive interference, dramatically improving mis-prediction rate in the OS. TAP is less effective for the user code in OLTP, as well as pgoltp.

SAP shows comparable improvements in the branch mis-predictions rates, achieving 24.8% and 15.53% in Apache and OLTP, respectively. However, the improvements for the OS code are lower than that in TAP, for several cases. While TAP achieves 23.27% improvement in the OS code for Apache, SAP shows only 15.5%, bringing down the overall improvement rates in SAP. This result is somewhat counter-intuitive since SAP achieves better improvement of instruction misses, indicating more localized instruction references (see Figure 6.5).

By concentrating particular system calls on particular cores, SAP multiplexes the execution of the same system call from multiple threads. However, the code path exercised in a system call can vary depending on data structures it operates on. For example, *read* system call on a network socket can take different code path than when invoked on a regular file descriptor. While repeated executions from the same code region in a specific system call allow SAP to improve the locality in the instruction caches, computations from the same system call with different arguments, invoked by multiple threads, lead execution through different code paths within the same region. This repeated difference in the branches from the same region increases the aliasing problem in the branch predictor, and causes higher mis-predictions, compared to TAP.

6.3 Computation Transfer Overhead

This section presents quantitative data on the computation transfer overhead — one of the major implementation challenges for realizing CSP like schemes in an OPMS. The overhead comes from two major components: latency and bandwidth. Saving the thread register state and restoring the state requires multiple store and load operations, and similar to other memory references they

incur latency overhead. The bandwidth overhead is primarily a result of interconnect messages required for communicating the register state among on-chip cores. Other data references that occur because of resuming computation on a different core than where the computation was paused, appear among rest of the messages. Because these communications through the interconnect are implicit, they are harder to identify. Therefore, only the bandwidth overhead from the explicit communication of thread state is illustrated in this section.

6.3.1 Latency

Figure 6.17 presents the computation transfer overhead seen in TAP. The two bars in each benchmark show the latency to save the register state (left), and the latency to restore the register state in the remote core (right). The result for the other CSP scheme SAP is identical to TAP, and hence omitted.

Saving the thread state involves a sequence of store operations that saves the register state in a pre-determined memory location (nearly 2.2KB, as discussed in Section 3.3.1). However, many of these locations are already cached in the L1 data caches, because these locations were accessed when the computation resumed on the current core. Due to frequent CT events, much of this state remains in the cache, and therefore saving state becomes mostly a local cache operation, reducing the latency observed. Across different benchmarks, the latency for saving state is similar, showing slight variation: 73 cycles in Apache and Zeus to 79 cycles in pgoltp. Since the computation phases in pmake and pgoltp are longer, the average latency in saving state is increased due to the occasional eviction of cache blocks maintaining thread state from the L1 caches.

Restoring state involves a sequence of load operations to access the memory locations containing the register state of a particular VCPU. This thread state was previously saved in a different core, and therefore resides in a remote L1 data cache. Consequently, restore operation requires

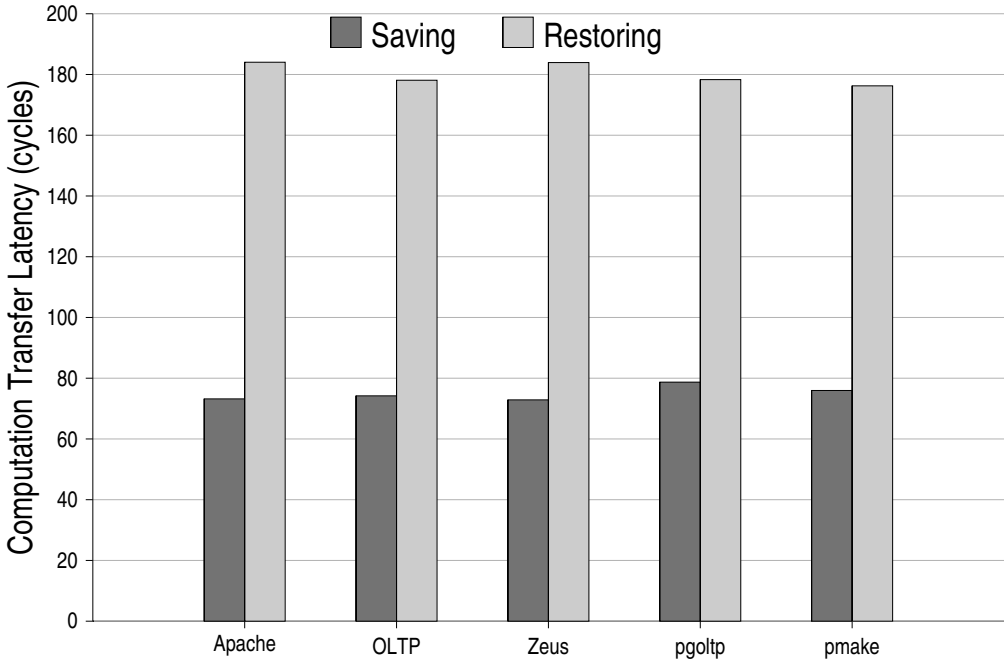


Figure 6.17: Average Latency Overhead of Computation Transfer (in cycles). Two bars for each benchmark illustrates the overhead in saving the thread state (left), and restoring the thread state (right), respectively.

several cache-to-cache transfers, increasing the latency seen for this operation. The latency seen is again similar across different benchmarks, as there is no difference in the basic mechanisms across different workloads. The average latency ranges from 176 cycles in pmake to 184 cycles in Apache.

While not negligible, this latency overhead from CT is quite modest in the multicore, even by utilizing a standard cache coherence and on-chip interconnect. The key reason behind this modest overhead stems from the superior bandwidth of on-chip communication. Although each CT requires communication of a significant volume of thread state, the entire communication is easily pipelined, either in the local cache or through the on-chip interconnect, dramatically reducing the end-to-end latency of the entire operation.

A special purpose interconnect that supports communication between on-chip cores without going through the caches can further reduce this computation transfer (e.g., using a scalar operand network [106]). However, the overall CT latency in the runtime is less than 2% in all cases (discussed later in Section 6.4.1), and even completely eliminating this latency is unlikely to make a qualitative difference in the results presented in the rest of this chapter.

6.3.2 On-chip Interconnect Bandwidth

Cache-to-cache messages, as well as access into the shared L2 consume on-chip interconnect bandwidth. While improving the efficiency of resources that exploit instruction stream properties, CSP schemes also incur additional costs in the interconnect bandwidth. An important design consideration is to quantitatively estimate the relative savings of the L2 access (by reducing L1 misses) against the overhead of computation transfer.

Figure 6.18 shows the comparison of interconnect bandwidth consumption between the baseline system and the two CSP schemes. For each benchmark, three bars show the result for the baseline (left), TAP (middle), and SAP (right), respectively. Each bar is further broken down to show: (a) address messages (requests or responses that do not include data), (b) data messages (data responses and write-backs), (c) bandwidth overhead from address messages due to CT, and (d) bandwidth overhead from data messages due to CT.

Despite an increase in L1 store misses for most benchmarks, overall L1 misses (which include both instruction and data) decrease for all the server workloads when using both TAP and SAP. This results in a noticeable improvement in the on-chip interconnect traffic and bandwidth used by these workloads. Except pgoltp, all benchmarks show an improvement in aggregate interconnect bandwidth usage when employing TAP. This improvement is achieved even after accounting for the bandwidth overhead from CT messages. Apache sees an improvement of 13%, while OLTP

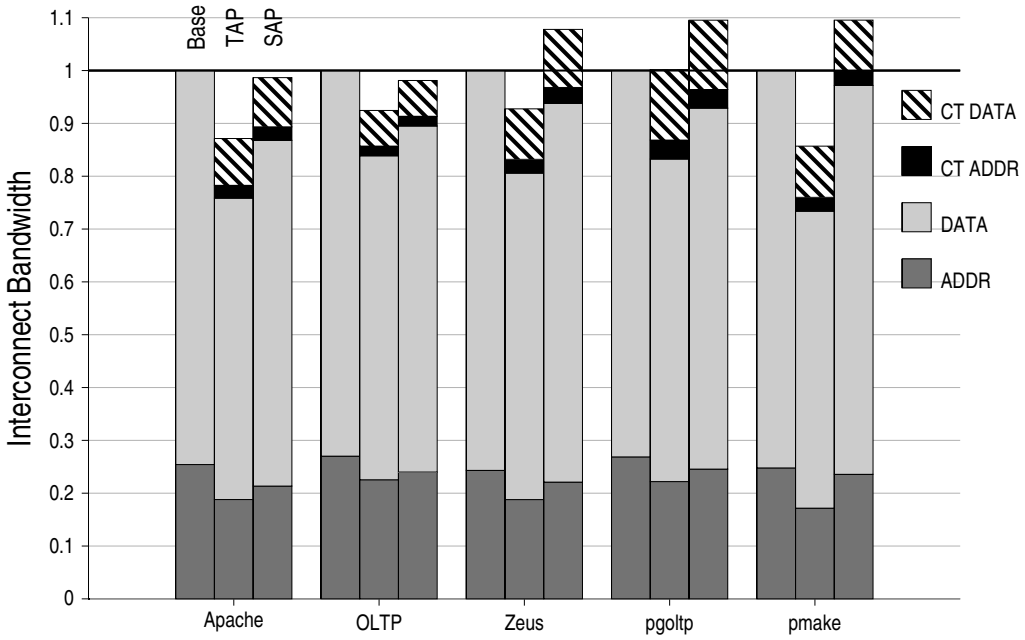


Figure 6.18: Interconnect Bandwidth Relative to Baseline

and Zeus both observe an 8% improvement. The total bandwidth overhead from CT messages in TAP varies from 15% in pgoltp to 8% in OLTP. In contrast, SAP shows an overall degradation in the bandwidth consumption, while exhibiting marginal improvement for Apache and OLTP. The additional load and store misses in SAP are the primary reasons behind this degradation.

6.4 Performance Impact

This section shows the overall performance impact of the two CSP schemes. Furthermore, it analyzes the two major design trade-offs in CSP (latency overhead from CT and core contention), and their impact on performance. To validate the CSP implementations, this section also revisits the code commonality characteristics (see Figure 4.1), and illustrates the outcome of this property

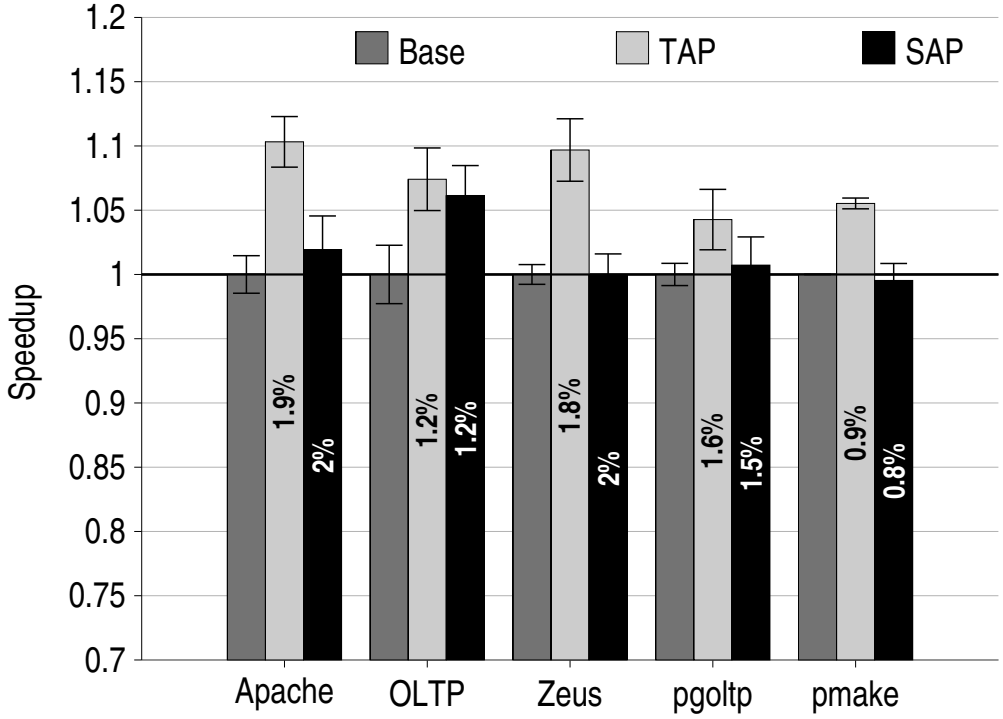


Figure 6.19: Speedup from CSP Techniques. For each benchmark, three bars show the result for the baseline (left), TAP (middle), and SAP (right), respectively. *Labels on TAP and SAP bars represent the runtime overhead of computation transfer*

in CSP.

6.4.1 Overall Speedup

The improvement in memory latency and branch prediction accuracy seen in Section 6.2 is likely to give speedups for the CSP schemes. But, the overhead from computation transfer is expected to diminish some of the runtime benefits. Therefore, it is instructive to inspect the overall speedups obtained through CSP.

Figure 6.19 shows the speedup achieved using the two CSP techniques. For each benchmark,

the figure shows the result for the baseline (left), TAP (middle), SAP (right), respectively. The runtime overhead of CT latency, measured by the time taken to save and restore register state, is shown as a percentage of the runtime in TAP and SAP. The cost of incurring additional cache misses for other data, purely due to CT, is modeled, but not included in this label. Even after including this runtime overhead, CSP yields significant improvements in the average memory response latency and branch misprediction rates for all the benchmarks except pmake, directly impacting the runtime of these workloads.

TAP showed dramatic improvement in the instruction locality, reducing the L1 instruction misses by 38% and 44% in Apache and Zeus, respectiely. As a result, TAP achieves maximum speedup in these two workloads, both achieving 10% speedup. Due to the large instruction stall component in OLTP, even smaller reduction of instruction misses results in a modest speedup of 7%. Speedups achieved by TAP in the other two benchmarks are smaller: 4% in pgoltp, and 5% in pmake, respectively.

SAP performs best for OLTP, achieving 6% speedup, but the speedups achieved in SAP is lower than TAP for all benchmarks. The relative degradation in data locality for SAP, combined with only marginal benefits in instruction locality over TAP, leads to its poor performance.

Unlike Apache and Zeus, pgoltp shows lesser benefit from CSP, as the performance of this workload is dominated by large data stalls. The latency in data references frequently cause stall in the front end as the instruction window fills up with dependent instructions of a long latency load (see Figure 6.6). Since CSP shows no benefit in reducing these data stalls, the overall impact of performance from instruction stall reduction does not make a noticeable impact.

Due to the superior communication bandwidth and latency between on-chip cores, the runtime overhead from CT is small. Across TAP and SAP these overheads are similar and comparable, since both these schemes triggers CT during the change of execution mode (i.e., OS to user and

user to OS). Among all workloads, the overhead is highest in Apache: 1.9% in TAP and 2% in SAP. Zeus also show comparable overhead of 1.8% and 1.9%, respecitviely. Both these benchmarks make more frequent transitions of the computation mode, leading to an increase in CT events and their associated overhead. CT events are triggered, on average, every 5K instructions in Apache (most frequent), and every 20K instructions in pmake (most infrequent). Due to the infrequent transitions in pmake, it exhibits minimum overhead of 0.9% among these benchmarks.

The performance of pmake, the only non-server workload evaluated in this dissertation, is less impressive. However, this result is expected, and two key factors contribute to this result. First, pmake shows the least amount of code sharing, especially among user code. Consequently, collocating user computation leads to only marginal improvement in instruction locality (Figure 6.5). Second, the component of data communication between user and OS is substantially more in pmake than other applications (relative contribution of C in Figure 4.3(b)). Since OS-to-user communication usually happens within a short interval, it typically leads to local L1 hits in the baseline, while causing cache-to-cache transfers in TAP and SAP. Moreover, especially for pmake, TAP and SAP only save a subset of misses in the other categories, leading to additional data misses overall.

6.4.2 Contention Overhead

In addition to the latency overhead from CT, the core contention in CSP also negatively impact the overall runtime (see Section 4.5.2). Figure 6.20 shows the directly measurable overhead from this core contention in the two CSP schemes. This contention overhead is measured by the average fraction of the time each VCPU is paused. While executing the CSP schemes, the occasional core contention necessitates pausing the computation from VCPUs. Depending on the runtime configuration and the dynamic execution pattern, different VCPUs are paused at different times. For each

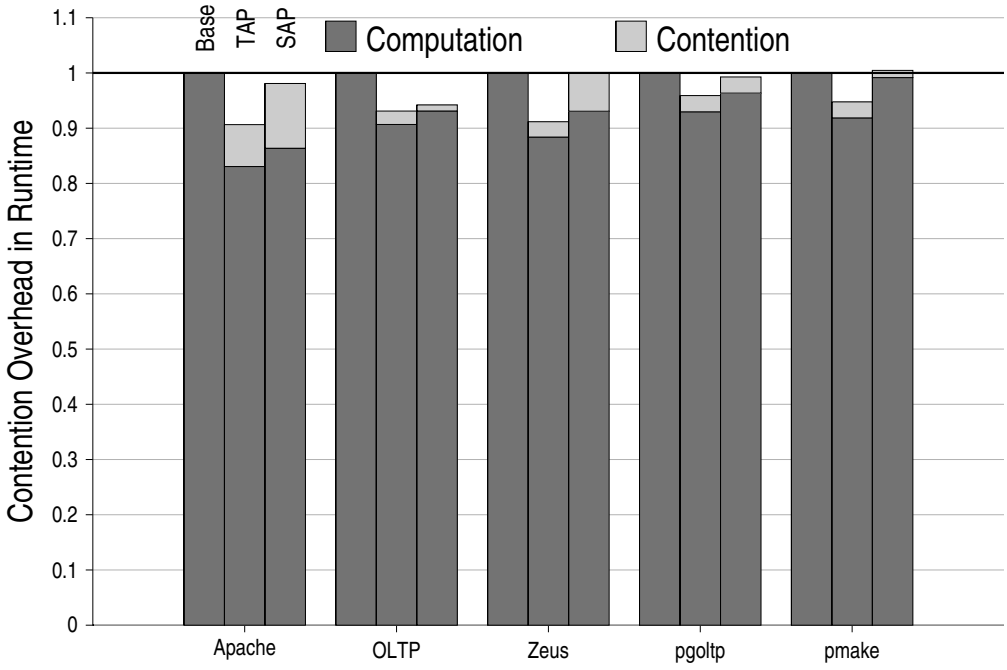


Figure 6.20: Runtime overhead from contention. Three bars for each benchmark shows the result for the baseline (left), TAP (middle), and SAP (right), respectively. Each bar is further broken down to show the fraction of the runtime spent in active computation (bottom), and the fraction wasted due to the contention overhead (top), respectively.

VCPU, the figure shows the average fraction of the runtime spent in the active computation (bottom), and the fractional time wasted due to contention (top), respectively. All the VCPUs in the baseline system are always in the active computation mode, and do not experience any contention.

Across different workloads, the contention overhead is modest in TAP. Apache shows the maximum overhead of 7.4%, while OLTP shows the minimum 2.4% overhead. The source of contention is different for these two benchmarks. In Apache, only user computations create core contention, as there are only four cores provisioned for user computation from eight VCPUs (see Section 5.5.1). Since the number of cores provisioned for OS computation matches the total number of such fragments (eight), OS computation does not create contention in Apache. The provisioning

is opposite in OLTP, and therefore the OS computations lead to contention in this benchmark.

Compared to TAP, SAP shows more contention in the OS intensive workloads Apache and Zeus, which manifests as a larger increase in execution cycles (by 12% in Apache). The increase in contention is because of more restriction in assigning OS computation in SAP. Specific system calls are assigned to specific cores, although commonly executed system calls are allowed to execute on more than one cores. Despite such an assignment, concurrent execution of a particular system call is limited in SAP. TAP places no such restrictions, and for Apache and Zeus, it allows concurrent execution of OS computation from all the VCPUs. On the other hand, the contention in pmake is substantially lower in SAP than TAP. SAP shows an overhead of only 2%, while TAP shows an overhead of nearly 5%. Here the assignment policy in SAP reduces contention because it allows OS computation from two VCPUs, executing two different system calls, to proceed simultaneously. However, in TAP if OS computations from these two VCPUs are mapped to the same core, they must execute sequentially, leading to the higher contention overhead.

6.4.3 Code Reuse With TAP/SAP

Figure 6.21 again presents the code commonality characteristics for the baseline, this time alongside the two CSP schemes. The *Common* portion from Figure 4.1 is further broken down into 2-4 cores and more than 4 cores. In every case the *Universal* section virtually disappears for TAP and SAP (as expected). There is a significant increase in the fraction of code used only up to 4 cores in TAP/SAP, compared to the baseline.

Though one expects a more prominent increase in the codes accessed exclusively or among only 2-3 cores in SAP (as described in Section 5.5.1, SAP provisions many common system calls to two cores, a few to three cores), especially for the two web-servers, the results in Figure 6.21 indicate otherwise. This is mainly because using system call numbers alone is not sufficient for

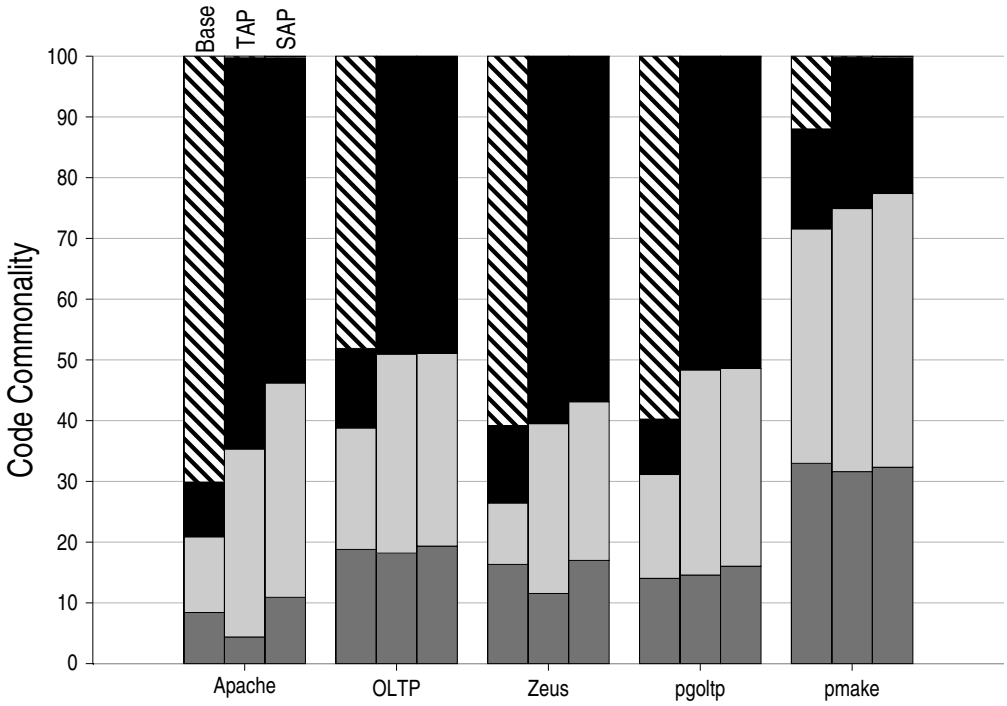


Figure 6.21: Code Reuse with TAP/SAP. *Each bar is broken down to show different degrees of reuse. From top to bottom these are: Universal, more than 4 cores, 2-4 cores and Exclusive, respectively.*

identifying similar and dissimilar computation fragments (or code paths). For example, consider two instances of the *read()* system call. If they are invoked with different types of file descriptors (e.g., a network socket and a regular file), then they are likely to execute substantially different code, though they are executed in the same core in SAP. Likewise, a *read()* and *write()* to the same type of file descriptor is likely to access similar code and data. To fully exploit the potential of CSP, the system requires better techniques to distinguish different computation fragments. Identifying and evaluating such techniques justify a separate and new body of independent research, and left as a possible future work stemming from this dissertation.

6.5 Energy Efficiency

Improving energy efficiency is one of the major goals of an OPMS system. Server applications demand increasing energy efficiency, but unlike many other domain of applications, these applications are not willing to compromise on performance [31]. Therefore, energy efficiency of the system must improve, while simultaneously improving the execution time or throughput in these applications. The previous section demonstrated the improvement in the execution time using an OPMS. This section will now focus on the impact of energy efficiency of this OPMS in the context of the multithreaded applications evaluated in this dissertation.

The section is organized as follows. Section 6.5.1 presents the utilization of on-chip cores, to demonstrate how the OPMS maintains the same *Simultaneously Active Fraction* of the baseline system, while enhancing the performance. Section 6.5.2 shows the total energy expended in the on-chip cores, and the contribution from both dynamic and leakage energy. Section 6.5.4 presents the thermal characteristics of the on-chip cores. Next, Section 6.5.6 discusses the energy expended in the on-chip caches. Finally, the overall energy efficiency of the system is presented in Section 6.5.7. This dissertation uses the *energy-delay* product as the metric of energy efficiency, which estimates the energy efficiency of the system by calculating the product of the execution time (delay), and the overall energy expended during this execution in the system.

6.5.1 Core Utilization

OPMS provisions more processing cores than the baseline system, but the number of simultaneously active cores never exceeds that in the baseline (eight). Moreover, both the assignment policies from CSP selectively assign computation on the on-chip cores, further reducing the simultaneously active cores. Figure 6.22 presents the average number of on-chip cores utilized by TAP and SAP. The baseline system, which always utilizes the available eight cores is also shown in the

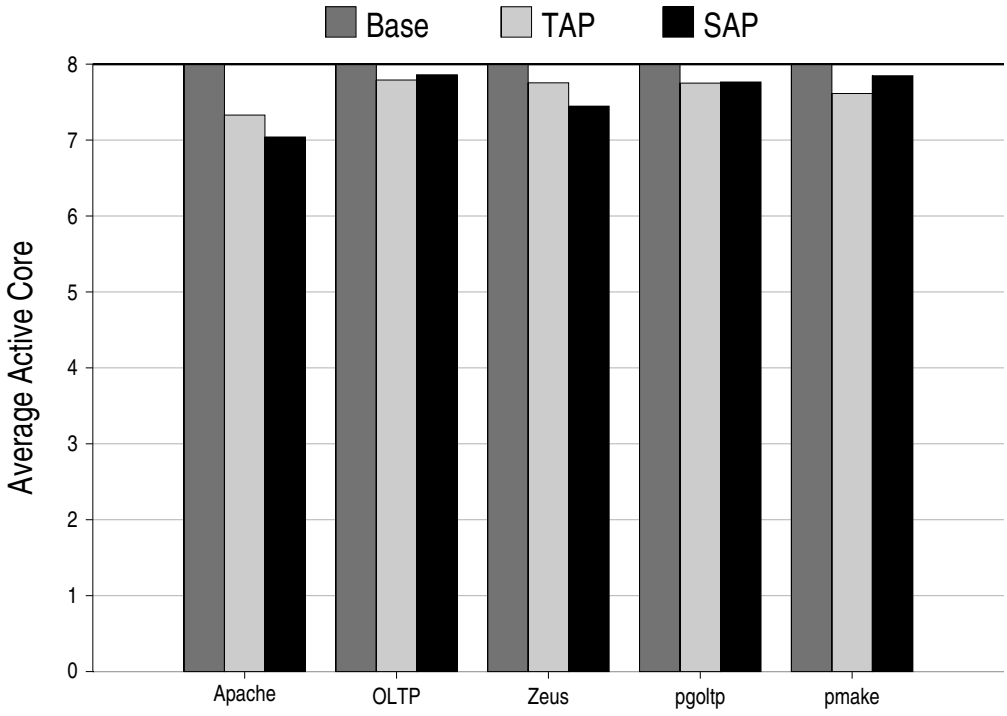


Figure 6.22: Comparison of the average number of on-chip cores engaged in active computation. For each benchmark, three bars show the result for the baseline system, TAP and SAP, respectively.

figure. This core utilization has significant impact on several results presented later, and is critical to understand the design trade-offs of the OPMS design paradigm.

Across all benchmarks, the average number of on-chip cores engaged in computation is less than the baseline in TAP and SAP. This reduction in the simultaneous core activity saves substantial power, since the inactive cores benefit from leakage tolerant techniques that dramatically reduce their static power (within 1-2% of the static power in an active core as discussed in Section 3.3.2). Compared to the power consumption of an active core, which includes both its dynamic and static power, the power consumption from an inactive core is negligible. Therefore, even a partial reduction in the average number of active cores, allows multiple inactive cores to subsist in

an OPMS, while remaining within the TDP of the baseline system.

TAP achieves more reduction in the OS intensive workloads: Apache and Zeus. In Apache, the average number of active cores in TAP is 7.3, while that in Zeus is 7.5. In these benchmarks, this reduction is a result of occasional contention on user cores. For example, when more than four VCPUs attempt to execute their respective user computations, some of them are paused, resulting in active computations in four user cores, and few OS cores (the number of active OS cores is less than four). Collectively, less than eight cores are employed in this scenario. Likewise, the average number of cores employed while executing pmake is 7.61 due to the contention in the OS cores.

The reduction in core activity is more in SAP for the OS intensive workloads due to additional restrictions placed on assigning OS computation on the on-chip cores. SAP employs averages of 7.0 and 7.4 cores for active computation, while executing Apache and Zeus, respectively. On the other hand, pmake in SAP exhibit more core activity than TAP. Much of these results for average active cores is expected from the extent of contention overhead described previously (see Section 6.4.2). Higher the contention, lower the average active cores, while lower contention increases the average active cores.

6.5.2 Core Energy

The improvement in runtime and compute efficiency through superior instruction throughput and branch prediction is expected to improve the energy expenditure from the core logic in an OPMS. Figure 6.23 presents this aggregate energy expended in all the on chip cores. Three bars for each benchmark show the total energy expended on the on-chip core logic for the baseline (left), TAP (middle), and SAP (right), respectively. The energy component shown in this figure includes all the on-chip processing cores, but do not include the private caches in the cores. Each bar is also broken down to show the relative contribution from dynamic energy and static energy from leakage.

The relative contribution of the static energy varies across these workloads. For example, in Apache static energy contributes 34% of the overall core logic energy, while in pmake it only contributes 20%. Due to the lower IPC in Apache (see Figure 6.1), a substantial fraction of the pipeline logic circuits spend the execution time waiting for memory references to satisfy. During this wait period, aggressive clock gating saves much of the dynamic energy, thereby increasing the relative contribution of the static energy. In contrast, the high IPC in pmake keeps the pipeline logic busy in computation, and therefore it exhibits lower contribution from the static energy.

TAP reduces the total energy from the core logic in all the applications. In Apache, this reduction is 10%, while Zeus see a reduction of 5%. Bulk of this energy savings in TAP is derived from the static energy component, although Apache and Zeus show marginal savings in the dynamic energy due to the superior branch predictor performance. The saving in static energy in TAP is counter-intuitive, as by employing more processing cores, the leakage was expected to rise due to its dependence on area [22].

However, three key reasons play important roles in this static energy savings: (a) runtime reduction, (b) reduction in core utilization, and (c) temperature reduction. The static energy is directly proportional to the runtime, and hence improvement in the runtime leads to savings of the static energy. Furthermore, although TAP employs more core logic area, the total core logic area in active computation mode is less than that in the baseline multicore due to the reduction in the average core utilization (see Figure 6.22). Bulk of the leakage is saved from the cores that are not engaged in active computation using the MTCMOS technology as discussed in Section 3.3.2. Apache sees maximum benefit in static energy savings for TAP, since both the runtime reduction and the reduction in core utilization is highest in this benchmark among the rest.

Leakage has an exponential dependence with the temperature [93]. Similar to the recent trend in power estimation methodology [66], this dissertation models leakage as a temperature dependent

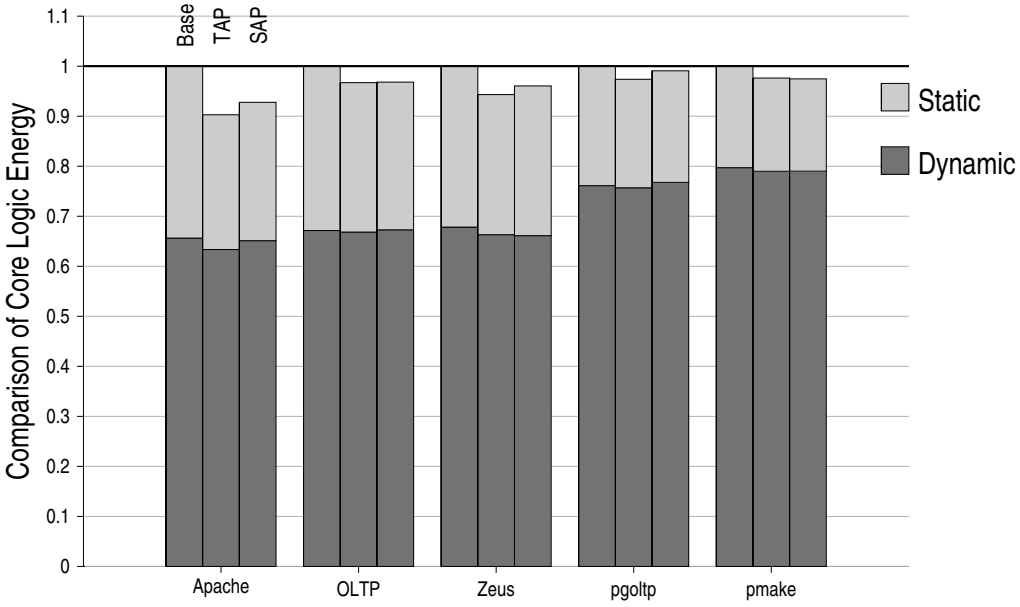


Figure 6.23: Comparison of Energy Expenditure in the Core Logic. For each benchmark, three bars show the result for the baseline system, TAP and SAP, respectively. Each bar is broken down to show the dynamic (bottom) and static energy (top) components

fraction of the dynamic power (see Section 5.4 for details). The reduction in temperature in various core logic components (presented later in Section 6.5.4), combines with other factors discussed above, to save the static energy.

The benefit in SAP is less than that of TAP in all cases. Despite a greater reduction in core utilization in SAP, the improvement in runtime is lower in SAP, which increases the static energy component.

6.5.3 Inactive Periods

The effectiveness of the MTCMOS technique to control leakage from the on-chip cores, as demonstrated in Figure 6.23, is an outcome of the length of the inactive periods seen in the OPMS cores.

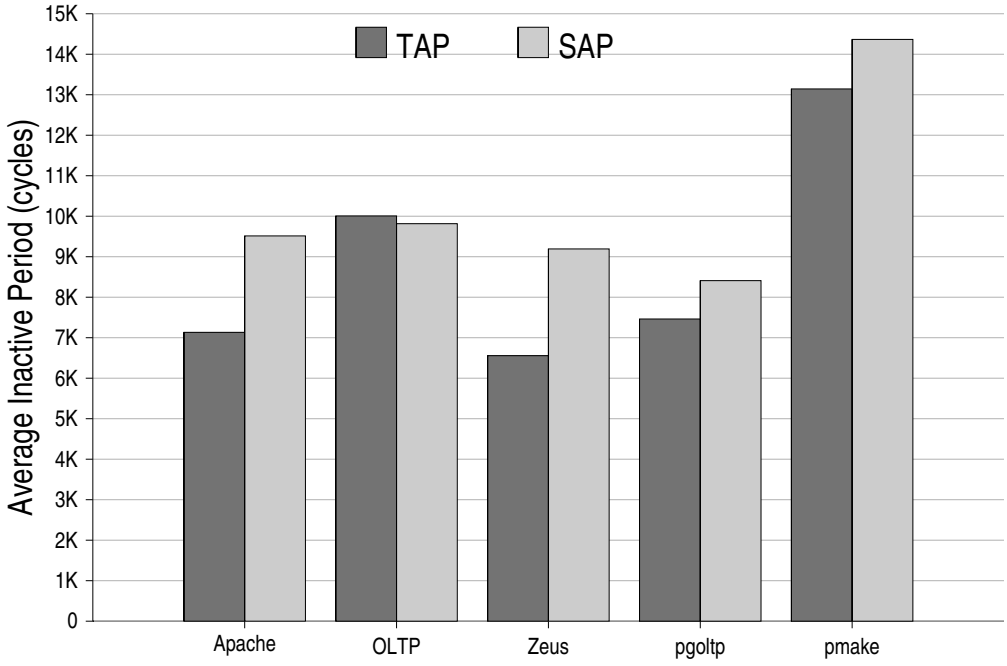


Figure 6.24: Average inactive period length in cycles. For each benchmark, two bars show the result for TAP and SAP, respectively.

Figure 6.24 shows the average length of inactivity seen in the OPMS cores across several workloads. These average periods are one to two orders of magnitude higher than the breakeven points reported in MTCMOS designs (10-100 cycles as discussed in Section 3.3.2), allowing this technique to be highly effective for reducing leakage in an OPMS. The periods are higher in pmake due to comparatively longer user phases and infrequent mode changes. SAP shows longer inactive periods, as the activity in OS cores are typically not uniformly distributed as compared to TAP.

6.5.4 Thermal Characteristics

Both the CSP schemes on OPMS achieves a reduction in aggregate energy expended in the core logic. However, since this reduction is achieved even while exploiting more processing cores, each

processing core sees a much larger reduction in the energy expended, compared to the baseline core—more than that could be achieved by simply employing additional cores for the same work. Furthermore, due to the specific assignment policies, both the CSP schemes avoid continuous use of processing cores. Together, the average energy reduction and intermittent use for computation, lead to a decline in temperatures seen in the core hotspots, improving the thermal characteristics of the multicore system. This reduction is primarily derived from the large reduction of per-core energy from using 12 cores instead of eight in the baseline, and partially due to the additional reduction from the improvement in the aggregate core energy. However, as the energy expenditure and temperature of a micro-architecture is closely coupled with non-linear dependence [97], it is impossible to accurately estimate the relative contributions in temperature reduction from these two modes of per-core energy reduction.

Figure 6.25 shows the thermal profile of two important hotspots in the processing cores: the instruction window and the register file [75]. For each of these structures, the figure shows the steady-state temperature in each benchmark for the baseline system (left), TAP (middle), and SAP (right), respectively. These steady state temperatures are obtained by collecting the power trace from the timing runs, and then simulating substantially longer runs to amortize the effect of transient temperatures [97]. See Section 5.4 for more details. The results represent the average across all the cores in the system.

Among all the micro-architecture components, these two blocks show the maximum temperature in the multicore system modeled in this dissertation. The baseline exhibits higher temperature in pmake and pgoltp due to the corresponding higher IPC and pipeline utilization in these benchmarks. Higher utilization cause a corresponding higher power consumption, increasing the temperature of critical blocks in these two benchmarks.

TAP achieves modest reduction in temperatures in both instruction window and the register

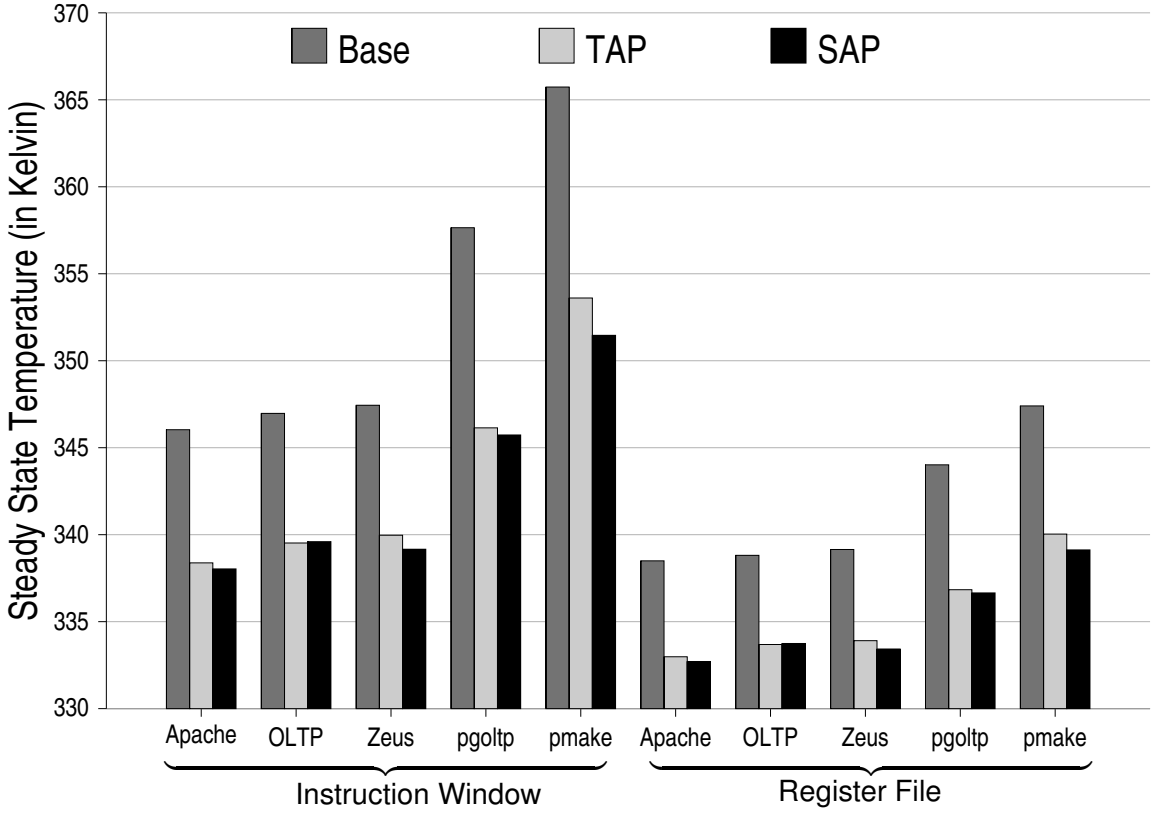


Figure 6.25: Comparison of Thermal Characteristics. Two sets of bar for all the benchmarks present the results for the Instruction Window (left), and Register File (right). For each benchmark within a given set, the three bars represent the baseline (left), TAP (middle), and SAP(right).

file. For pmake and pgoltp, which show highest utilization of the micro-architecture components, diminishing the heat load in each core reduces temperature most. In contrast, Apache, OLTP, and Zeus achieves smaller reduction since heat load on processing cores are less intense in the baseline for them. In Apache, TAP reduces the average temperature of the instruction window and the register file by 9 degrees and 6 degrees, respectively. On the other hand, pmake achieves a reduction of 12 and 8 degrees, respectively. The reduction in temperature for SAP is higher than TAP in several cases due to the lower core utilization in SAP.

6.5.5 Comparison of di/dt

While the intermittent use of processing cores improves their thermal characteristics, it can potentially introduce voltage fluctuations in the power supply network due to the variation in transient load or current (see Section 3.3.3). However, unlike uniprocessors, several on-chip cores contribute their respective current load on the power supply network. Consequently, the transient effects of disabling/enabling one core in the system is relatively smaller in the multicore system. Moreover, the baseline system also experience substantial current transients due to clock-gating and occasional burst in activity [61]. Therefore, it is imperative to inspect the transient effects in an OPMS, relative to the baseline.

As none of the experiments presented thus far vary the voltage supply, the current transients can be estimated by observing the variations in the overall chip power drawn [89]. Current variations occur at all time granularities, but its variation at the resonance frequency is critical for voltage fluctuations [61, 89]. Table 6.2 presents the comparison of current variation between all possible consecutive windows of 64 cycles. Results for other time-periods were also observed, but they do not show any significant deviation.

Benchmark	Average Current Variation		Maximum Current Variation	
	TAP	SAP	TAP	SAP
Apache	4.75%	2.58%	4.67%	-1.57%
OLTP	4.75%	-0.77%	5.79%	2.75%
Zeus	6.81%	9.97%	5.54%	1.43%
pgoltp	0.73%	-1.52%	0.55%	1.02%
pmake	1.83%	2.72%	1.43%	1.73%

Table 6.2: Increase in Current Variations (lower is better).

The table shows the percentage increase in the average current variation during the simulation run, and the worst case variation seen between any two consecutive windows. Results are shown

for TAP and SAP, relative to the baseline. The OPMS schemes show rise in the current variation for most benchmarks, but variations remain within 10% of the baseline system. Both pgoltp and pmake show substantially smaller rise in current variations, as computation transfers—the main cause of current transients in OPMS relative to the baseline—are less frequent in these two benchmarks.

There are several techniques that can deal with load transients in a multicore system. To allow per-core voltage-frequency control, many modern systems are incorporating on-chip voltage regulators that are effective in neutralizing small but frequent current transients [61]. Other solutions also exist that can increase the effectiveness of decoupling capacitance by actively increasing the voltage across capacitance terminals [43]. A combination of these solutions are likely to mitigate the challenge of the increased di/dt in an OPMS.

6.5.6 Cache Energy

The OPMS design introduces several trade-offs in the cache energy consumption. Compared to the baseline, OPMS schemes have a larger number of L1 caches. Unlike the core logic, where state retention is unnecessary, caches and branch predictors must retain their state in the inactive state. Consequently, they continue to leak in the drowsy mode (albeit at a lower rate). However, less leakage from a smaller L2 cache helps compensate the additional leakage. Furthermore, OPMS is likely to give some improvement in dynamic energy by reducing L2 accesses, while improving leakage energy through improved runtime. These trade-offs are now investigated with quantitative data.

Figure 6.26 shows the comparison of energy expended in the cache hierarchy. Again, the three bars show results for the baseline, TAP, and SAP, respectively. Each bar is further broken down into five parts: from the bottom these are L1 instruction cache, L1 data cache, L2 dynamic energy, L2 static energy, and the energy overhead due to CT, respectively.

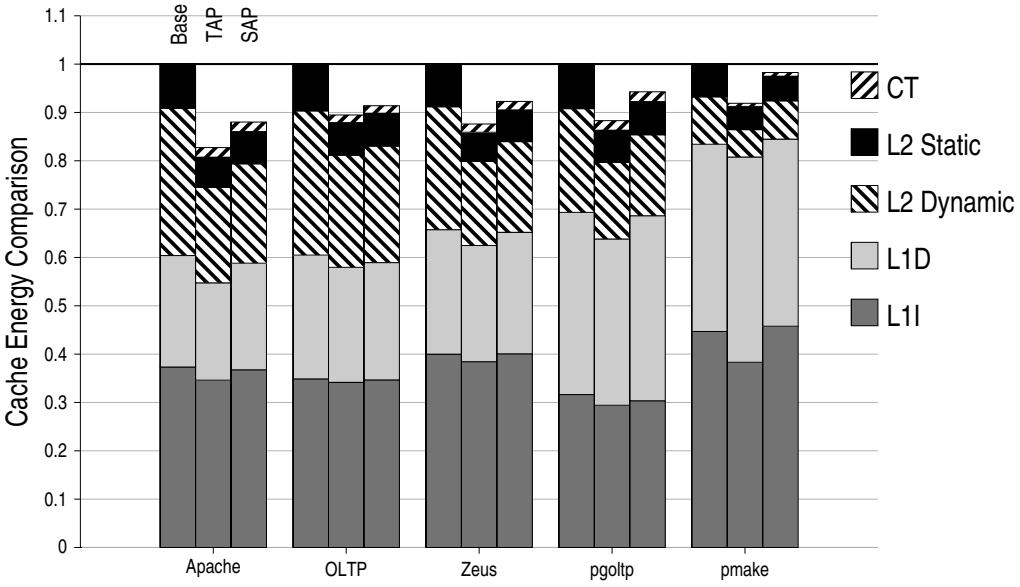


Figure 6.26: Cache Energy Comparison

TAP shows improvement in the total energy expended in the on-chip caches, even after accounting for the additional energy for caches accesses during CT events. In Apache, TAP achieves a substantial 20% improvement, while pmake shows a modest 8% improvement. SAP exhibits slightly lower improvements: Apache achieving 16% improvement, while pmake achieving 9% improvement. A major portion of this overall improvement is derived from the savings in the L2 caches.

Across all benchmarks, TAP shows modest improvements in L1 energy by balancing the extra leakage mainly by improvements in runtime, and partially by lower core utilization, which allows more than four cores to enable drowsy mode in their caches. Furthermore, in several cases, the dynamic energy is also improved due to lower misses, and reduced branch mis-speculations, resulting in fewer accesses.

In contrast, L2 energy shows significant savings in both the CSP configurations. L2 caches are

smaller in an OPMS, thereby saving leakage energy, and this savings is enhanced further by the improvement in the runtime. However, the bulk of the overall energy saving comes from reducing the dynamic energy through lesser accesses into the L2 cache. By enhancing the locality of references, CSP avoids destructive interference between the OS and the user computation modes, especially for instructions, resulting in better use of the private L1 caches, and substantially lower access into L2. Apache, for example, observes 35% and 32% reduction in L2 energy from TAP and SAP, respectively. Considering the individual components of dynamic and static energy from L2, Apache shows reductions of 35% and 32%, respectively in TAP.

The energy overhead for data transfers due to CT is marginal in the CSP schemes (up to 1.7% in Apache). Many of the references during CT event is satisfied by the local caches (e.g., saving state). Requests that go to the L2 often do not require to read the data array, since they typically access modified data from another L1 cache, saving significant energy compared to other L2 requests that need to access both the tag and data arrays.

6.5.7 Energy Delay

Figure 6.27 shows the improvement in the energy efficiency achieved using an OPMS. The energy efficiency of the system is measured by the product of aggregate energy consumed, and the total execution time (ED). Although server applications typically use ED^2 as their energy efficiency metrics, this dissertation chose the ED metric to lay more weight on the energy consumption. Three bars for each benchmark shows the result for the baseline (left), TAP (middle), and SAP (right), respectively.

TAP shows improvement in energy efficiency across these workloads. The most improvement is achieved in benchmarks which also show higher speedup, indicating more efficient use of micro-architectural structures in TAP. In Apache, TAP demonstrates a substantial improvement of 21%,

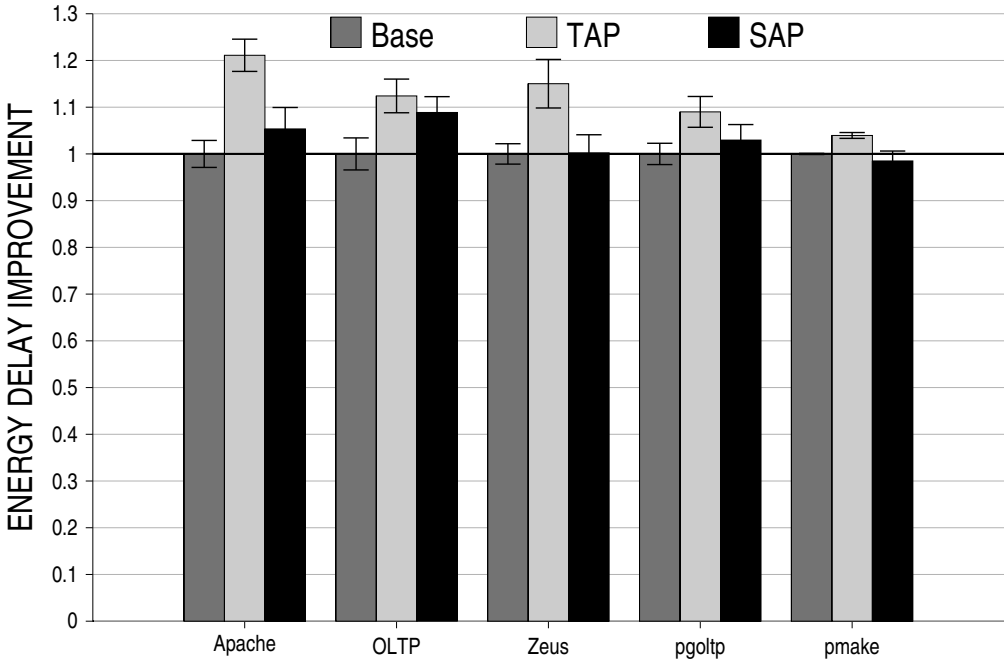


Figure 6.27: Improvement in Energy Efficiency. Energy efficiency is estimated using the energy delay product. Three bars for each benchmark show the results for the baseline (left), TAP (middle), and SAP (right), respectively.

while Zeus shows 14% improvement. On the other hand, pmake shows a marginal improvement of 3%. Interestingly, this improvement in energy efficiency is achieved by simultaneous improvement in both the execution time, and the energy expenditure in the system. SAP also shows improvement, albeit at a lesser extent than TAP. SAP achieves most improvement in OLTP (9%), while showing a marginal 2% degradation in pmake.

Figure 6.27 demonstrates the improvement in the energy efficiency achieved through the OPMS design. But the energy efficiency of the entire system depends on several other components (e.g., memory, I/O), which remain unaffected by the OPMS design. Consequently, the bottom line benefit in system efficiency from solely improving the energy efficiency of its multicore will be lower than that reported in Figure 6.27. However, a recent study has shown that multicores are still the

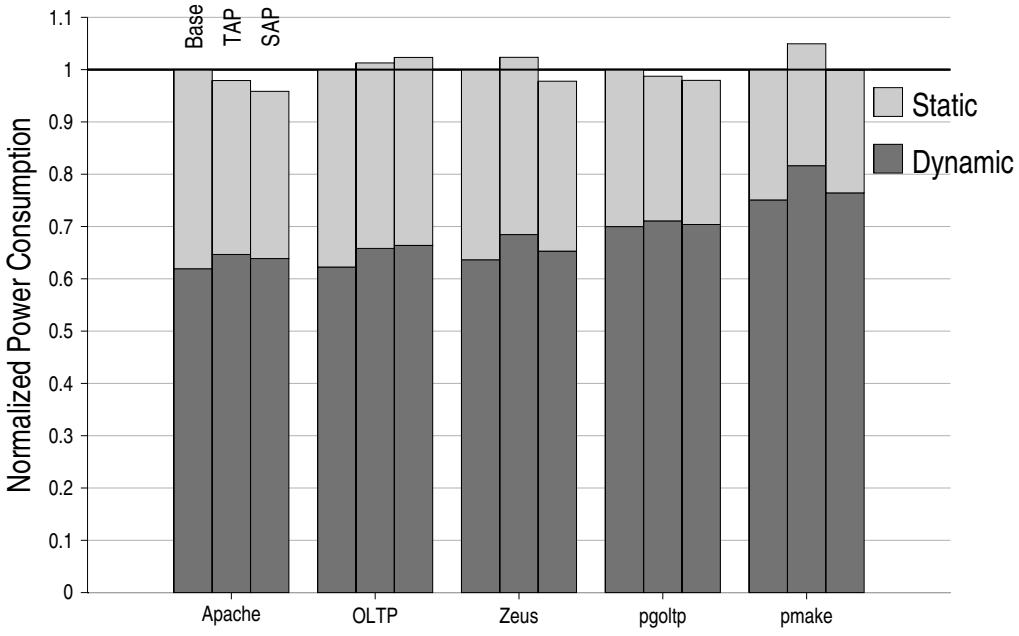


Figure 6.28: Normalized Power Consumption. Three bars for each benchmark show the results for the baseline (left), TAP (middle), and SAP (right), respectively. Each bar is also broken down to show the contribution from the dynamic power (bottom), and the static power (top).

leading contributors to the overall system power and energy in a large server setup, constituting a substantial 38% of the system energy [31]. An OPMS tackles this major component of system energy, and in combination with techniques that improve other system components, can yield superior energy efficiency of the whole system.

While the OPMS schemes improve the energy efficiency, it is imperative to inspect if it can be designed within the TDP of the baseline system. To illustrate this design aspect, Figure 6.28 shows the power consumption of the OPMS schemes, normalized to the baseline. Three bars for each benchmark shows the result for the baseline (left), TAP (middle), and SAP (right), respectively.

Overall, both TAP and SAP remains on or just above the TDP of the baseline. TAP exceeds the baseline TDP by 5% in pmake, but remain within 1% of the baseline TDP for all other workloads.

On the other hand, SAP exceeds the baseline TDP by 2% in OLTP, but remain at or below the baseline TDP for all other benchmarks.

The dynamic power is increased in TAP for several benchmarks, but this is compensated by the decrease in leakage power. The higher IPC observed in the computation phases in TAP increases the dynamic power, while the smaller L2 and the lower core utilization balance this rise by a reduction in static power. This increase is substantial in Apache and Zeus, which also show the most leakage reductions from their lower core utilization. For OLTP, SAP shows the maximum rise in dynamic power, which also results in a overall 2% power increase.

In summary, both the CSP schemes are able to achieve substantial improvement in various performance characteristics that are important for these server class workloads. Execution time is improved by 2-10%, while energy efficiency is improved by 5–21% in TAP, while remaining at or marginally above the TDP of the baseline system.

6.6 Sensitivity Analysis

This section presents a detailed sensitivity analysis of the evaluation results described earlier in this chapter. The purpose of this analysis is to understand the key contributing factors behind the improvement in various performance metrics demonstrated in the previous section.

So far, this chapter has evaluated CSP on an OPMS, where the multicore is provisioned with additional on-chip cores for increasing the flexibility in assigning computation. However, the CSP proposal can also be applied in a conventional multicore, and Section 6.6.1 presents a comparative study to demonstrate the relative contributions from CSP and OPMS, respectively. Although comparable in area and thermal design power, the OPMS employs a different micro-architecture than the baseline system. Section 6.6.2 presents a study that use the identical micro-architecture of the OPMS, but use this micro-architecture in the same spirit as the baseline system. The final

two sections inspect the impact of core micro-architecture by investigating performance results in multicores that use in-order cores (Section 6.6.3), and the impact of L1 cache sizes (Section 6.6.4).

6.6.1 Evaluating CSP in Conventional Multicore

The evaluation presented in Section 6.2 combined two aspects of multicore design paradigms: over-provisioning and locality enhancement via CSP. However, the computation assignment via CSP is not limited to a over-provisioned system: it can also be used in a conventional multicore [24]. Figure 6.29 presents the runtime comparison of the CSP in a conventional system with the baseline and CSP in a over-provisioned system. From left to right for each benchmark, the figure shows the result for the baseline, the TAP assignment of CSP in an OPMS, and the TAP assignment in a conventional multicore (marked as TAP_C). Each bar is broken down to show the fractional time spent in computation and contention, respectively.

Figure 6.29 demonstrates a significant performance loss for CSP—despite its enhanced locality—causing severe runtime degradations ranging from 22% in Zeus to 48% in pmake. As evident from the figure, the contention overhead is the primary factor in this degradation, nullifying the benefits of dynamically specializing each core's predictive structures. The fractional time spent in the computation is at or below that in the baseline system, and often comparable to TAP in an OPMS, indicating the effectiveness of locality enhancement of CSP. However, the restriction placed on assigning computation to achieve this locality enhancement, coupled with the lack of flexibility in assigning computation in a conventional multicore, lead to many fold increase in the contention overhead. For example, compared to TAP in OPMS, the contention overhead is $6X$ more in Apache, and $16X$ more in pmake, when using TAP on conventional multicore. The large performance improvement using TAP in OPMS compared to TAP in a conventional multicore—with its similar locality enhancement technique—indicates that the OPMS design paradigm is critical

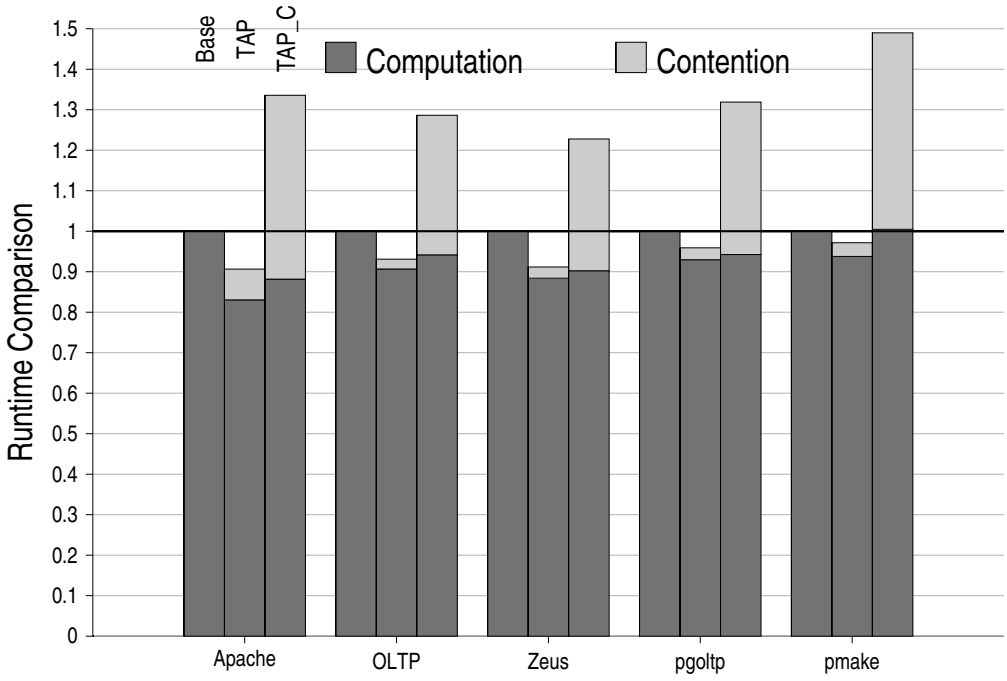


Figure 6.29: Runtime comparison with CSP in a conventional multicore. Three bars for each benchmark, from left to right, show the result for the baseline , TAP in an OPMS, and TAP in the conventional multicore (marked as TAP_C), respectively. Each bar is further broken down to show the fraction of the runtime spent in active computation (bottom), and the fraction wasted due to the contention overhead (top), respectively.

for achieving overall benefits in the execution time, instead of just locality enhancement.

The performance results for SAP (omitted) in a conventional multicore is worse than the results for TAP due to the additional restrictions placed on assigning OS computation, increasing the contention overhead even further.

6.6.2 The Role of the OPMS Micro-Architecture

The OPMS evaluated thus far employs a different micro-architecture than the baseline multicore, where the former contains more processing cores, but smaller L2 cache. Therefore, a key aspect

in this analysis is to comprehend the role of this alternate micro-architecture in several benefits demonstrated throughout this chapter. To resolve this issue, this section presents a comparative analysis where the same OPMS micro-architecture is employed, but in the spirit of the baseline system.

Unlike the OPMS, which restricts simultaneous computation on all cores, the baseline system targets full utilization of its on-chip cores. Therefore, this section compares a 12 core multicore system with a smaller 12MB L2, identical to the OPMS multicore. In this multicore, all the cores are exposed to the OS to schedule threads. Such a configuration can better exploit the application concurrency or thread level parallelism to improve throughput by executing 12 software threads simultaneously, instead of eight. Although, not all applications achieve good scaling behaviors, the workloads evaluated in this dissertation do indeed show good scaling properties, and ensure good use of additional cores.

Since this configuration would exceed the TDP of the baseline system when operated at the baseline frequency, this section also evaluates several 12-core systems where frequency scaling is applied to remain within the same power envelope of the baseline system. Voltage scaling is unlikely to be a viable means at the design point considered in this dissertation due to the increasing engineering challenges (see Section 2.3.1). Despite the impracticality of voltage scaling, the results will show the impact of using this means.

Comparing the baseline, OPMS and this new scheme that uses the OPMS micro-architecture imposes a methodological challenge. Both the baseline system and the OPMS expose eight VCPUs to the OS, allowing the use of the same workload checkpoint for each benchmark in all the experimental results presented so far. However, exposing 12 VCPUs from the OPMS multicore compels the use of a different checkpoint where 12 application threads are scheduled on those VCPUs. To mitigate this challenge, experiments presented in this section use substantially longer

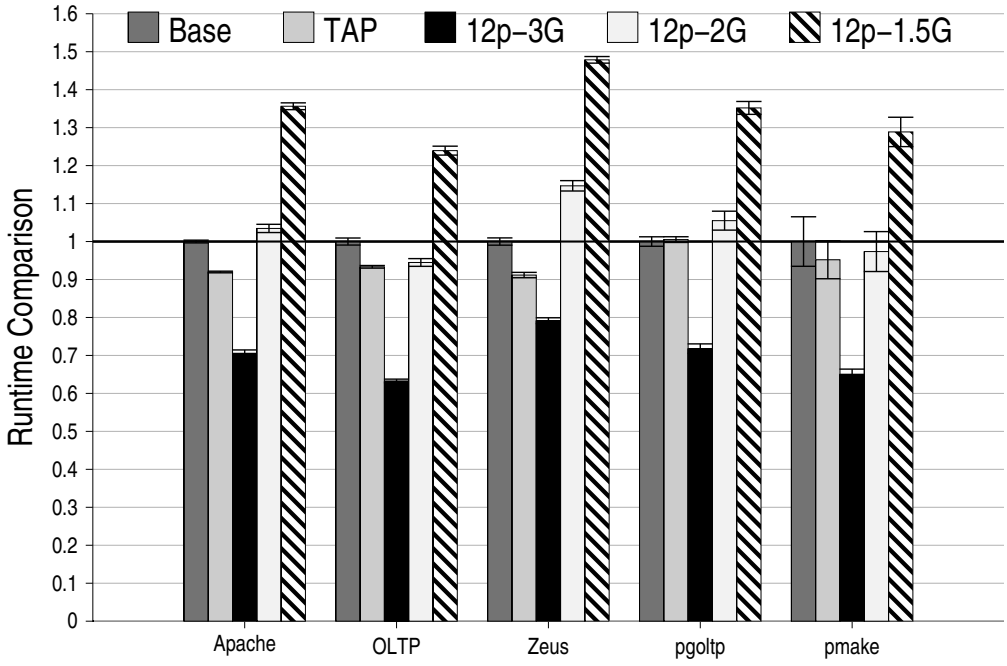


Figure 6.30: Runtime comparison with 12 VCPUs exposed to the OS. From left to right, five bars in each benchmark shows the results for the baseline 8 core system, TAP, and several 12 core systems operating at different frequency levels indicated in their respective labels.

simulations to make the best effort for alleviating the inherent challenges of comparing two different application checkpoints (See Section 5.1.1).

6.6.2.1 Runtime

Figure 6.30 presents the runtime comparison of the baseline and TAP, with the 12 core system that use the OPMS micro-architecture, but exploits application concurrency by simultaneously executing 12 application threads. Three different configurations of this 12 core systems are shown that vary in the frequency scaling applied on them. From left to right, five bars in each benchmark shows the result for the baseline system, TAP, 12 core system with the same frequency of the baseline/TAP (3 GHz), 12 core system operating at 2GHz and 1.5GHz, respectively.

By observing the bar showing the 12 core without frequency scaling (12p-100%), the figure demonstrates the overall good scaling property seen in these benchmarks. The ideal scaling is 67% reduction in runtime due to the use of 12 cores instead of 8. OLTP and pmake achieve super-linear scaling, showing runtime reductions of 63% and 65%, respectively. On the other hand, Zeus shows sub-par scaling of 79%, while Apache and pgoltp show 70% and 71%, respectively. Except Zeus, the execution times in these benchmarks are within 5% of the ideal scaling behavior.

Figure 6.30 also demonstrates the impact of frequency scaling on runtime for these workloads. Reducing the frequency dampens the pipeline throughput and increases on-chip latencies, severely affecting almost all the workloads. Although several of these benchmarks show large stalls from memory references, most of these stalls are from on-chip latencies rather than the main memory latency. Therefore, these benchmarks show marginal benefits from the stability of main memory latency across different on-chip frequencies. Comparing the results between the 12 core systems, Zeus shows runtime increases of 45% and 86% when operating at 67% and 50% baseline frequency, respectively.

6.6.2.2 Power

Figure 6.31 shows the power consumptions of the different multicore configurations shown in Figure 6.30. Several observations can be made from this figure. First, the figure shows that the power consumption from the 12-core multicore goes up substantially, when no frequency scaling is applied. Apache and OLTP show increases of 60% and 55%, respectively, more than the 50% addition of core resources. For both these workloads, the relative rise in their dynamic power is even higher: 67% in Apache and 58% in OLTP. Both these workloads exhibit higher IPC in their 12-core configurations compared to the baseline eight core system, causing a disproportionate rise in their dynamic power. Despite using identical application configurations, and substantially longer

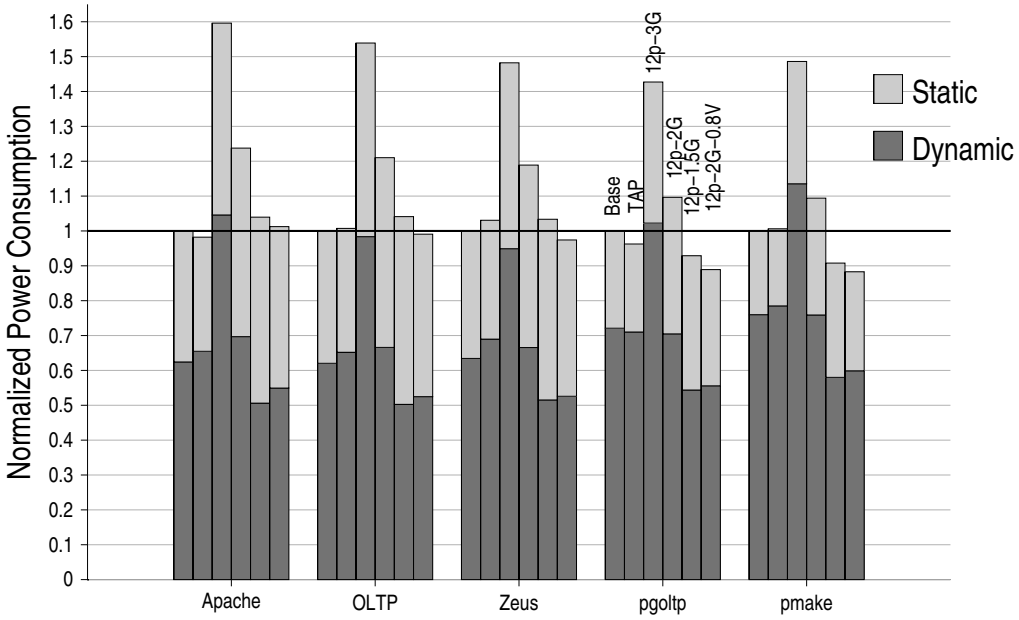


Figure 6.31: Normalized power consumption with 12-core Multicores. From left to right, six bars in each benchmark shows the results for the baseline 8 core system, TAP, and several 12 core systems operating at different frequency levels indicated in their respective labels. The rightmost bar shows the result for a voltage scaled (0.8V) 12 core system. Each bar is also broken down to show the contribution from the dynamic power (bottom), and the static power (top).

simulation runs to avoid transient effects, the difference in system configuration (8 cores versus 12 cores) causes this change in workload behavior.

Second, the static power in all the 12-core systems are substantially higher than the OPMS, despite employing the identical micro-architecture. This is an expected result from the 50% increase in on-chip cores that are actively engaged in computation simultaneously. Because of multiple leakage tolerant techniques implemented in large caches, the L2 cache does not contribute a major fraction of the chip leakage [94]. Therefore, a 25% reduction in the L2 size, seen in both the OPMS and 12-core configurations, provides only marginal savings in the chip leakage. In contrast, most of the low threshold devices, which contribute a large fraction of the leakage when active,

are concentrated on processing cores [102]. Consequently, these active cores contribute the bulk of the entire chip's leakage [94], and by increasing the active cores by 50% compared to OPMS, the 12-core systems show a substantial increase in static power from leakage.

Third, the figure shows that workloads with higher static power components require larger frequency reductions to achieve a power consumption level comparable to the baseline and TAP. For pmake, the 12-core must operate below 67% of the baseline frequency to achieve comparable power, while the corresponding requirement in Apache is 50% of the baseline frequency. As discussed previously, frequency scaling does not affect static power in the absence of voltage scaling (see Section 2.3.1). Therefore, increasing the number of simultaneously active cores reduces the dynamic power budget of each core–by more than the fractional increase in cores.

Fourth, the rightmost bar indicates that if voltage scaling was applicable, the power consumption of the 12 core system operating at 2 GHz would become comparable to the baseline across all the benchmarks.

6.6.2.3 Energy

Although the power consumption in the 12-core multicore system increased substantially, the reduction in execution time is expected to bring the total energy expenditure at a comparable level with the baseline system. Figure 6.32 shows the comparison of energy consumption. Compared to the baseline system with 8 cores, the 12-core multicore shows an increase in total energy in several cases (without frequency scaling). For example, Apache and Zeus show increases of 13.1% and 17.5%, respectively. In contrast, OLTP and pmake demonstrates improvements in energy by 2%. Even then, the total energy expenditure from TAP is lower than that of the 12-core system in all workloads.

Frequency scaling adversely affects the energy expenditure for all the 12-core configurations.

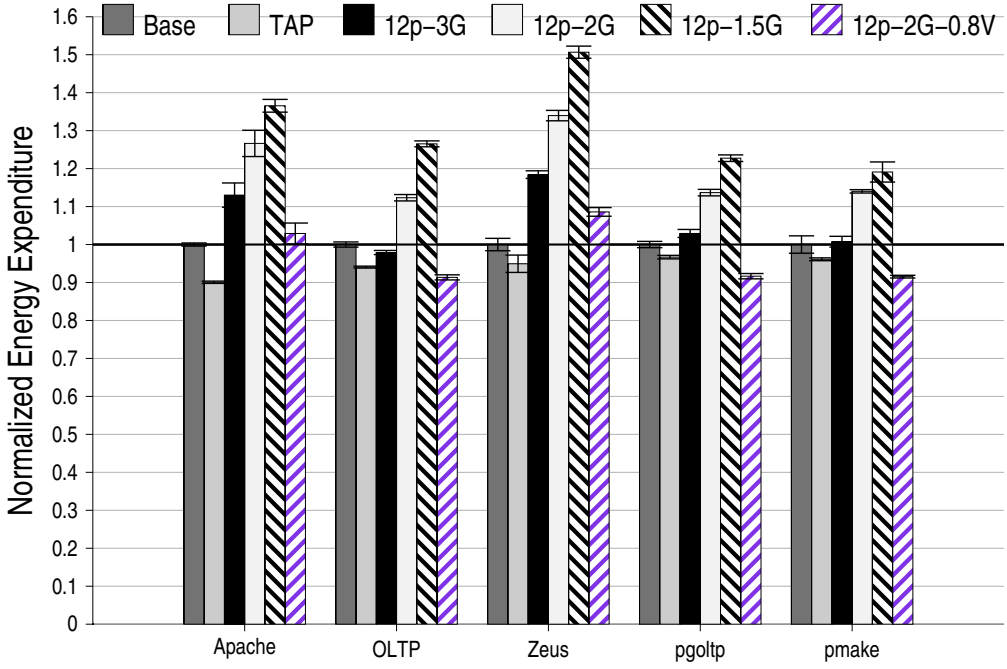


Figure 6.32: Energy expenditure comparison with 12-core Multicores. From left to right, six bars in each benchmark shows the results for the baseline 8 core system, TAP, and several 12 core systems operating at different frequency levels indicated in their respective labels. The rightmost bar shows the result for a voltage scaled (0.8V) 12 core system..

At 50% frequency scaling, Zeus shows the maximum relative increase in energy expenditure of 48%, due to the large increase in runtime combined with similar power consumption of the baseline 8-core system. In contrast, pmake shows an increase of 17%, as at this frequency, the power consumption in pmake is substantially lower than the baseline system (Figure 6.31). The rightmost bar demonstrates that applying voltage scaling can reduce the energy consumption substantially.

6.6.2.4 Energy-Delay Product

Finally, Figure 6.33 shows the comparison of energy efficiency—expressed in terms of the energy-delay product metric—across the baseline system (8-core), TAP, and several 12-core multicore

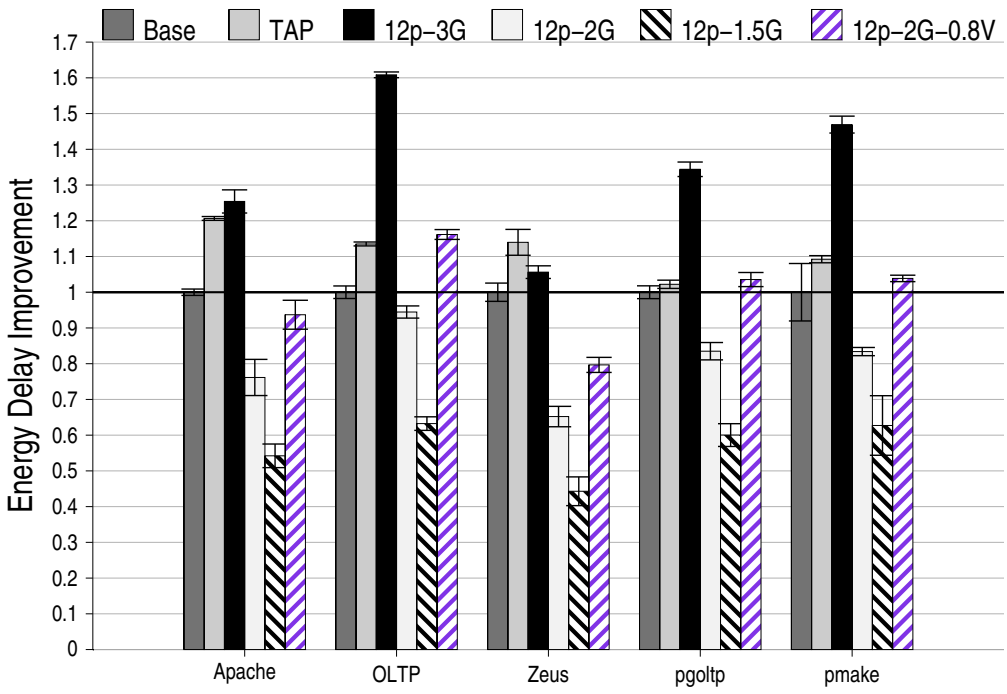


Figure 6.33: Energy delay comparison 12-core multicore configurations. From left to right, six bars in each benchmark shows the results for the baseline 8 core system, TAP, and several 12 core systems operating at different frequency levels indicated in their respective labels. The rightmost bar shows the result for a voltage scaled (0.8V) 12 core system..

configurations that vary in the extent of frequency scaling applied on them.

First, the figure shows that across all benchmarks except Zeus, the 12-core system operating at the full frequency of the baseline and TAP, achieves the maximum improvement in the energy delay product. OLTP and pmake demonstrate improvements of 61% and 58%, respectively. Both these benchmarks achieve more than linear reduction in execution time from scaling (i.e., executing 12 concurrent application threads instead of 8 in the baseline system), along with a savings in total energy expenditure, leading to a dramatic improvement in the energy-delay product. However, this improvement in energy efficiency comes at the expense of a substantially higher TDP. For example, OLTP and pmake show 53% and 50% increase in the average power consumption,

respectively (see Figure 6.31). Zeus shows the least improvement in energy delay, as its improvement in runtime from using the 12-core system is lowest among all benchmark (see Figure 6.30), while also showing a modest increase in the energy expenditure (Figure 6.32).

Second, when considering multicore configurations that are within 10% of the power envelope for the baseline system in respective workloads, 12-core systems show substantial degradations in the energy efficiency. From Figure 6.31, the 12-core system must operate at 50% of the baseline frequency when executing Apache, Zeus and OLTP, while for pgoltp and pmake it needs to operate at 67% of the baseline frequency to restrict within 10% of the baseline power envelope. Within this category of multicores (at the same power budget), TAP achieves maximum improvement in the energy delay product, across all benchmark. Although pmake and pgoltp requires smaller reduction in frequency, relatively higher energy expenditure and near linear increase in execution time (Figure 6.30) cause the degradation in the energy efficiency. In contrast, memory-bound workloads like Apache require greater reduction in frequency to offset their larger static power component. Consequently, they show large increases in energy expenditure from their higher runtimes as shown in Figure 6.30, and lead to the degradation of energy efficiency.

Finally, the rightmost bar in the figure demonstrates that voltage scaling can improve the energy-delay product of the system, while simultaneously operating at a comparable power budget to the baseline. However, the OPMS still achieves substantially better improvements in Apache and Zeus, while delivering comparable results in the other benchmarks. Most importantly, this competitive advantage in OPMS is achieved without resorting to voltage scaling, which will be impractical in future technology.

In summary, these results demonstrate that the benefits derived from the OPMS is a combined outcome of the locality enhancement and the OPMS design paradigm, instead of the specific micro-architecture it employs.

6.6.3 In-order cores

This section evaluates a completely different multicore design point: multicore built with in-order processing cores. Few industry vendors have chosen to implement in-order cores for server platforms, and therefore it is important to evaluate the OPMS design in such a context. Fundamentally, in-order core cannot hide the latency of data references, due to the sequential instruction issue. Consequently, both instruction and data misses affect the pipeline performance comparably.

Figure 6.34 shows the speedups achieved in multicore built with in-order processing cores. The configuration of the in-order core used in this study is described in Section 5.3.1. From left to right, three bars show the result for the baseline multicore, TAP, and SAP, respectively.

The figure shows that TAP is able achieve speedups comparable to those presented earlier in Section 6.4.1, even in multicores with in-order cores. The speedups are within 1% to the earlier result, except in Apache and Zeus. Apache sees an increase in speedup from 10% to 16% in this figure. In addition to the improvement in instruction cache performances, Apache achieves substantial performance gains in data cache performances as well. This data cache performance improvement positively impact the pipeline throughput in the in-order cores, resulting in the differential gain in performance. In contrast, the large data cache stalls in Zeus, and the lack of improvement in data cache performance, mask the benefits of instruction throughput improvement, diminishing the performance gains seen in Zeus (from 10% to 3%).

In contrast, the degradation in data cache performances seen in SAP, substantially impacts the performance with in-order cores. For example, OLTP observes an improvement of 2%, compared to 10% improvement seen with out-of-order cores. While SAP achieved a 4% improvement in pmake, with in-order cores, it performs worse than the baseline system (1%). In summary, TAP continues to be useful within the context of in-order cores, but SAP loses most of the performance gains seen due to the degradation in data cache performances.

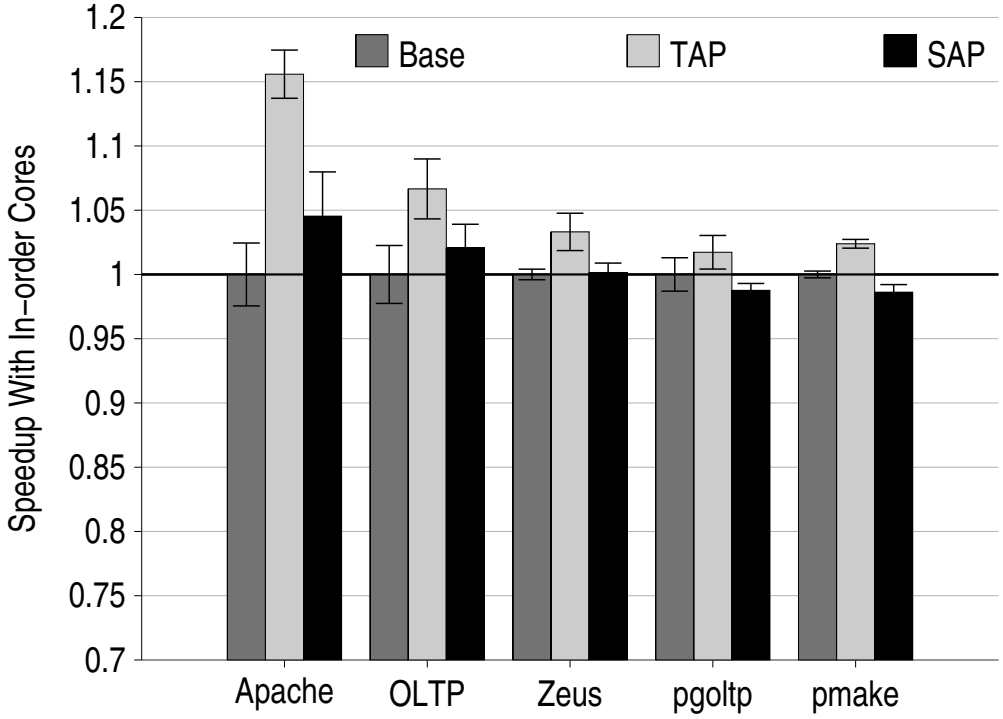


Figure 6.34: Speedup achieved with In-order Cores. From left to right, three bars for each benchmark show the results for the baseline, TAP, and SAP, respectively.

6.6.4 L1 Cache Sizes

This section investigates the impact of L1 cache sizes on the performance impact of CSP. L1 caches are closely coupled with the processing cores, and the cache access time play a key role in determining the clock frequency [42]. Consequently, the sizes of L1 caches grow slowly, despite the abundance of transistor resources. The size of the caches modeled in this study is fairly common among out-of-order processor core implementations, most notably Alpha 21264 [60]. However, two different L1 sizes are evaluated in this section: 32KB and 128KB. The shared L2 remains unchanged in all these experiments.

Figure 6.35 shows the speedup achieved when using L1 sizes of 32KB. Both L1 Instruction

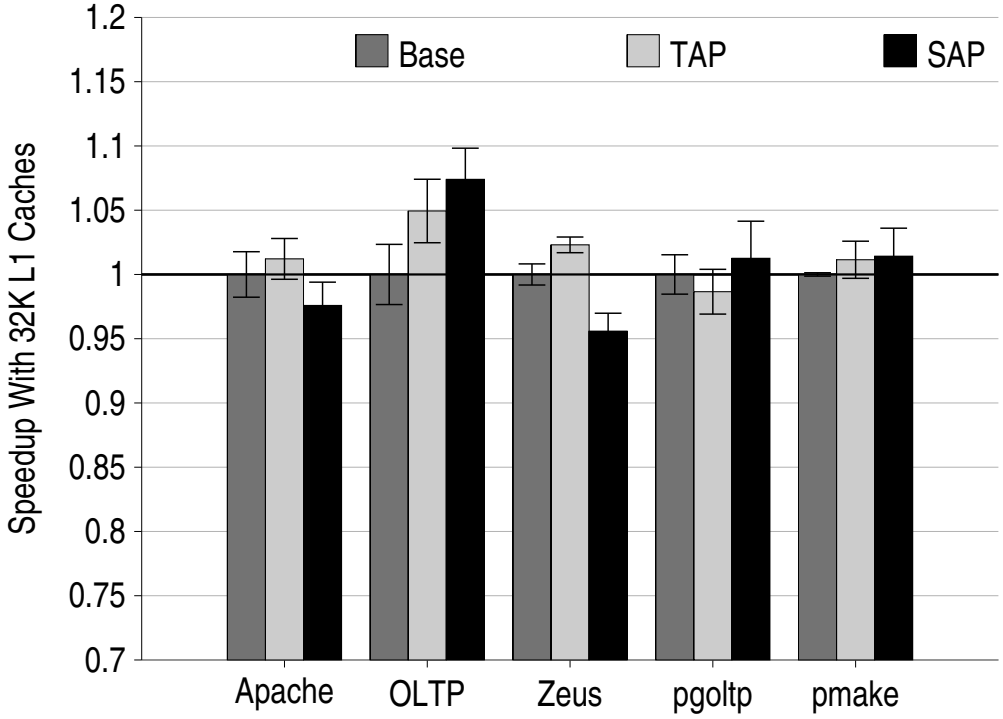


Figure 6.35: Speedup with 32KB L1 Caches. From left to right, three bars for each benchmark show the results for the baseline, TAP, and SAP, respectively.

cache, and the L1 Data cache are 32KB, and 8-way associative in this experiment. Across several benchmarks, the speedups achieved using this cache configuration is lower than that seen with the 64K caches. The smaller cache reduces the amount of predictive state retained in inactive cores, which leads to lesser benefit from the dynamic specialization through CSP. TAP, for example, yields only 2% speedup in Apache compared to 10% seen earlier. SAP, on the other hand, leads to performance degradation in Apache and Zeus, while achieving lower speedups in the rest.

Figure 6.36 shows the speedup achieved with 128K L1 caches. TAP achieves 18% speedup in Apache, substantially more than the 10% speedup achieved using 64K caches. For Apache, TAP improves both instruction and data cache performances, and this improvement is relatively larger

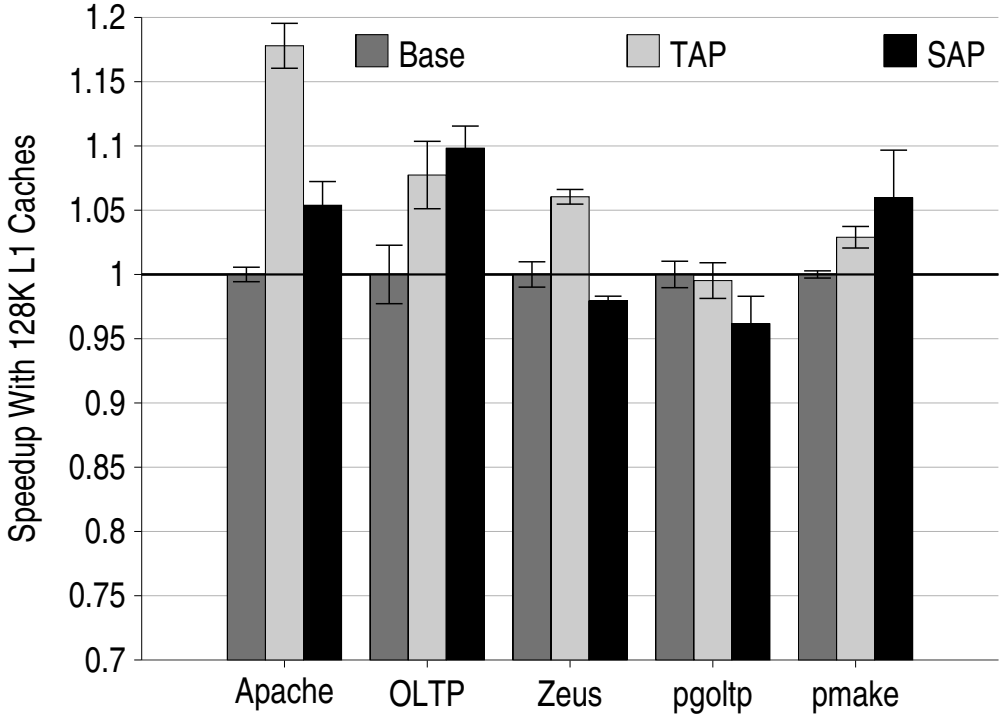


Figure 6.36: Speedup with 128KB L1 Caches. From left to right, three bars for each benchmark show the results for the baseline, TAP, and SAP, respectively.

particularly for instruction caches. As the instruction reference latency is a primary bottleneck in this benchmark, using larger L1 Instruction caches enables more state to be retained, leading to more improvement. OLTP performs similar to the results in 64K cache. However, Zeus shows lower speedup of 6%, compared to 10%. The performance of data caches has greater impact on Zeus runtime, and using 128K worsens the relative performance of the data cache in TAP and SAP. Likewise, SAP performs slightly better in three benchmarks: Apache, OLTP and pmake. But due to the worse data performance, SAP exhibits a degradation in the runtime for pgoltp and Zeus.

6.7 Related Work

A significant amount of past research targets voltage-frequency scaling as an effective means for delivering power-performance benefits. For multicore designs, recent studies demonstrate the potential for achieving a superior energy-delay product, while restricting the performance loss [52, 68]. Much of this work relies on device-level power reduction through voltage scaling, which is unlikely to continue in coming technology generations (see Section 2.3.1). Frequency scaling, in the absence of voltage scaling, yields a worse power performance trade-off for a broad class of multithreaded applications. Instead, this dissertation demonstrates aggressive coarse-grain techniques for SAF reduction in a power-constrained microprocessor, and illustrate how such designs can achieve improvements in the energy-delay product.

Several previous proposals have investigated the power-performance implications of varying resource provisioning for caches and cores using multi-programmed workloads (e.g., [68]) and multithreaded scientific workloads (e.g., [77]). These studies assume a traditional multicore design where the pool of processing cores are simultaneously active. In contrast, the OPMS design paradigm advocates an alternative multicore design where only a subset of the available cores participate in active computation. This section rigorously evaluated the performance and energy efficiency of this design in the context of multithreaded server class applications.

Many past proposals advocate activity migration to mitigate thermal hot spots [37, 69, 97]. While adopting different mechanisms and underlying microarchitectures, all of them attempt to take rectifying action after a certain threshold temperature is sensed, indicating a forthcoming thermal emergency. Rather, an OPMS naturally offers a spatial slack in computational resources, and CSP exploited this slack by continually moving the computation around to avoid exercising one particular resource for a long enough duration. Powell, et al., exploit spatial slack in a conventional multicore by modifying the system software, and primarily focus on the power density problem

[37]. However, they do not investigate the implications of TDP or SAF in their design, neither do they address complex multithreaded workloads, and their solution is not transparent to the OS.

Heterogeneous multicores have been proposed to improve the energy efficiency of various applications, by employing different processing cores that vary in ILP extraction and energy efficiency [63, 79]. This dissertation focuses on multicores built from homogeneous processing cores that are growing in prominence due to their modular and cost effective designs. However, the OPMS design paradigm is also applicable to heterogeneous multicores, especially by allowing designers to aggressively integrate many more cores than would be possible in a conventional multicore system.

6.8 Summary

This chapter presented quantitative results based on a rigorous evaluation carried out for understanding the major trade-offs in an OPMS design that employs Computation Spreading (CSP). Using CSP on an OPMS, significantly improved the performance of the multithreaded server applications evaluated in this dissertation, demonstrating 10% speedups in Apache and Zeus using TAP, respectively. A major component of these speedups came from the dramatic improvement in L1 instruction misses (up to 44% in Zeus), while the improvements in the branch mis-prediction also played an important role. Despite frequent computation transfers, the data latencies remain comparable to the baseline using TAP, thereby limiting its relative impact on performance as compared to the baseline. In addition, this chapter demonstrated that using an OPMS substantially reduces the contention overhead of CSP, and the superior bandwidth of on-chip communication restricted the latency overhead from the CT within manageable limit, eventually realizing performance gains, even after accounting for these overheads.

In the second part, this chapter demonstrated the potential of an OPMS design in achieving

energy efficiency. Despite using more processing cores, the chapter showed that the selective assignment of computation in an OPMS led to a partial reduction in the number of cores engaged in computation on average. Along with the runtime improvements, this reduction in core activity, achieved an overall savings in the core energy, while the cache energy was improved by better utilization of private caches, and saving accesses into the shared cache. Overall, this chapter demonstrated improvements of 4–20% in energy efficiency in an OPMS, while remaining within the power envelope of the baseline system.

In the final part, this chapter presented results from several sensitivity analysis to understand the relative contributions of CSP, the OPMS design paradigm, and various system level configuration. By comparatively evaluating CSP on a conventional multicore, and a 12-core system—identical configuration as the OPMS micro-architecture—in the conventional way (i.e., exposing all the cores to the OS for exploiting application level concurrency), this chapter demonstrated that the energy efficiency improvements are largely derived from the OPMS design paradigm itself, rather than exclusively from the locality enhancement of CSP, and the specific micro-architecture employed.

Chapter 7

Conclusions

Today, the design of microprocessors is at a juncture of intriguing design choices that lie ahead. While the emergence of multicore systems is making a large impact on the entire computer industry, the adoption of energy efficiency requirements in general purpose microprocessors is poised to radically alter the shape of future microprocessors.

In this context, this dissertation presented an in-depth exploration of a novel multicore design paradigm that can substantially impact future microprocessor designs. Section 7.1 summarizes the contributions of this dissertation. Section 7.2 presents possible future research directions stemming out of this work.

7.1 Contributions

The main contributions of this dissertation were: the definition and application of an abstract model to represent the power constraint in microprocessors (Chapter 2), the OPMS framework and its implementation (Chapter 3), a classic application of the OPMS design via Computation Spreading (Chapter 4) and its rigorous evaluation (Chapter 6). The key results from each of them

are outlined next.

7.1.1 Simultaneously Active Fraction (SAF)

Despite the increasing demand of energy efficiency in microprocessor designs, accurate estimation of the power consumption in the early phase of their designs remains impractical. Instead, this dissertation presented a model to capture the first order impact of the power constraint, and demonstrated its utility in guiding multicore designs. This work defined *Simultaneously Active Fraction (SAF)* to denote the largest allowable fraction of on-chip devices that can remain active simultaneously without violating the power envelope of the chip. Due to the continued exponential growth of on-chip devices, coupled with the limited improvement in energy efficiency from device scaling, this dissertation demonstrated a monotonically decreasing SAF in future technology generations.

To meet their power constraints, future multicores must incorporate novel techniques for SAF reduction, and investigate how a reduction of SAF impacts their high-level functional designs early in the design phase. This work demonstrated two specific uses of SAF considerations using analytic models in multicore design. First, this work extended the previous work by Hill and Marty [46], and incorporated SAF for explicitly modeling power constraints in various multicore designs. Interestingly, this work found that with the increasing power constraint, modeled using different levels of SAF, the asymmetric multicore approached the performance of the dynamic multicore. At exactly $SAF = 1/2$, and lower, the asymmetric multicore performed as good as the dynamic multicore, altering one of the major conclusions in the previous work [46].

In the second application, this work investigated the use of over-provisioning as a fall out from the chips with decreasing SAF. The asymmetric multicore was able to better adapt to the increasing power constraints, as it over-provisioned the core resources, and dynamically distributed

its power budget among on-chip cores. With the abundance of an area budget, this general strategy appeared promising, and this dissertation presented an analytic model to investigate its utility in multithreaded applications. Based on the characteristics of the multithreaded server applications, the work showed the superior ability of over-provisioned multicores to dynamically adapt to the workload characteristics. This early promise of improving performance via over-provisioning core resources in multicores led to the in-depth analysis of such systems in the subsequent chapters.

7.1.2 Over-provisioned Multicore System (OPMS)

This work defined the broad framework for an Over-provisioned Multicore System (OPMS), discussed its functional requirements, and presented an elaborate discussion on cost-benefit analysis. By design, an OPMS integrates more processing cores on its die than allowed by the chip Thermal Design Power. In this framework, a subset of the on-chip cores carry out the collective computation, while the rest of the cores are maintained in an inactive state. Fundamentally, this design paradigm captures the shifting trends in the underlying semiconductor technology, where integrating cores on a single die comes with a significantly lower marginal cost than the engineering and economic cost to power them up simultaneously.

Two major functional requirements in this framework are: interfacing with the system software, and power efficient management of inactive core resources. To satisfy the first requirement, this dissertation implemented a solution based on virtualization techniques. The OPMS framework is supported by a virtualizing the processor cores, and providing a mechanism to transfer computation between on-chip cores. Both these are implemented in a firmware component, transparent to the software. The second requirement is met by incorporating aggressive leakage tolerant techniques so as to reduce the power consumption from the inactive cores. The core logic structures are augmented with sleep transistors based on MTCMOS, which can reduce leakage in inactive cores

within 2% of the corresponding leakage in the active mode. The private caches maintain their predictive states in a drowsy state during inactive periods, consuming 20% static power compared to that in the active mode.

The OPMS design paradigm facilitates a tremendous flexibility in assigning computation on cores by integrating more processing cores, and dynamically distributing the power budget among on-chip cores. This dissertation proposed and evaluated Computation Spreading that exploited this unique multicore framework.

7.1.3 Computation Spreading

In the canonical model of assigning computation from multithreaded applications onto multicore systems, an entire software thread is assigned to a single processor for execution. This assignment policy leads to extensive commonality in the code executed on different on-chip cores. This dissertation found that 48–70% instructions blocks were accessed by all the on-chip cores in server applications. The extensive code reuse severely degraded the effectiveness of micro-architecture structures that are designed to exploit instruction stream properties.

To mitigate this problem, and exploit the OPMS framework, this dissertation proposed Computation Spreading (CSP). Conceptually, CSP aimed to collocate similar computation fragments from different threads on the same core, while distributing the dissimilar computation fragments from the same thread across multiple cores. Each core in the OPMS became dynamically and temporally specialized for executing a set of specific computation fragments by retaining the states (such as instruction cache contents and branch predictor entries) necessary to perform each computation efficiently.

Guided by the general design considerations of CSP, this dissertation presented a specific application of CSP where it separated the user and OS computation from each thread. Different sets of

on-chip cores were provisioned to carry out these computations. The locality of memory references were improved, as this distribution avoided destructive interference from OS and user computation, while enhancing the constructive interference between multiple computation fragments belonging to the same class.

A rigorous evaluation of this application was presented next. This dissertation found that CSP dramatically reduced L1 instruction misses by 20–51% in server benchmarks, improving the average instruction latency observed by 24–57%. Despite frequent computation transfers, the data latencies remained comparable to the baseline in the best assignment policy. Overall, such locality enhancement in memory references via selective use of on-chip cores led to 4–10% speedup improvement, along with 9–21% improvement in energy efficiency. A detailed sensitivity analysis further demonstrated that this improvement was a combined result of the OPMS design paradigm and locality enhancement from CSP, rather than exclusively from CSP or the particular OPMS micro-architecture.

7.2 Future Directions

The work presented in this dissertation is an initial foray into an area of immense potential for future multicore designs. The general paradigm of multicores where computation resources are dynamically employed can impact a range of applications, while providing new avenues to meet design objectives. This section presents few possible research directions that can benefit from the insights gained out of this dissertation.

7.2.1 OPMS Management

The OPMS framework is a promising execution platform for delivering energy efficiency without demanding additional power budget. Different forms of hardware accelerators, and other modes of achieving dynamic specialization from structurally identical cores as demonstrated by CSP, will likely be developed in the near future. Managing and effectively exploiting these growing computation resources will be the two key challenges in this domain, as the conventional policy of using the OS to manage computation resources at the granularity of a processing core will not scale.

This work demonstrated one example for managing these resources in an OPMS via a lightweight virtual machine layer. While this technique is indeed a viable mechanism, other alternatives exist with different engineering trade-offs. Exposing an abstract notion of this OPMS to the application layer through the runtime system appears promising, since it can facilitate both management and effective use of the underlying computation resources. For example, application developers can provide hints to aid in the appropriate micro-architecture assignment of computation fragments, as well as restructure their algorithm and data-structures to facilitate moving computation to data, similar to CSP.

7.2.2 CSP

This dissertation presented a specific application of CSP in the context of multithreaded applications. However, CSP is applicable to a broader range of applications without just relying on separating OS and User computation. Many domains of applications spend negligible fractions of their execution time in the OS (e.g., SPEC CPU benchmarks [114]). For such applications, CSP must look for more sophisticated means for partitioning computation fragments. Nevertheless, the general design considerations mentioned in Section 4.3 will continue to be useful in developing

new applications of CSP.

Beyond the server class workloads, data stalls are the major impediments to performance in a vast majority of benchmarks, with a substantially smaller instruction stall component [44]. The assignment of computation fragments onto multiple cores must then consider both the code path and the section of data footprint that a particular fragment operates on. Moreover, the true data dependency in the program must be separated from the dependencies in the stack operations, as many of the latter are false dependencies. Thus, when the computation moves around on different cores, the hardware/software must ensure that stack dependencies are limited to avoid unnecessary cache-to-cache transfers.

Conceptually, CSP improves the utilization of private resources by carefully mapping computation fragments onto appropriate cores. This general policy will become increasingly important with the proliferation of compute resources in future multicores, causing a corresponding growth in aggregate private resources. Poor utilization of these resources will exacerbate the impact on performance, as increasing wire delays in future technology will invariably drive up the access latency of large shared resources.

CSP alters the specific instruction streams that exercise a given micro-architecture component in a modern processor core. This fundamental property in CSP can dramatically improve the effectiveness of micro-architecture components, and reduce the burden of design complexity necessary for handling a wide variance in instruction stream properties. This dissertation demonstrated the effect on branch predictors and caches, but the promise of CSP extends well beyond those two structures, influencing the fundamental design considerations in future microprocessors.

References

[1] Agarwal A., Mukhopadhyay S., Raychowdhury A., Roy K., and C.H Kim. Leakage power analysis and reduction for nanoscale circuits. *IEEE Micro*, 26(2), 2006.

[2] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions of Computer Systems*, 6(4):393–431, 1988.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th Annual Very Large Databases (VLDB)*, 1999.

[4] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[5] Monther Aldwairi, Thomas Conte, and Paul Franzon. Configurable string matching hardware for speeding up intrusion detection. *SIGARCH Computer Architecture News*, 33(1), 2005.

[6] Gene M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference*, 1967.

[7] Hari Ananthan and Kaushik Roy. A fully physical model for leakage distribution under process variations in nanoscale double-gate cmos. In *Proceedings of the 43rd Annual Conference on Design Automation*, 2006.

[8] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[9] Krste Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, Dec 2006.

[10] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[11] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 International Conference on Measurement and Modeling of Computer Systems*, 1998.

[12] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer architecture*, 1998.

[13] Bradford M. Beckmann. *Managing Wire Delay in Chip Multiprocessor Caches*. PhD thesis, University of Wisconsin-Madison, 2006.

[14] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, 2004.

[15] Sekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4), 1999.

[16] Shekar Borkar. On the future of on-chip interconnection architectures (NOCs and multicores): Keynote. In *Proceedings of the 2007 International Symposium of Low Power Electronics and Design (ISLPED)*, 2007.

[17] Shekhar Borkar. Microarchitecture and design challenges for gigascale integration: Keynote. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, 2004.

[18] Shekhar Borkar et al. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Annual Conference on Design Automation*, 2003.

[19] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield. New methodology for early-stage, microarchitecture-level power performance analysis of microprocessors. *IBM Journal of Research and Development*, 47(5/6), 2003.

[20] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

[21] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional Computing Series, 1997.

[22] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proceedings of the 33th Annual International Symposium on Microarchitecture (MICRO)*, 2000.

[23] Lei Chai, Qi Gao, and Dhabaleswar K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *International Symposium on Cluster Computing and the Grid*, 2007.

[24] Koushik Chakraborty, Philip Wells, and Guri Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[25] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. A case for over-provisioned multicore systems. Technical Report TR1607, University of Wisconsin, Madison, Aug 2007.

[26] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[27] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS)*, 2007.

[28] Nathan Clark, Amir Hormati, and Scott Mahlke. Veal: Virtualized execution accelerator for loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[29] Barry Dennington. Low power design - from technology challenges to great products: Keynote. In *Proceedings of the 2006 International Symposium of Low Power Electronics and Design (ISLPED)*, 2006.

[30] Steven Dropsho, Volkan Kursun, David H. Albonesi, Sandhya Dwarkadas, and Eby G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*, 2002.

[31] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[32] Alexandra Fedorova, David Vengerov, and Daniel Doucette. Operating system scheduling on heterogeneous core systems. In *Proceedings of 2007 Operating System Support for Heterogeneous Multicore Architectures*, 2007.

[33] Krisztin Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

[34] Joshua Fredrich et al. Design of the Power6 microprocessor. In *Proceedings of the 2007 International Solid-State Circuits Conference (ISSCC)*, 2007.

[35] Kanad Ghose, Pradip Bose, and Kevin Skadron. Computer architecture research directions: Panel discussion on low power design and temperature management. Workshop in HPCA, 2007.

[36] Nicolas Gloy, Cliff Young, J. Bradley Chen, and Michael D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[37] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[38] Brian Greskamp and Josep Torrellas. Paceline: Improving single-thread performance in nanoscale cmps through core overclocking. In *Proceedings of the 16th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[39] P. Guerdoux-Jamet and D. Lavenier. Samba: hardware accelerator for biological sequence comparison. *Bioinformatics*, 13(6), 1997.

[40] Richard Hankins, Trung Diep, Murali Annavaram, Brian Hirano, Harald Eri, Hubert Nueckel, and John P. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[41] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Third Annual Conference on Innovative Data Systems Research*, 2007.

[42] Stavros Harizopoulos and Anastassia Ailamaki. STEPS towards cache-resident transaction processing. In *Proceedings of the 30th International Conference on Very Large Databases*, 2004.

[43] Raymond Heald et al. A third-generation sparc v9 64-b microprocessor. *IEEE Journal of Solid-State Circuits*, 35(11), 2000.

[44] John Henning. Performance counters and development of spec cpu2006. *SIGARCH Computer Architecture News*, 35(1), 2007.

[45] John Henning. Spec cpu2006 memory footprint. *SIGARCH Computer Architecture News*, 35(1), 2007.

[46] Mark D. Hill and Michael Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(8), 2008.

[47] Mark Horowitz, Elad Alon, Dinesh Patil, Samuel Naffziger, Rajesh Kumar, and Kerry Bernstein. Scaling, power and the future of cmos. In *IEEE International Electron Devices Meeting*, 2003.

[48] Mark Horowitz, Donald Stark, and Elad Alon. Digital circuit design trends. *IEEE Journal of Solid-State Circuits*, 43(4), 2008.

[49] 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 *Proceedings of the 15th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[50] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium of Low Power Electronics and Design (ISLPED)*, 2004.

[51] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future CMPs. In *Proceedings of the 10th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[52] 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 the 39th Annual International Symposium on Microarchitecture (MICRO)*, 2006.

[53] ITRS. Executive summary. *International Technology Roadmap For Semiconductors*, 2005.

[54] ITRS. Process integration, devices and structures. *Intnt'l Technology Roadmap For Semiconductors*, 2005.

[55] ITRS. Executive summary. *International Technology Roadmap For Semiconductors*, 2007.

[56] ITRS. Systems drivers. *International Technology Roadmap For Semiconductors*, 2007.

[57] ITRS Working Group Models: Process Integration Chapter. MASTAR: Model for assessment of CMOS technologies and roadmap. http://www.itrs.net/models.html.

[58] Hans Jacobson et al. Stretching the limits of clock-gating efficiency in server-class processors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[59] Aamer Jaleel, Matthew Mattina, and Bruce Jacob. Last level cache (LLC) performance of data mining workloads on a CMP a case study of parallel bioinformatics workloads. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[60] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2), 1999.

[61] 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 *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[62] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003.

[63] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[64] Patrik Larsson. di/dt noise in cmos integrated circuits. *Analog Integrated Circuits Signal Process.*, 14(1-2), 1997.

[65] James R. Larus and Michael Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track USENIX Annual Technical Conference*, 2002.

[66] Jian Li and Jose F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[67] Tao Li, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, and Juan Rubio. Understanding and improving operating system effects in control flow prediction. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[68] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraint. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[69] Weiping Liao, Lei He, and Kevin Lepak. Temperature and supply voltage aware performance and power modeling at the microarchitecture level. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7), 2005.

[70] D. Lilja, F. Marcovitz, and P. C. Yew. Memory referencing behavior and a cache performance metric in a shared memory multiprocessor. Technical Report CSRD-836, University of Illinois, Urbana-Champaign, Dec 1988.

[71] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[72] Peter Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[73] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2002.

[74] Richard McDougal and Jim Mauro. *Solaris Internals(TM): Solaris 10 and OpenSolaris Kernel Architecture*. Prentice Hall, 2006.

[75] Francisco Javier Mesa-Martinez, Joseph Nayfach-Battilana, and Jose Renau. Power model validation through thermal measurements. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[76] Nihar R. Mohapatra, Madhav P. Desai, Siva G. Narendra, and V. Ramgopal Rao. Modeling of parasitic capacitances in deep submicrometer conventional and high-k dielectric mos transistors. *IEEE Transactions on Electron Devices*, 50(4), 2003.

[77] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: A power/performance/thermal view. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS)*, 2006.

[78] Gordon E. Moore. Progress in digital electronics. In *Proceedings of the 1975 International Electron Devices Meetings*, 1975.

[79] Tomer Y. Morad et al. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. In *IEEE Computer Architecture Letters*, 2006.

[80] Trevor Mudge. Power: a first-class architectural design constraint. *IEEE Computer*, 34(4), April 2001.

[81] Siva G. Narendra and Anantha Chandrakasan (Editors). *Leakage in Nanometer CMOS Technologies*. Springer, 2006.

[82] Umesh Gajanan Nawathe, Mahmudul Hassan, King C. Yen, Ashok Kumar, Aparna Ramachandran, and David Greenhill. Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1), 2008.

[83] Open Source Development Labs. Database test suite. `http://osdldbt.sourceforge.net/`. Viewed 5/29/2007.

[84] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[85] Ehsan Pakbaznia, Farzan Fallah, and Massoud Pedram. Charge recycling in mtcmos circuits: concept and analysis. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, 2006.

[86] Ehsan Pakbaznia, Farzan Fallah, and Massoud Pedram. Charge recycling in MTCMOS circuits: concept and analysis. In *Proceedings of the 43rd Annual Conference on Design Automation*, 2006.

[87] Soyeon Park, Weihang Jiang, Yuanyuan Zhou, and Sarita Adve. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[88] PostgreSQL. PostgreSQL: the world's most advanced open source database. `http://www.postgresql.org/`.

[89] Michael D. Powell and T. N. Vijaykumar. Pipeline damping: a microarchitectural technique to reduce inductive noise in supply voltage. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[90] Alex Ramirez, Luiz Andre Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, 2001.

[91] P. Ranganathan and N Jouppi. Enterprise IT trends and implications on system architecture research. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[92] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 9th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[93] Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2), 2003.

[94] S. Rusu et al. A 65-nm dual-core multithreaded Xeon processor with 16-MB L3 cache. *IEEE Journal of Solid-State Circuits*, 2007.

[95] Nabeel Sakran, Marcelo Yuffe, Moty Mehalel, Jack Doweck, Ernest Knoll, and Avi Kovacs. The implementation of the 65nm dual-core 64b Merom processor. In *IEEE Solid-State Circuits Conference*, 2007.

[96] Xudong Shi, Feiqi Su, Jih kwon Peir, Ye Xia, and Zhen Yang. CMP cache performance projection: Accessibility vs. capacity. In *Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2006.

[97] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture: Extended discussion and results. Technical Report CS-2003-08, University of Virginia, 2003.

[98] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[99] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[100] Evan Speight, Hazim Shafi, Lixin Zhang, and Ram Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[101] Blaine Stackhousel, Brian Cherkauer, Mike Gowan, Paul Gronowski, and Chris Lyles. A 65nm 2-billion-transistor quad-core ltanium processor. In *Proceedings of the 2008 International Solid-State Circuits Conference (ISSCC)*, 2008.

[102] Benjamin Stolt, Yonatan Mittlefehldt, Sanjay Dubey, Gaurav Mittal, Mike Lee, Joshua Friedrich, and Eric Fluhr. Design and implementation of the power6 microprocessor. *IEEE Journal of Solid-State Circuits*, 43(1), 2008.

[103] Sun Microsystems. *UltraSPARC III Cu: User's Manual*, 2.2.1 edition, January 2004.

[104] Kunihro Suzuki. Parasitic capacitance of submicrometer mosfet's. *IEEE Transactions on Electron Devices*, 46(9), 1999.

[105] David Tarjan, Shyamkumar Thoziyoor, and Norman Jouppi. Cacti 4.0. Technical Report 20060606, HP Labs, 2006.

[106] Michael Bedford Taylor, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2), 2005.

[107] James E. Thorton. Parallel operation in the control data 6600. In *Proceedings of the Fall Joint Computer Conference*, 1964.

[108] Josep Torrellas, Anoop Gupta, and John Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the 5th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[109] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary. In *Proceedings of the 1993 International Conference on Measurement and Modeling of Computer Systems*, 1993.

[110] James W. Tschanz, Siva G. Narendra, Yibin Ye, Bradley A. Bloechel, Shekhar Borkar, and Vivek De. Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *IEEE Journal of Solid-State Circuits*, 38(11), 2003.

[111] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5), 2005.

[112] U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency. `http://www.energystar.gov/index.cfm?c=prod_development.server_efficienc%y_study`, Aug 2007.

[113] Carl A. Waldspurger. Memory resource management in vmware esx server. In *OSDI'02: Proceedings of the 2002 Operating System Design and Implementation*, 2002.

[114] Reinhold P. Weicker and John Henning. Subroutine profiling results for the cpu2006 benchmarks. *SIGARCH Computer Architecture News*, 35(1), 2007.

[115] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Adapting to intermittent faults in multicore systems. In *Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[116] Philip M. Wells and Gurindar S. Sohi. Serializing instruction in system-intensive workloads: Amdahl's law strikes again. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[117] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th Annual Symposium on Operating Systems Principles (SOSP)*, 2001.

[118] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2004.

[119] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[120] Ahmed Youssef, Mohab Anis, and Mohamed Elmasry. Dynamic standby prediction for leakage tolerant microprocessor functional units. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, 2006.

[121] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[122] Dakai Zhu, Rami Melhem, and Daniel Mosse. The effects of energy management on realiability in real-time embedded systems. 2004.