MODULAR ARCHITECTURES AND OPTIMIZATION TECHNIQUES FOR POWER AND RELIABILITY IN FUTURE MANY CORE MICROPROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Paula Petrica

January 2012

MODULAR ARCHITECTURES AND OPTIMIZATION TECHNIQUES FOR POWER AND RELIABILITY IN FUTURE MANY CORE MICROPROCESSORS

Paula Petrica, Ph.D.

Cornell University 2012

Power and reliability issues are expected to increase in future multicore systems with a higher degree of component integration. As the feature sizes of transistors continue to shrink, more resources can be incorporated in microprocessors to address a broader spectrum of different application requirements. However, power constraints will limit the amount of resources that can be powered on at any given time. Recent studies have shown that future multicore systems will be able to power on less than 80% of their transistors in the near future, and less than 50% in the long term. The most difficult challenge is deciding which transistors should be powered on at any given time to deliver high performance under strict power constraints. At the same time, device reliability issues - the proliferation of devices that will either be defective at manufacturing time or will fail in the field with usage - are projected to be exacerbated by the continued scaling of device sizes.

We present a modular, dynamically reconfigurable architecture as a promising unified solution to the problems of dark silicon (the inability to power all available computing resources) and reliability. Our modular architecture implements deconfigurable lanes within the decoupled sections of a superscalar pipeline that can be easily powered on or off to isolate faults or create an energy-efficient hardware configuration tailored to the needs of the running software.

At the system level, we propose a novel framework that uses surrogate response surfaces and heuristic global optimization algorithms to characterize the behavior of applications at runtime and dynamically redistribute the available chip-wide power to obtain hardware configurations customized for the software diversity and system goals. Our reconfigurable architecture is able to provide high performance under a strict power budget, maintain a certain performance level at a reduced power cost, and in the case of hard faults, restore the system's performance to pre-fault levels.

BIOGRAPHICAL SKETCH

Paula Petrica was born and raised in Timisoara, Romania. She attended the English-intensive high school William Shakespeare in Timisoara in the Physics and Mathematics-track class. She qualified and participated in the Romanian national-level Olympics in both Mathematics and English, and graduated first in her class in 2001. She arrived in the United States in August 2001 to pursue a degree in engineering at Brown University. She graduated with honors in 2005 with a Bachelor of Science degree in Electrical Engineering. Paula joined Cornell University in August 2005 and enrolled in the M.S./Ph.D. program in Electrical and Computer Engineering. She started working under the mentorship of Dr. David H. Albonesi in the Computer Systems Laboratory in May 2006 and performed research in the field of hardware reliability.

In 2008, Paula worked as an intern for Intel Corporation for seven months, joining their efforts to parallelize an internal microarchitecture simulator and performing research in the field of transient faults.

Following her return to Cornell University, Paula resumed research on power and reliability-aware multiprocessors, focusing on reconfigurable architectures and global optimization techniques.

Paula defended her Ph.D. thesis in September 2011 and plans to join Intel Corporation in Hillsboro, OR as a Microprocessor Design Engineer in the Visual and Parallel Computing Group.

To Andra, Gabriela, and Mircea Petrica, who taught me that a future is built, not dreamt. Va iubesc.

ACKNOWLEDGEMENTS

It is a pleasure to express my gratitude towards my adviser, Dr. David H. Albonesi for all his help and patience. His knowledge, integrity, clarity of thought, and dedication have inspired and humbled me throughout the past six years. Dave, thank you for keeping me motivated, for steering me in the right direction, for tirelessly working along side me, for believing in me even when I did not, for taking time to talk for hours about my future, and for constantly being there for me both in my academic and personal life. I would have been lost without you.

Many thanks to my committee members, José Martínez and Rajit Manohar, for asking the hard questions and sometimes answering them as well! I am in awe of your intellect and it has been an honor to meet and work with you.

I wouldn't be here without the help of Tim Correia, my rock and dearest friend. It is hard to think of anything that he did NOT help me with. He was the first to show me that engineering is fun, he explained everything to me (sometimes even hundreds of times!) better than any professor could, he encouraged and helped me in research, pushed me to push myself, and perhaps most importantly he is the owner of the best shoulder to cry on that I have ever encountered.

Many thanks to my wonderful office mates Basit Sheikh, Jonathan Winter, Mark Cianchetti, and Matt Watkins, for creating a work environment that encouraged numerous talks on both research and the randomest topics outside research. Jonathan, thank you for being my unofficial second adviser. Basit, thank you for being my most loyal friend, for the many intellectual discussions and the ideas that came out of them, and for the unforgettable memories. For you, a million times over.

I owe my loving thanks to Wacek Godycki who put up with me during my hardest time as a Ph.D. student, continuously encouraged me, and made me smile

every day.

I am forever indebted to my dear friends in Ithaca who kept me sane throughout this endeavour, filled the past years with joy, and made Ithaca unforgettable.

Finally, I want to thank my parents, my sister, bu and buni, mamaie and tataie, and all my extended family for unconditionally loving and supporting me and for teaching me what hard work, resilience, and love mean. I hope I have made you proud.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Power and performance challenges are expected to increase in future multicore systems with a higher degree of component integration. As the feature size continues to shrink, more resources can be incorporated in microprocessors to address a broader spectrum of different application requirements. However, power constraints will limit the amount of resources that can be powered on at any given time. Recent studies have shown that, without innovation, future multicore systems will be able to power on less than 80% of their transistors in the near future, and less than 50% in the long term [26]. Others [73] cite even more drastic effects due to the exponential drop in the percentage of a chip that can actively switch. A difficult challenge is deciding which transistors should be powered on at any given time to deliver high performance under strict power constraints.

Another serious issue is device reliability - the proliferation of devices that will either be defective at manufacturing time or will fail in the field with usage. Both types of failures are projected to be exacerbated by the continued scaling of device sizes. Inaccuracies in the manufacturing process will become more prominent as feature size is decreased and the manufacturing process becomes more complex. Thus, the International Technology Roadmap for Semiconductors is warning that improving yield will become just as important as performance and power [5]. Moreover, the increased circuit density of future microprocessors will result in higher probabilities of device wear-out, limiting overall microprocessor lifetime.

In order to achieve acceptable yield, a chip with intrinsic manufacturing defects should have a high probability of being reconfigured in the factory in a way that creates a functional chip that achieves close to the throughput of a pristine chip that

is devoid of defects. Similarly, to achieve acceptable levels of lifetime reliability [68], the system must detect the onset of a wear-out fault, determine its source, and in most cases, be able to reconfigure the chip in a way that isolates the affected region yet maintains operability at close to peak throughput.

We propose a modular dynamically reconfigurable architecture as a unified technique to address the problems of dark silicon (potential computing power that cannot be used at once) and reliability. The architecture implements deconfigurable lanes within the decoupled sections of a superscalar pipeline that can be easily enabled or disabled to create an energy-efficient hardware configuration tailored to the needs of the running software. Moreover, the modularity of our design is amenable to fault isolation, allowing faulty chips to maintain correct functionality and dormant performance boosting techniques can be enabled to recoup the associated performance loss. Our techniques are orthogonal to other power saving or performance enhancing techniques such as dynamic voltage and frequency scaling, the recently proposed conservation cores [73], or accelerators.

In order to take advantage of the large real estate that will be available, we envision a system that incorporates both regular cores and a number of accelerators and static or dynamic custom hardware optimized for different goals. At any given point, a different subset of the chip transistors is powered on to provide performance tailored to the software diversity. This subset of transistors will most likely consist of some of the regular cores and some of accelerators, but the proportion of one to the other will vary depending on the currently running software. Traditional methods include dynamic voltage and frequency scaling or turning off entire cores to match the wide range of optimal power that should be allocated to regular cores. With the breakdown of voltage scaling, the opportunities for power savings with

DVFS diminish significantly, and exclusively turning off cores might come at a high performance cost. Moreover, single threaded performance remains an important system goal, as it is highly unlikely that all or most of the applications will be able to be parallelized in their entirety to the degree needed to permint only weak cores in the microprocessor. Our modular architecture provides designers with another degree of freedom to dynamically optimize the power-performance of the integrated regular cores with less single threaded performance loss than weak cores. It can thus increase the amount of power available to customized hardware, maintain tolerable single-threaded performance guarantees, and optimize the performance of the remaining running cores.

At the system level, we present a novel framework that uses surrogate response surfaces and heuristic global optimization algorithms to characterize the behavior of applications at runtime and dynamically redistribute the available chip-wide power to obtain hardware configurations customized for the software diversity and system goals. Through the judicious use of resources (lanes and performance boosting functions) based on their energy efficiency for particular applications, our reconfigurable architecture is able to provide additional performance under a strict power budget, maintain a certain performance level at a reduced power cost, or in the case of hard faults, restore the system's performance to pre-fault levels.

This dissertation makes a number of significant contributions:

- We introduce a lane-based modular architecture where cores are homogeneously designed and dynamically reconfigured into a heterogeneous system that addresses both power and reliability concerns;
- We propose a formal methodology for dynamically characterizing application behavior using surrogate response surfaces;

- We show how expensive sampling evaluations can be reduced and their accuracy improved through methodical experimental design;
- We take advantage of the variety in application characteristics and tailor the underlying hardware to individual and global goals by redistributing power among the cores of a chip multiprocessor;
- We identify the problem of pipeline imbalances due to hard faults and the subsequent hardware deconfiguration, and show that these imbalances are application phase dependent;
- We show that dynamic deconfiguration of other fully operational pipeline sections (using mechanisms already present for fault tolerance) can save significant power at little performance cost;
- We propose to transfer this saved power to alternative boosting techniques, and we identify three complementary techniques that work well for a variety of applications;
- We develop online heuristic optimization techniques that permit scaling our approach to large-scale CMPs.

CHAPTER 2

RELATED WORK

2.1 Processor Adaptivity

A number of prior efforts have focused on architectural techniques that adapt a single core's components to workloads. Albonesi et al. propose Complexity-Adaptive Processors that dynamically disable underutilized hardware to improve performance or power efficiency [3, 4]. Iyer and Marculescu develop a run-time profiling technique to detect program hotspots and adapt the processor configuration to match the hotspot demands [40]. Huang et al. propose a positional approach that uses program subroutines as the granularity for reconfiguration [37]. Hu et al. [36] employ a run-time virtual machine to detect application segments with different characteristics, compare possible hardware configurations, and direct the hardware to adapt to the best option.

Much of the work in this area examines a particular processor structure and makes it more efficient through adaptation. Buyuktosunoglu et al. [18] design an adaptive issue queue, and develop coordinated adaptive fetch and issue mechanisms [19]. Folegnani and Gonzalez also develop a resizable issue queue [28]. Balasubramonian el al. [8] investigate caches with variable sizes and associativities and variable sized TLBs for power-performance efficiency. Dropsho et al. [24] extend this work by developing a more precise way of adapting caches, and use limited histogramming to more effectively configure the issue queues, load-store queue, and reorder buffer. Ponomarev et al. focus on adapting the issue queue, load-store queue, and reorder buffer based on historical usage patterns, showing that optimal sizing is often correlated [57]. Bahar and Manne [7] examine a more

coarse grain adaptation that disables an entire back-end execution cluster to save power.

In the multicore domain, proposals include asymmetric chip multiprocessors [49] consisting of cores of varying computational strengths. The die composition is static, but the hope is to match the demands of the currently running workload to one of three available core sizes. This technique incurs the highest area overhead if flexibility is desired, since a number of separate cores are needed for each application. In contrast, our technique adapts a single core with much smaller overhead to the same computational capabilities. Venkatesh et al. [73] recently proposed Conservation Cores, which are specialized, energy-efficient processors to reduce energy per operation and are integrated on a chip in addition to general purpose cores. Our technique works elegantly in conjunction with this proposal, maximizing performance for sequential applications or sequential portions of parallel applications and engaging conservation cores for the parallel portion. One interesting adaptive technique is Core Fusion [38] in which small clusters are fused together or operated separately as distinct processing elements. This approach is promising despite its overhead, but requires monolithic structures for coordinating fetch, steering, and commit, which become single points of failure. Finally, Gupta et al. [31] propose a unified approach to power and reliability with Core Genesis. They propose slicing the pipeline vertically, which incurs very high interconnect overheads. Moreover, this technique requires compiler directed instruction steering, which makes it incompatible with legacy software. In addition, none of the techniques for multicore adaptivity described above provides an in depth comparison with both powerful (high ILP) and weak (low ILP) cores in the context of two common power management techniques: dynamic voltage and frequency scaling and core disabling.

2.2 CMP Power Management

While there has been much work on power management for CMPs, we focus on the most related work where performance is maximized under a chip-wide power constraint. Isci et al. [39] developed the widely cited per-core maxBIPS algorithm. Sharkey et al. extend this work by exploring algorithms based on both DVFS and fetch toggling, and explore a number of design tradeoffs such as local versus global management [62]. Bergamaschi et al. also conduct further work on maxBIPS and compare its discrete implementation to using continuous power modes [9]. Kim et al. develop and analyze on-chip voltage regulators to allow for per-core DVFS and, using an offline algorithm, show significant performance benefits from applying DVFS at a fine granularity [45]. Finally, Teodorescu and Torrellas [72] consider global power management in the presence of process variations and propose using linear optimization to efficiently find a near optimal allocation of power to cores.

2.3 Detection and Deconfiguration of Faulty Processor Components

Prior research on hard errors falls into several categories: (1) developing architectural models for manufacturing defects and lifetime wear-out and reducing the occurrence of these errors; (2) detecting the presence of permanent faults and isolating their impact; and (3) maintaining processor functionality despite the occurrence of an error.

Srinivasan et al. [66] were among the first to look at lifetime reliability from an architectural perspective. They developed a model called RAMP for studying

the impact of microarchitectural design decisions and runtime behavior on lifetime wear-out and proposed dynamic techniques to increase reliability. Kang et al. [42] develop a method for correlating changes in leakage power to increases in NBTI degradation. Blome et al. [10] design an online hardware unit for the detection of gate oxide breakdown and to study this failure mechanism at the microarchitectural level. Feng et al. [27] extend that work by using the wear-out detection units to intelligently schedule jobs to manage lifetime wear-out.

Austin [6] developed a technique, called DIVA, for detecting hardware faults at the architectural level using simple checkers at the commit pipeline stage. Chatterjee et al. [20] continue to improve the checker to make it more performance efficient. Bower et al. [13] extend the capabilities of DIVA, adding mechanism to isolate the faults, correct the errors, and deconfigure faulty units. Distributed built-in self-testing and checkpointing techniques are devised by Shyam et al. [65] for detecting and recovering from defects. Meixner et al. [53] consider a different approach to error detection in simple cores that verifies that the four invariants of von Neumann-style processors hold during execution. Yilmaz et al. [77] focus on techniques to detect delay faults that cause timing errors in functional units. Schuchman and Vijaykumar [61] likewise focus on developing means for testing and isolating faults in the core logic. LaFrieda et al. [51] propose using dynamically coupled cores in a chip multiprocessor to provide fault detection through redundancy in a far more efficient manner than traditional static binding of core pairs. In this dissertation, we assume the use of the above techniques for detecting and isolating faults in our chip multiprocessor, so that these faulty units can be deconfigured.

Shivakumar et al. [64] are the first to propose that the inherent redundancy in

a processor can be exploited for hard error tolerance. Bower et al. [12] describe a new method of detecting and recovering from errors in processor array structures. Their mechanism uses spare rows in the structure that replace faulty ones that are mapped out. Srinivasan et al. [67] propose two methods to increase the processor lifetime: structural duplication and graceful performance degradation. Aggarwal et al. [1] study mechanisms for isolating faulty components in a CMP and reducing an error's impact through reconfiguration. Meixner and Sorin [54] describe a technique for automatically modifying software in a way that maintains its functionality but changes the application's usage of the hardware to circumvent a faulty component. A number of papers develop schemes that tolerate permanent faults and allow the microprocessor to remain functional. At a coarse grain, ElastIC [70] and Configurable Isolation [2] propose disabling faulty cores in a chip multiprocessor. Both proposals assume the availability of a large number of redundant cores. At a finer grain, StageNet [30], and Core Cannibalization [60] propose slicing the pipeline vertically, disabling stages in a simple 5-stage pipeline, and recombining the remaining active stages in one pipeline with stages in other pipelines. StageNet only works for simple architectures targeted at the embedded domain and requires a complex interconnection network between the stages. Core Cannibalization reduces the complexity of the interconnect, but only lends some pipeline stages to faulty pipelines. Many of the schemes tolerate hard errors by deconfiguring faulty components, keeping cores functional but in a degraded state.

CHAPTER 3

LANE-BASED PIPELINE ARCHITECTURE

This chapter presents a microarchitecture suitable for power-limited environments where hardware reliability is also of concern. A four step system level operation is periodically engaged through a Runtime Manager as shown in Figure 3.1 and detailed in Chapter 4. The system uses a lane-based modular architecture where cores are homogeneously designed and dynamically reconfigured into a heterogeneous system that addresses both power and reliability concerns. In this chapter, we describe the hardware modifications required to support modular reconfiguration.

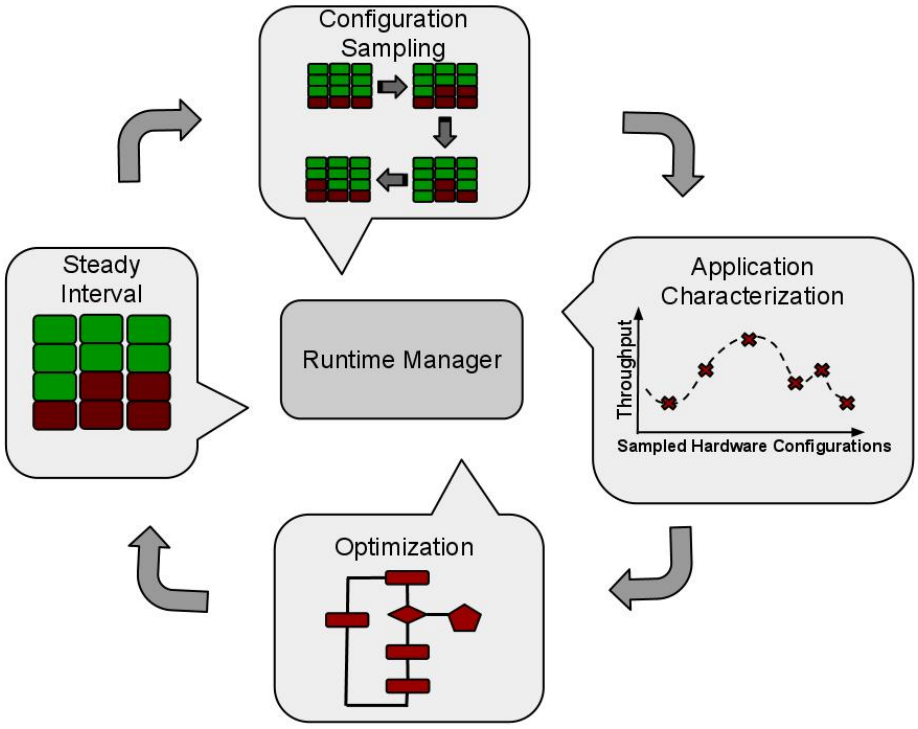


Figure 3.1: System-level diagram.

The pipeline within each core is divided into three regions: Front End (FE), Back End (BE) and Load Store Queue (LSQ), each of which has four lanes (Fig-

ure 3.2). The pipeline resources associated with each region are shown in Table 3.1. Each pipeline lane includes a sub-bank of the associated queues, even though they are not technically part of the pipeline "width." As the peak bandwidth of a region is reduced by deconfiguring a lane, the buffering requirements (and the issue window requirements) are reduced commensurately. This permits the associated queues within the region to be downsized to save power. We exclude single point of failure structures from our study (such as the Integer Mult/Div or the FP Simple ALU and Mult/Div) since these structures do not have redundancy that would allow the chip to still operate correctly in a degraded state.

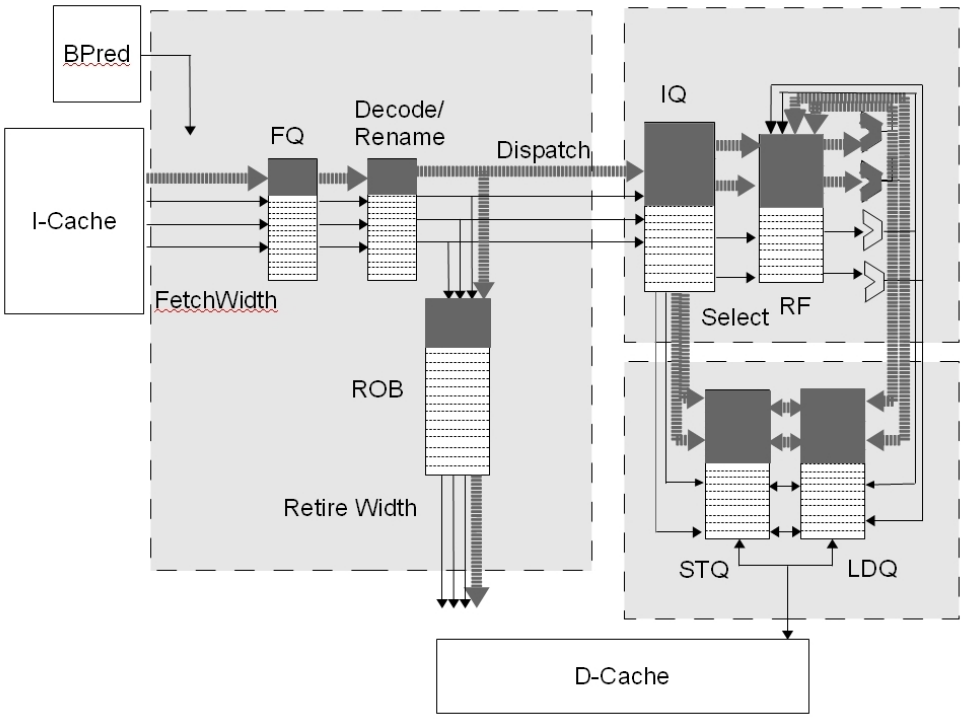


Figure 3.2: Lane-based pipeline microarchitecture showing the FE, BE, and LSQ regions. One FE lane, two BE lane, and two LSQ lanes have been deconfigured to match a scheduled application.

To affect lane-based deconfiguration, we implement both *physical gating* and *logical correctness* mechanisms. The physical gating mechanisms include the sleep

Front End	Back End	Load Store Queue
Fetch Width	Issue Queues	Load Queue
Fetch Queue	ALUs	Load Queue Ports
Decode Width	Select	Store Queue
Rename Width	Wakeup	Store Queue Ports
ROB	Register Files	
Retire Width		

Table 3.1: The three pipeline regions and their corresponding structures

transistors that are engaged to power down each of the blocks that constitute a lane. In addition, supply voltage levels are slightly increased to account for the voltage drop across the sleep transistors, and additional decoupling capacitance is provided to reduce voltage fluctuations in the power grid [41].

The logical correctness mechanisms ensure proper pipeline operation when lanes are deconfigured. These mechanisms always remain powered on and are described below for each pipeline region.

3.1 Logical support for lane-based pipeline architecture

3.1.1 Front End

Conventional caches, such as the L1 instruction cache, incorporate redundant rows and columns that permit fully-functional operation in the face of manufacturing defects or wear-out faults. Moreover, when a lane is deconfigured, the associated instruction decoder can simply be gated off so long as the fetch logic is prevented from slotting instructions into the deconfigured lane. The more challenging task is deconfiguring the Fetch Queue, the Rename Logic, and the Reorder Buffer (ROB).

Bower et al. [12] developed circular array structures with spares that can be deconfigured at a fine-grain, per-entry, level by feeding fault information into the head and tail pointer advancement logic. We adapt these techniques to our coarser-grain deconfiguration of the Fetch Queue and the ROB. Here, the queues are banked and an entire bank is deconfigured, thus requiring a fault map of only four bits, one for each bank that can be deconfigured (Figure 3.3). Unlike [12], our architecture does not include spare banks; therefore, the buffer size is also updated when banks are deconfigured or reconfigured.

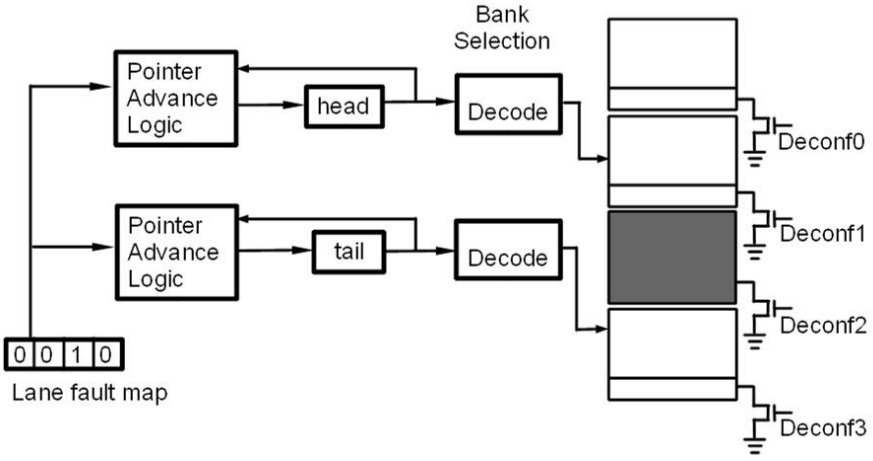


Figure 3.3: Mechanism for deconfiguring banks of circular queues (adapted from [12]).

The decode stage is deconfigured by gating off one of the four instruction decoders, while the rename stage consists of two parts: dependency checking and logical to physical register mapping. The former is deconfigured by gating the circuit that compares an instruction's source registers to previous instructions' destination registers.

In the absence of a rename fault (i.e., either a fault in another part of the front-end or symbiotic deconfiguration of the front-end), deconfiguring a front-end lane disables the associated read port of the map table. In addition, the relevant

dependency check logic comparators are gated off.

There are at least two ways to implement register mapping: the first uses a RAM indexed by the logical register number to store physical register numbers; the second uses a CAM with the same number of entries as physical registers to store logical register numbers. The recovery mechanism differs for the RAM and CAM-based rename schemes. For the RAM-based scheme, spare rows [12] are required for recovery. With the CAM approach, spares can be implemented or the associated physical register can be prevented from appearing on the free list since it can no longer be mapped to a logical register. We model a RAM-based scheme implemented with spare rows. Thus, whenever the front-end is deconfigured, only the associated rename ports are disabled. Albonesi et al. [4] present a comprehensive discussion on the rename downsizing operation.

3.1.2 Back End

For the issue queue, we adapt the approach of Dropsho et al. [24], who demonstrate a coarse-grain partitioned RAM/CAM based issue queue that dynamically adapts its size to program demands. Unlike [24], in which one partition is always active, each of our banks incorporates its own precharge and sense amp circuitry to allow deconfiguration of any of the partitions (Figure 3.4).

The select logic is designed as an arbiter tree [56], and selection priority is based on Issue Queue position. Each arbiter cell makes a local selection decision between four instructions. For four-wide issue and an Issue Queue size of 32 entries, two arbiter cells are associated with each lane. When a lane is deconfigured, the associated arbiter cells are gated off and the request lines are pulled low (Figure 3.5).

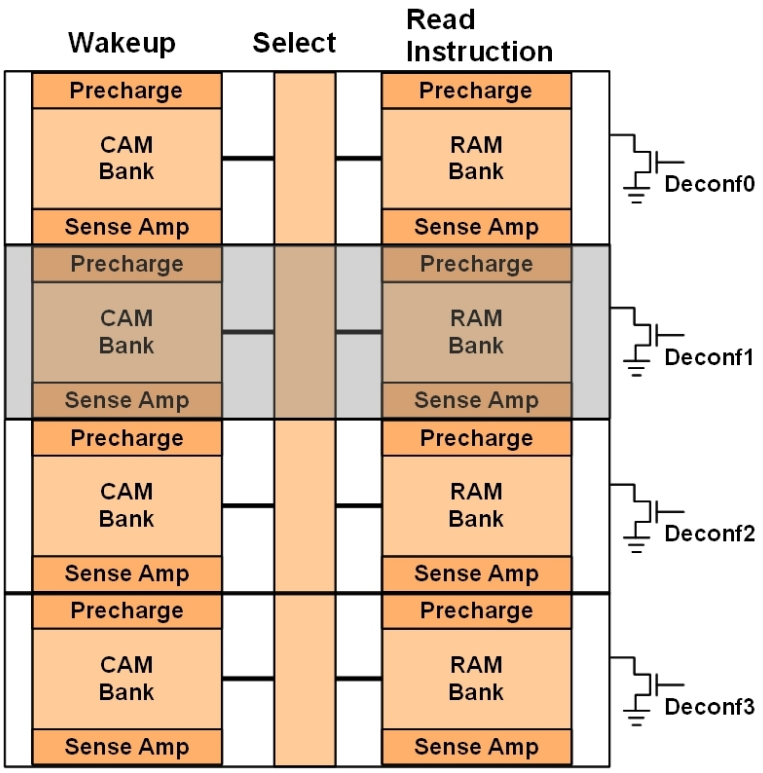


Figure 3.4: Issue Queue deconfiguration (based on [4]).

Register file deconfiguration can be done in a number of ways. If the register file is fault-free, one option is to simply deconfigure the associated read and write access ports when a back-end lane is deconfigured. A register file fault can be handled at a fine-grain level through spare RAM rows. Alternatively, the register file RAM can be banked and deconfigured at a coarse-grain level similar to the Issue Queue RAM. Each of the four register file banks has an associated free list in the rename stage, similar to the MIPS R10000 [76]. When a register file bank is deconfigured, the associated free list in the front-end is disabled as well, and registers are only allocated from the remaining free lists. This effectively eliminates the rename stage's capability to map a new destination register that is present in the deconfigured register file bank. To maintain full rename bandwidth, the free list FIFOs can be augmented to have two read ports instead of one. Alternatively,

l

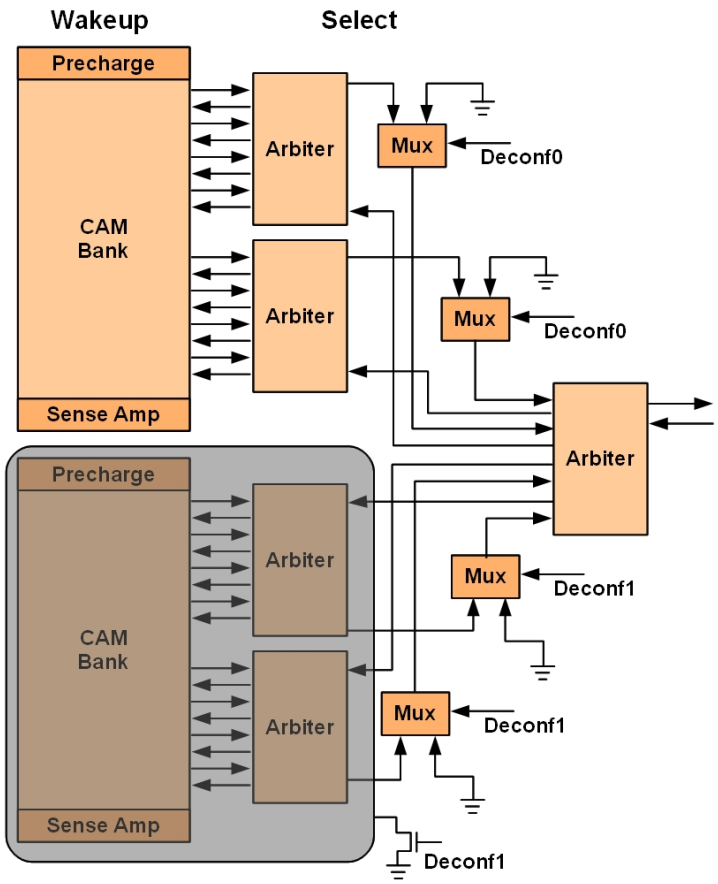


Figure 3.5: Wakeup and Select deconfiguration.

the architecture can simply tolerate this reduction in functionality. We model the latter coarse-grain option in our evaluation.

Finally, a functional unit associated with a deconfigured lane is marked as perpetually in use, and its Issue Queue access ports are gated off.

3.1.3 Load Store Queue

The Load Store Queue RAM is a circular array structure that can be deconfigured following the same procedure described for the ROB and Fetch Queue. The Load Store Queue also includes a content addressable memory that can be associatively searched to determine conflicts with older stores and loads. The CAM part of the LSQ can be partitioned and deconfigured in a similar fashion as the Issue Queue CAM.

3.2 Physical Gating Mechanisms

Physical gating of deconfigured functionality within a lane can be achieved through power-gating techniques proposed to reduce leakage power and to implement microprocessor deep sleep states, such as C6. Intel Core i7 microprocessors implement power-gating transistors to shut off idle cores [50] and a number of designers have proposed a variety of power-gating techniques for finer grained blocks [21, 41, 63]. Either high-V_t PMOS (header) or NMOS (footer) transistors are used to connect or disconnect the permanent power supply from the circuit virtual power supply. Power gating can be implemented at a fine grain [21], where each standard cell has a sleep transistor, or at a coarse grain, where clusters of gates in the same voltage domain have an array of sleep transistors distributed in a ring or grid style [63]. Fine-grained sleep transistor schemes usually incur a higher area overhead and are sensitive to process, voltage, and temperature variations (PVT). Coarse-grained sleep transistor implementations share charge and discharge current and are thus both smaller and less sensitive to PVT, but suffer more from ground bounce.

Sleep transistor area overhead estimates vary from 2% to 6% depending on the implementation, size of clusters, and technology node [63, 44]. Moreover, advanced sleep transistor sizing algorithms can considerably reduce the area overhead [21]. In addition to the sleep transistors, area overheads are introduced by additional decoupling capacitance that has to be incorporated to reduce voltage fluctuations, resulting in a total estimated overhead of 15% [41]. While dynamic power is slightly increased (by approximately 2% according to [41]), static power can be reduced by almost 90%.

A PowerTransfer design leverages the sleep transistors that are increasingly implemented in commercial microprocessors for leakage reduction and deep sleep states. Power-gated functional blocks are aggregated into 12 individually controllable power-gated lanes, four for each of the FE, BE, and LSQ regions. The logical correctness circuitry remains powered on at all times to ensure correct pipeline operation.

CHAPTER 4

OPTIMIZATION TECHNIQUES FOR POWER EFFICIENCY

In this chapter, we present optimization techniques that exploit the lane-based architecture from Chapter 3 in order to maximize performance under different power constraints.

4.1 Application Characterization

An efficient allocation of hardware resources is application specific and is dependent on a quantitative understanding of the software characteristics. Ideally, each application can be profiled offline and its power-performance behavior under each hardware configuration stored to be used at runtime by a decision algorithm that selects the optimal hardware allocation according to an optimization goal. However, it is unreasonable to assume that such profiles will be available for all possible real-world applications and moreover that this information will be distributed with every deployed hardware system. As such, we propose to characterize applications at runtime, every time they are scheduled on an active processor. We accomplish this by sampling the behavior of the active application for short periods of time under a variety of hardware allocations (configurations) and then building a response surface that approximates its characteristics for use in an optimization protocol.

Modern operating systems schedule processes to be run on the CPUs from a queue of runnable applications. There are many more processes than available CPUs and the operating system needs to ensure that all of them make progress in a timely fashion. As such, most operating systems support scheduling - determining which process should be executed, and preemptive multitasking - an interrupt

mechanism that suspends the operation of a currently running process (switches it out) to allow another process to run on the CPU. The latter ensures that all processes will be alloted some amount of CPU time at any given moment, which is generally referred to as the operating system time quantum or time slice. Every time slice, the OS scheduler is run to determine which process (or processes in multicore systems) should be executed next; that process is then run on a core until its time quantum expires, after which the state of the running process is saved, the process switched out, and the scheduler invoked again. The time quantum should be long enough that the scheduler and context switch overhead are minimized, and short enough that the jobs in the ready queue have a reasonably small waiting time until they are scheduled to run. For modern operating systems, the time quantum can take on values between 50 and 200 ms. In this dissertation, we assume that each application is allowed to run for 100 ms before it is switched out. We also assume that each process starts with a cold cache at the beginning of a time quantum, as there most likely have been a number of intervening running processes scheduled on the core which have evicted useful cache blocks even in the presence of operating system scheduler modifications such as processor affinity scheduling.

We propose splitting the time quantum into four intervals: sampling interval, surrogate surface fitting, optimization interval, and steady interval, as shown in Figure 4.1. In the sampling interval, we collect information about the currently running application and its hardware resource needs by changing the underlying hardware configuration and executing the application for short periods of time on these different configurations. As we will show in Section 4.3.2, the choice of individual sample length is important and involves tradeoffs between sample accuracy and steady interval performance. Based on the samples collected, the response sur-

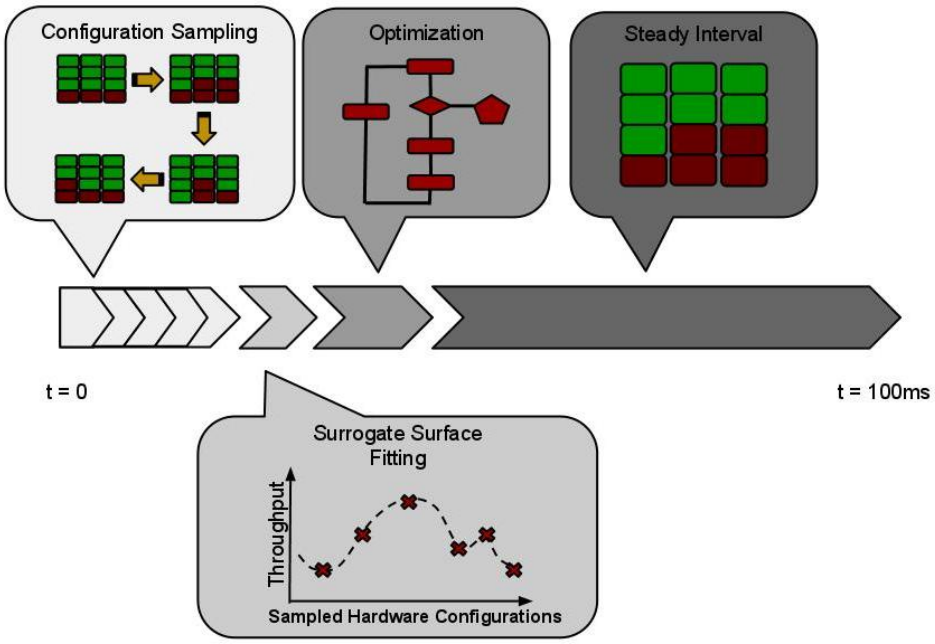


Figure 4.1: Events timeline within an operating system time quantum.

face fitting interval predicts the behavior of the application at other design points (hardware resource allocations) by fitting a function to the observed data. This surrogate function is then used by an optimization algorithm to select individual core hardware allocations that optimize a global goal. Lastly, in the Steady State Interval, the best configuration found is run for the remainder of the time quantum. The sampling and optimization intervals should be short enough compared to the steady interval; otherwise most of the OS allocated time quantum is spent testing configurations that are suboptimal. At the same time, the same sampling and optimization intervals should be long enough to minimize the observed data error and to allow enough time for a good solution to be found. Otherwise, the configuration selected for the steady interval will be suboptimal.

4.1.1 Experimental Design Approach

We formulate the characterization of each running process as a multivariate statistical experimental design, which results in an empirical model that correlates hardware resource allocation with power and performance. This design can then be used to optimize the hardware-software system for a variety of goals. For example, one goal can be maximizing the global performance of a chip multiprocessor under a certain power budget. Other optimization goals are discussed in Section 4.2.

There are two types of variables in a multivariate optimization procedure: *responses* and *factors*, where the responses are observed (or sampled) output values dependent on the values taken on by the factors. The response variables for this experimental design are the throughput (BIPS) and power usage of the running application, and the goal is to characterize the effect of different lane allocations on these variables in order to obtain an optimal resource allocation. The factors are the controlled independent variables that affect the response of the system. In this example, the three pipeline regions (FE, BE, and LSQ) are the factors of the experiment, denoted as X_1, X_2, and X_3, respectively. Each of the factors can take on three different levels (4 active lanes - fully provisioned, 3 active lanes, and 2 active lanes). Thus, there are $3^3 = 27$ hardware configurations, or *treatments* for our system.

$$x \rightarrow \frac{X - a}{b}, \quad where \tag{4.1}$$
$$a = \frac{X_H + X_L}{2} \quad and \quad b = \frac{X_H - X_L}{2}$$

The levels of factors X_1, X_2, X_3 are transformed into coded variables x_1, x_2, and x_3, which are dimensionless, have mean 0 and the same standard deviation.

Using Equation 4.1, we linearly transform the original measurement scale such that the high setting (X_H, all 4 lanes active) for each pipeline region becomes 1 and the low setting (X_L, 2 active lanes) becomes -1. Each factor will have 3 symmetrically spaced levels, -1, 0, 1, corresponding to 2 active lanes, 3 active lanes, and 4 active lanes, respectively.

4.1.2 Sampling Techniques

In the most straightforward case, all 27 treatments are sampled and their effect on the response variables measured. Such a design is called a full factorial design and is depicted in Figure 4.2 (left). The design space can be graphically represented as a cube, where the edges are the levels of factors and the corners correspond to the high and low values of each factor. The blue circles represent all factor level combinations. Full factorial designs have the advantage that the response surface is fully described by the samples (no error in the coefficients of the response surface) and that both main effects (individual effects of each of the factors) and higher order effects (interactions between the factors) can be studied. However, the large number of samples needed for a full factorial design limits its usefulness in runtime applications, as a large portion of the time needs to be spent sampling suboptimal configurations. Moreover, it is likely that some of the higher-order interactions are negligible, and only some of the factors are actively contributing to significant changes in the response variables. For example, the size of the Back End (x_2) significantly impacts the throughput of a CPU bound application like apsi, whereas it affects the BIPS response for a memory bound application like art to a much lesser extent. These characteristics argue for designs that use a reduced set of experimental runs to estimate the system response.

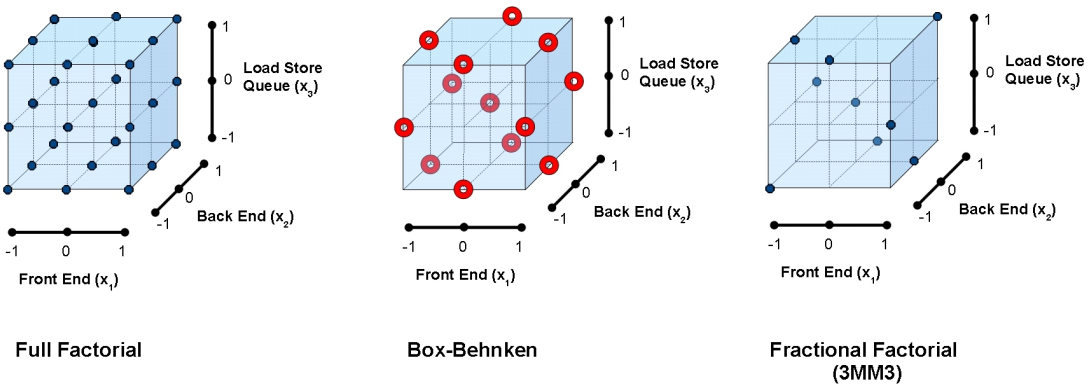


Figure 4.2: Sampled treatments for the Full Factorial (left), Box-Behnken (center), and Fractional Factorial (right) designs.

We use two well established methods of reducing the cost of experimentation that have been proven to estimate response surface parameters with high precision: Box-Behnken design [14] and Fractional Factorial design, both consisting of a subset of the treatments needed for a full factorial design. The designs are based on the sparsity-of-effects principle, which states that it is most likely that main (single factor) and low-level (two factor) interactions are the highest contributors to responses. Moreover, they are both balanced and orthogonal, which ensures optimal efficiency.

4.1.2.1 Box-Behnken Design

The Box-Behnken design was developed by George E. P. Box and Donald Behnken in the 1960s, and selects treatments that are at the midpoints of the edges of the design space and also one at the center, as shown graphically in Figure 4.2 (center). This design is particularly suited for our system because it requires at least three factors each with at least three levels. Since we suspect that the effect of the factors on the dependent variable is not linear, the Box-Behnken design is ideal because

it allows for quadratic response surface fitting. The number of samples required for a Box-Behnken design is:

$$N = 2k(k-1) + C$$

where k represents the number of factors and C represents the number of center points. For our particular design there are three factors and we include one center-point (0,0,0), which results in 13 required samples. This design more than halves the number of runs required for a full factorial design, thus increasing the amount of time available during the steady interval, when an optimal configuration is run. In order to obtain the design matrix, each of the three factors is separately fixed at its center point and then combined with the full factorial of the other two factors, as shown in Figure 4.3 (a).

x_1	x_2	x_3
-1	-1	0
-1	1	0
1	-1	0
1	1	0
-1	0	-1
-1	0	1
1	0	1
1	0	-1
0	-1	-1
0	-1	1
0	1	-1
0	1	1
0	0	0

(a)

x_1	x_2	x_3
-1	-1	-1
-1	0	1
-1	1	0
0	-1	1
0	0	0
0	1	-1
1	-1	0
1	0	-1
1	1	1

(b)

Figure 4.3: Design Matrix for Box-Behnken (a) and Fractional Factorial (b) designs.

4.1.2.2 Fractional Factorial Design

We employ a class of fractional factorial designs called 3^{k-p} designs, where k is the number of factors and 3 represents the number of levels of each factor. A 3^{k-1} design reduces the number of samples by three, and 3^{k-2} reduces the number of samples by nine. It is unfeasible to construct an accurate response surface for three factors using only three samples. For example, a quadratic response surface has 10 coefficients as described later in Section 4.1.3, which means that the coefficients have to be estimated from a system of three equations with ten unknowns. This would limit the type of function that could be fitted to the data to one that only uses three coefficients. Thus, we choose to use a 3^{k-1} design, which reduces the number of samples to nine[1]. The procedure to generate the nine samples is as follows:

1. Start with a smaller full factorial design using only two of the three factors, for example x_1 and x_2, listed in the first two columns of Figure 4.3(b).
2. Construct factor x_3 from interactions between factors x_1 and x_2 using the function:

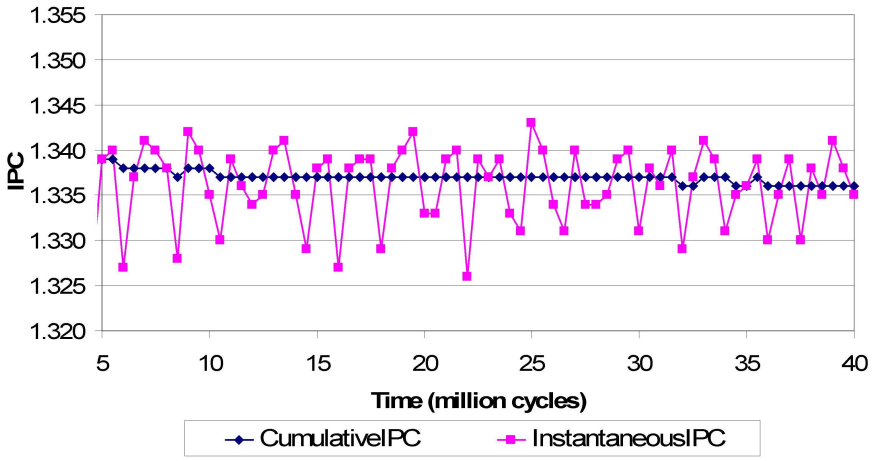
$$x_3 = mod_3(3 - (x_1 + x_2 + 2)) - 1 \tag{4.2}$$

We refer to the fractional factorial design obtained with Equation 4.2 as the 3MM3 Design. The number 2 is added to the sum of x_1 and x_2 in order to transform them from negative to positive by changing the factor level scale from (-1,0,1) to (0,1,2). The subtraction of 1 from the modulus transforms x_3 back to the original (-1,0,1) scale.

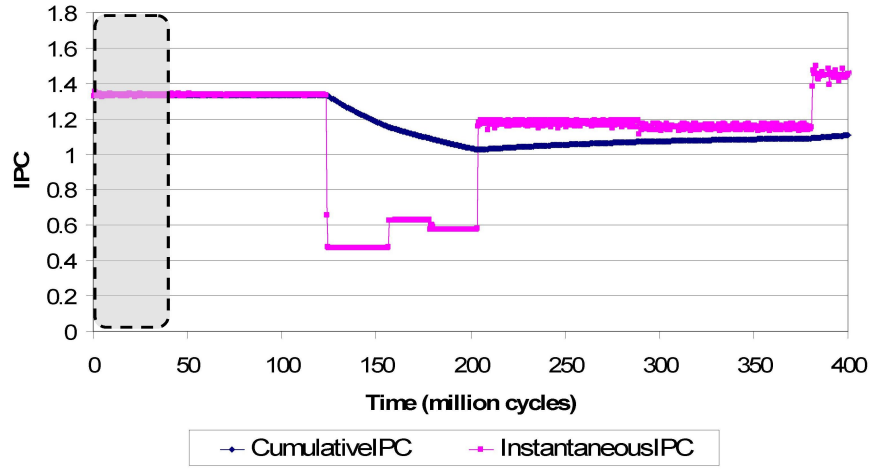
[1]Note that 9 samples are not enough to obtain all 10 coefficients for a quadratic response surface. We discuss in Section 4.1.3 an alternative response function.

4.1.2.3 Dynamic System Considerations

The analysis in the previous sections attempts to characterize the behavior of each application over the entire decision interval of 100 ms. In the ideal case, each treatment is run for the entire decision interval, and its power and performance characteristics averaged over the 100 ms. However, a real runtime manager must estimate the long-term performance of the application by running all treatments for a short period of time (sampling interval) as previously shown in Figure 4.1. Sampling introduces noise in the system because the behavior of an application during a short sample is possibly different from its longer run behavior. There are two types of noise observed in our system, which we refer to as high and low frequency noise. High frequency noise occurs when small adjacent samples of the same configuration do not have the same behavior, as seen in Figure 4.4(a). As such, it is difficult to interpret whether changes in the responses are due to the factor levels or due to microarchitectural events inherent to the benchmark. On the other hand, low frequency noise occurs if the average difference between samples of the same configuration is small, but they are not representative of a longer run of the same application (Figure 4.4(b)). This effect is mostly due to phase shifts in the application behavior. The latter noise is hard to minimize without dynamic phase detection and resampling, which is difficult to implement in many core systems and is left for future research. High frequency noise can be reduced by increasing the size of the samples, which at the same time reduces sensitivity to pathological microarchitectural events and cold cache effects on the first samples of the time quantum. However, increasing the length of the samples reduces the length of the steady interval during which the optimal configuration is run and increases the time spent sampling suboptimal configurations. An alternative is to divide each treatment run into multiple smaller samples taken at different points



(a)



(b)

Figure 4.4: Sources of noise for applu (a) High frequency noise within the sampling period; (b) Low frequency noise. The shaded region represents the sampling period and is not representative of the behavior over the 125 to 200 million cycles interval.

in the application. Samples are thus condensed, replicated, and averaged as shown in Figure 4.5. The figure shows how three treatments that were originally sampled for one continuous 1 ms block are each split into three groups with a duration of 1/3 ms. The first group for all three treatments is run first, followed by the second

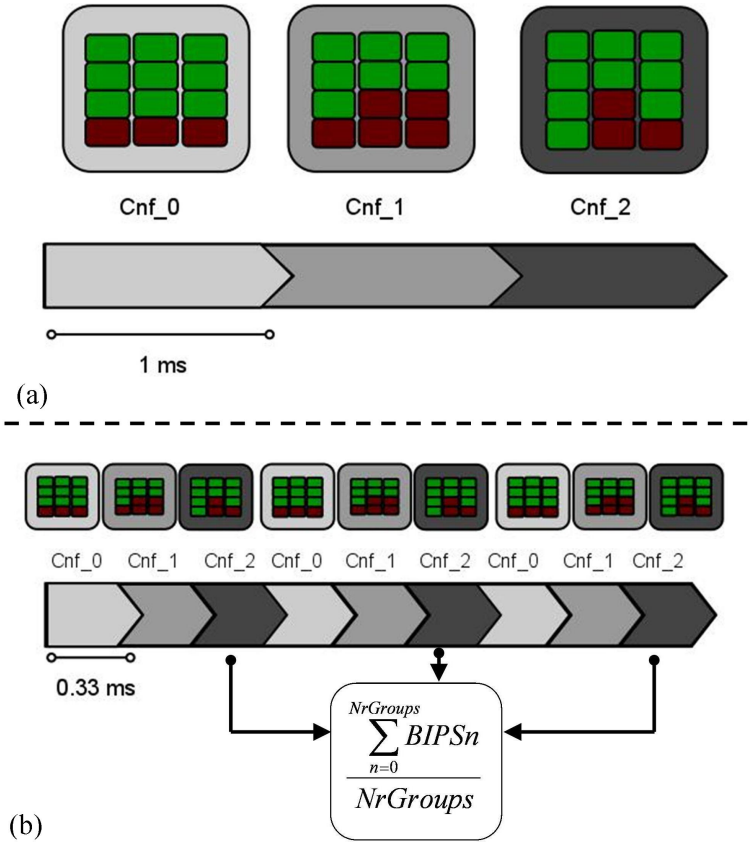


Figure 4.5: Configuration sampling possibilities: (a) Each configuration is run once for 1 ms; (b) Each configuration is run N times, for 1/N ms. The gathered statistics (Throughput and Power) are averaged over the N instances.

and third groups. Samples corresponding to the same treatments are evaluated at slightly different points in the application and their responses averaged, effectively filtering out some of the high frequency noise of the application. In our work, we evaluate sample replication 1, 2, 4, and 8 times. As the number of replicates or groups is increased, the total sampling interval stays constant, resulting in the smaller samples shown in Table 4.1. We will show in Section 4.3.2 that replicating the samples 8 times results in the most accurate results.

Number of replications	1	2	4	8
Replicate Size	1ms	0.5 ms	0.25 ms	0.125 ms

Table 4.1: Number of sample replicates and their corresponding runtime.

Theoretically, the samples become more representative with increasing the number of replicates. However, there are a few limitations to the number of times the samples can be replicated without increasing the total sampling time. This is due to the fact that the time for each individual sample linearly decreases with increasing number of replicates. First, as the samples become smaller, microarchitectural events such as cache misses and branch mispredictions affect the response variables more, partially hiding the effects of the factors in the experimental study. Second, each deconfiguration and reconfiguration incurs some overheads that are normally insignificant when the samples are hundreds of microseconds long. When the samples are decreased to the order of tens of microseconds or even hundreds of nanoseconds, the overheads start dominating the sample time. We limit the number of replicates to 8 groups, where each sample is run for 0.125 milliseconds.

4.1.3 Response Surface Models

Response surface models (or surrogate models) are inexpensive approximations of computationally expensive functions that need to be optimized. By computationally expensive functions we mean functions for which an *a priori* description or formula is not available, and information can only be obtained through time-consuming direct evaluation of the functions. In our system, each application is characterized by a different function and its response is obtained by sampling a subset of the input combinations as shown in Section 4. Since each configuration

sample (whether replication is used or not) has to be run for at least one millisecond in order to obtain significant results, the optimization process is dominated by the function evaluations (samples). With 27 treatments, or combinations of the independent variables, sampling all of them in order to obtain the exact description of the function to be optimized would consume almost 30% of the operating system time quantum. Surrogate models are particularly well-suited to our problem, since they construct a response function from a small subset of function evaluations. Moreover, our objective functions (global throughput and power) are nonlinear and nonconvex, with a large number of local minima, making standard nonlinear programming methods unsuited for finding the best solution.

Optimization algorithms based on surrogate models [33, 58] are usually iterative algorithms that use the metamodel to identify promising points for additional treatment evaluations through either derivative based or derivative free methods, update the response surface with the newly sampled points, and then repeat the process to obtain an optimal solution. Such implementations are time consuming and assume that the factors are continuous variables. Our system has four distinct properties that render these classical approaches non-optimal:

- **Discrete Variables**: The pipeline regions (variables) can only take on discrete levels, ranging from 2 to 4. Classical response surface methods are suited for continuous variables, which means that for our system the optima found by these methods need to be transformed back into discrete values, adding one more computational step and possibly resulting in suboptimal choices.
- **Small number of variables:** Previously proposed classical methods solve problems with a large number of variables that have theoretically infinite lev-

els due to their continuous nature. This makes the objective function very bumpy and heuristic algorithms converge very slowly to a good solution. On the other hand, our system has a small number of variables per core, each with only three levels, with the source of complexity arising from increasing the number of cores. Since we fit a response surface for each core, the complexity of previously proposed algorithms is not needed.

- **Monotonic response:** Within individual cores, the response monotonically increases with increasing factor levels. This provides an inherent guideline for heuristic search algorithms, which makes them efficient in the context of our optimization.
- **Online optimization:** Iterative response surface methods target offline optimization. While they are faster than non-response surface methods, they are not fast enough for the very strict time constraints of runtime optimization.

We choose to use the response surface methodology to obtain estimates of the function values at the points that were not sampled, and use heuristic search algorithms that are well suited to black box functions to find acceptable solutions in the optimization space. We study three flavors of surrogate functions, and build two metamodels $T(x_1, x_2, x_3)$ and $P(x_1, x_2, x_3)$ to approximate the throughput and power responses of the system. The next subsections explore the three response surfaces that we consider.

4.1.3.1 First Order Polynomial Surrogate Function

The simplest response surface is a first order polynomial (linear function) described by Equation 4.3.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 \tag{4.3}$$

Calculating coefficients for the first order polynomial is the least involved and least time-consuming out of the three surfaces we consider. Moreover, integer linear programming can be used to efficiently optimize linear functions with linear constraints. However, we show in Section 4.3.3 that linear functions do not predict the responses (especially power) of our system well.

4.1.3.2 Second Order Polynomial Surrogate Function

The system under discussion has three levels for each factor, which are sufficient to quantify its behavior as a quadratic (second order polynomial) function. Low order polynomial response surfaces [15, 55] are popular functions because they fit a variety of scientific designs and are still manageable in terms of complexity. The surrogate model can be described as

$$f(x) = \hat{y} + \epsilon \tag{4.4}$$

where $f(x)$ is the system output, $\hat{y}$ is the surrogate model output, and ϵ is the error between them. A second order polynomial surrogate function is described by:

$$\hat{y} = \beta_0 + \sum_{i=1}^{k} \beta_i x_i + \sum_{i=1}^{k} \sum_{j>i}^{k} \beta_{ij} x_i x_j + \sum_{i=1}^{k} \beta_{ii} x_i^2 \tag{4.5}$$

which expands to

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \beta_{33} x_3^2 \tag{4.6}$$

for our system. Note that there are ten β coefficients associated with a quadratic response surface that has three factors. Assuming that the number of treatments

sampled is n, the system can be described as

$$\mathbf{y} = \boldsymbol{\beta}\mathbf{X} \tag{4.7}$$

where $\mathbf{y}$ is a 1 by n column vector of the measured responses, $\mathbf{X}$ is a n by 10 matrix of the factor levels used to obtain the measured responses, and $\boldsymbol{\beta}$ is the 1 by 10 column vector of the coefficients as shown in equations 4.8, 4.9, and 4.10.

$$\mathbf{X} = \begin{bmatrix} 1 & x_{1\_1} & x_{2\_1} & x_{3\_1} & x_{1\_1}^2 & x_{2\_1}^2 & \dots & x_{1\_1}x_{3\_1} & x_{2\_1}x_{3\_1} \\ 1 & x_{1\_2} & x_{2\_2} & x_{3\_2} & x_{1\_2}^2 & x_{2\_2}^2 & \dots & x_{1\_2}x_{3\_2} & x_{2\_2}x_{3\_2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 1 & x_{1\_(n-1)} & x_{2\_(n-1)} & x_{3\_(n-1)} & x_{1\_(n-1)}^2 & x_{2\_(n-1)}^2 & \dots & x_{1\_(n-1)}x_{3\_(n-1)} & x_{2\_(n-1)}x_{3\_(n-1)} \\ 1 & x_{1\_n} & x_{2\_n} & x_{3\_n} & x_{1\_n}^2 & x_{2\_n}^2 & \dots & x_{1\_n}x_{3\_n} & x_{2\_n}x_{3\_n} \end{bmatrix} \tag{4.8}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_{11} \\ \beta_{22} \\ \beta_{33} \\ \beta_{12} \\ \beta_{13} \\ \beta_{23} \end{bmatrix} \tag{4.9}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} \tag{4.10}$$

Characterizing an application thus means finding the coefficient vector that results in a surrogate surface that fits the real system response the best.

The least squares method of fitting a polynomial model to the observed data minimizes the sum of square residuals $L = \sum_{i=1}^{n} \epsilon_i^2$, where the system residual error is $\epsilon = \mathbf{y} - \hat{\mathbf{y}}$. Based on equations 4.4 and 4.7:

$$L = \sum_{m=1}^{n} \left(\mathbf{y}_m - \beta_0 - \sum_{i=1}^{k} \beta_i x_{i\_m} - \sum_{i=1}^{k} \sum_{j>i}^{k} \beta_{ij} x_{i\_m} x_{j\_m} - \sum_{i=1}^{k} \beta_{ii} x_{i\_m}^2 \right)^2 \tag{4.11}$$

$$= \epsilon^T \epsilon \tag{4.12}$$

$$= (\mathbf{y} - \boldsymbol{\beta}\boldsymbol{X})^T (\mathbf{y} - \boldsymbol{\beta}\boldsymbol{X}) \tag{4.13}$$

To minimize the error L, the derivative with respect to β is taken and set to zero:

$$-2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = 0 \tag{4.14}$$

and solving for β gives

$$\boldsymbol{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y} \tag{4.15}$$

It is important to note that matrix $\mathbf{X}$ stays the same for both response variables (power and throughput). Moreover, it is also not dependent on the running application, but on the sampled configurations, which are the same across running processes. As such, the transpose matrix $\mathbf{X}^T$ is also constant across applications, making vector $\mathbf{y}$ the only term on the right side of equation 4.15 dependent on the application behavior. As such, the computation $\left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T$ can be performed once, offline, regardless of what application is scheduled. The online computation consists of multiplying the resulting 10 by n (where n is the number of sampled treatments) matrix with the n by 1 column vector $\mathbf{y}$. The "Surrogate Surface Fitting" interval in Figure 4.1 consists of this one matrix multiplication and of using the newly computed beta coefficients in equation 4.6 to obtain the remaining (27 - n) response values, thus making the duration of this period insignificant with respect to the full 100ms time quantum.

4.1.3.3 Radial Basis Surrogate Function

The first and second order polynomial functions are non-interpolating: the value of the surrogate function $\hat{\mathbf{y}}$ is not necessarily equal to the value of the real function $f(x)$ at the sampled points. Moreover, fitting a quadratic surface requires at least as many treatment runs as coefficients. To overcome these limitations, we also evaluate an interpolating model that places a radial basis function (RBF) φ at each sampled point [33]. A radial basis function has form $\varphi(||x - c||)$ whose value depends only on the Euclidian distance from the center c. Assuming that there are n samples, there are $\underline{\mathrm{x}}_1, \underline{\mathrm{x}}_2, ..., \underline{\mathrm{x}}_n \in \mathbb{R}^d$ centers, each with its corresponding radial basis function, where $\underline{\mathrm{x}}_n$ are the sampled points in a d-dimensional real space, and d is the dimension of the independent variables (i.e., d=3 beacause there are three factors in our system). Each point $\underline{\mathrm{x}}_n$ is the n^{th} sampled treatment of the three factor levels $(x_{1\_n}, x_{2\_n}, x_{3\_n})$. The interpolating RBF response surface is of the form:

$$\hat{\mathbf{y}} = \sum_{i=1}^{n} \lambda_i \varphi(||\underline{\mathrm{x}} - \underline{\mathrm{x}}_i||) + p(\underline{\mathrm{x}}) \tag{4.16}$$

where λ_i are the coefficients of the response function, $||\cdot||$ is the Euclidean distance between two d-dimensional points, and $p(\underline{\mathrm{x}}) = \mathbf{b}^T \underline{\mathrm{x}} + a$ is a polynomial tail. Without the polynomial tail, the n by n matrix $\mathbf{\Phi}$ with elements $\Phi_{ij} = \varphi(||\underline{\mathrm{x}}_i - \underline{\mathrm{x}}_j||)$ described in Equation 4.20 might become singular. (More information is found in Gutmann's paper on global optimization with radial basis function response surfaces [33].)

We use a cubic radial basis function

$$\varphi(\underline{\mathrm{x}}_{ij}) = \left(||\underline{\mathrm{x}}_i - \underline{\mathrm{x}}_j||\right)^3 \tag{4.17}$$

that needs a linear polynomial tail:

$$p(\underline{x}) = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3 \tag{4.18}$$

Characterizing an application implies obtaining the set of coefficients $\boldsymbol{\lambda}$ and $\mathbf{b}$, which is accomplished by solving the system of equations:

$$\begin{aligned}
y_1(\underline{x}_1) &= \lambda_1\varphi(||\underline{x}_1 - \underline{x}_1||) + \lambda_2\varphi(||\underline{x}_1 - \underline{x}_2||) + \ldots + \lambda_n\varphi(||\underline{x}_1 - \underline{x}_n||) + p(\underline{x}_1) \\
y_2(\underline{x}_2) &= \lambda_1\varphi(||\underline{x}_2 - \underline{x}_1||) + \lambda_2\varphi(||\underline{x}_2 - \underline{x}_2||) + \ldots + \lambda_n\varphi(||\underline{x}_2 - \underline{x}_n||) + p(\underline{x}_2) \\
&\vdots \\
y_n(\underline{x}_n) &= \lambda_1\varphi(||\underline{x}_n - \underline{x}_1||) + \lambda_2\varphi(||\underline{x}_n - \underline{x}_2||) + \ldots + \lambda_n\varphi(||\underline{x}_n - \underline{x}_n||) + p(\underline{x}_n)
\end{aligned} \tag{4.19}$$

where each equation represents the response of one sample: $y_n(\underline{x}_n)$ is either the measured throughput or the measured power of the core configured with pipeline region levels $\underline{x}_n = [x_{1\_n}\ x_{2\_n}\ x_{3\_n}]$. Similar to the quadratic response model, we build a surrogate RBF surface for the throughput response and one for the power response. The methodology to obtain both of them is identical, the only difference being the $y_n(\underline{x}_n)$ values.

If

$$\boldsymbol{\Phi} = \begin{bmatrix} \Phi_{11} & \Phi_{12} & \ldots & \Phi_{1n} \\ \Phi_{21} & \Phi_{22} & \ldots & \Phi_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ \Phi_{n1} & \Phi_{n2} & \ldots & \Phi_{nn} \end{bmatrix} \quad \text{, where} \quad \Phi_{ij} = \varphi(||\underline{x}_i - \underline{x}_j||) \tag{4.20}$$

and

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} \underline{x}_1 & 1 \\ \underline{x}_2 & 1 \\ \vdots & \vdots \\ \underline{x}_n & 1 \end{bmatrix} \qquad \boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix} \qquad \mathbf{c} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_0 \end{bmatrix} \tag{4.21}$$

Then the system of equations 4.19 can be rewritten in contracted form:

$$\begin{bmatrix} \boldsymbol{\Phi} & \mathbf{P} \\ \mathbf{P}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{c} \end{bmatrix} = \begin{bmatrix} \mathbf{Y} \\ \mathbf{0} \end{bmatrix} \tag{4.22}$$

The coefficients of the system are thus:

$$\begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{c} \end{bmatrix} = \underbrace{\begin{bmatrix} \boldsymbol{\Phi} & \mathbf{P} \\ \mathbf{P}^T & \mathbf{0} \end{bmatrix}^T}_{\text{offline computation}} \begin{bmatrix} \mathbf{Y} \\ \mathbf{0} \end{bmatrix} \tag{4.23}$$

Analogous to the second order polynomial response surface methodology, the majority of the computation to obtain the coefficients can be performed offline as noted in Equation 4.23, resulting in extremely fast surface fitting.

4.2 Global Optimization and Runtime Management

Once the running applications are characterized by response surfaces, the system can be optimized according to specific desired targets. Given an N core chip multiprocessor, a per core surrogate function for throughput

$$\hat{T}_{core_i}(x_{1\_core_i}, x_{2\_core_i}, x_{3\_core_i}),$$

and a surrogate function for power

$\hat{P}_{core_i}(x_{1\_core_i}, x_{2\_core_i}, x_{3\_core_i})$,

the hardware resources can be dynamically tailored to the running processes and the system targets. For example, optimal factor levels can be found for a variety of goals:

- Maximizing global fair throughput under a maximum power constraint:

$$max\left(\sqrt[N]{\prod_{i=1}^{N}\hat{T}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})}\right), \quad \sum_{i=1}^{N}\hat{P}_i(x_{1\_i}, x_{2\_i}, x_{3\_i}) < P_{max} \tag{4.24}$$

- Minimizing the power consumption under a certain minimum performance guarantee:

$$min\left(\sum_{i=1}^{N}\hat{P}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})\right), \quad \sqrt[N]{\prod_{i=1}^{N}\hat{T}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})} > T_{min} \tag{4.25}$$

- Maximizing pure throughput under a maximum power constraint:

$$max\left(\sum_{i=1}^{N}\hat{T}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})\right), \quad \sum_{i=1}^{N}\hat{P}_i(x_{1\_i}, x_{2\_i}, x_{3\_i}) < P_{max} \tag{4.26}$$

- Maximizing system throughput while prioritizing certain applications:

$$max\left(\sum_{i=1}^{N}\Big(weight_i * \hat{T}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})\Big)\right), \quad \sum_{i=1}^{N}\hat{P}_i(x_{1\_i}, x_{2\_i}, x_{3\_i}) < P_{max} \tag{4.27}$$

4.2.1 Runtime Manager

Our work focuses on the first optimization goal, maximizing global throughput and fairness under a power constraint. A runtime manager is employed to coordinate the chip-wide effort to reallocate power among the cores to accomplish this

target. The runtime manager collects online information about the application running on each core through sampling, employs the surrogate surface methodology to gather predicted responses for all possible hardware configurations, and then determines what lanes should be deconfigured to meet the power target in the most performance-efficient way. The runtime manager operates at the OS time quantum granularity of 100ms.

There are a number of alternatives for implementing the runtime manager. In order to obtain performance and power statistics and to deconfigure units, the runtime manager requires access to low-level hardware information. Consequently, one option is to implement it as an embedded microcontroller similar to the Foxton Technology Controller included in Intel's Montecito [44]. The advantages of this approach are direct access to hardware and the fast, real-time responsiveness of an on-chip controller. However, implementing the whole manager in hardware would incur the highest die area and hardware complexity costs. Furthermore, it would be the least amenable to upgrades, which may be quite useful if the system optimization targets change. An alternative to a full hardware solution is to dedicate hardware to deconfigure components and gather statistics, and implement the re-allocation logic in software. The optimization and hardware re-allocation algorithms could be incorporated into a low-level hypervisor (supervisor) level thread, or at a higher level as part of the operating system. The main factors dictating the best option would be the ease of implementation, the desire to expose applications to the decision process, and the desired granularity at which the manager should operate.

Each core can be configured in 27 ways, corresponding to all the combinations of the three pipeline regions and their three levels of operation (2, 3, or 4 active lanes).

For a four core CMP, this results in a total number of $27^4 = 531441$ chip-wide combinations. For each of these over half a million combinations, the chip-wide power and global throughput (geometric mean of the predicted responses) must be computed, the combinations that exceed the power budged must be eliminated, and the configuration that maximizes throghput must be chosen. In general, for an N core CMP, the total number of combinations is 27^{N}, making runtime exhaustive exploration of the space impractical. We employ heuristic optimization algorithms to solve the global optimization problem and select a good hardware configuration, even though this configuration might not be the global maximum.

The runtime manager must solve the constrained integer global optimization problem of maximizing CMP performance under a given power budget. The objective function to be maximized should incorporate both performance and fairness. A sum of throughput or arithmetic mean of the throughputs approach results in algorithms that always penalize low IPC benchmarks in order to obtain a not always proportional increase in high-IPC benchmarks. We choose the geometric mean of the throughputs in order to incorporate both metrics into the objective function:

$$f(\vec{x}) = \sqrt[N]{\prod_{i=1}^{N} \hat{T}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})} \tag{4.28}$$

where N is the number of cores, $\vec{x}$ is a vector of size N consisting of the current configuration for each core, and $\hat{T}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})$ is the BIPS of the i^{th} core.

The objective function further has the constraint of meeting a certain power budget, so Deb's constraint handling method [22] is employed to differentiate between feasible (under power budget) and infeasible (over power budget) solutions. This type of constraint handling penalizes configurations that consume more power

than allowed, thus ensuring that infeasible solutions are never chosen over feasible solutions. The final function to be maximized has the form:

$$F(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } g(\vec{x}) <= maxPower \\ 1 - g(\vec{x}) & \text{if } g(\vec{x}) > maxPower \end{cases} \tag{4.29}$$

where $g(\vec{x})$ is the constraint violation function and is defined as the current power consumption of the entire core: $g(\vec{x}) = \sum_{i=1}^{N} \hat{P}_i(x_{1\_i}, x_{2\_i}, x_{3\_i})$.

The solution for the objective function is the vector $\vec{x}$, the configuration of each core that results in the best global performance. The solution vector consists of discrete rather than continuous variables, which makes it difficult to solve the objective function using classical mathematical techniques such as derivative or limit-based methods. Moreover, an integer programming approach is also unsuitable since neither $\hat{T}$ nor $\hat{P}$ are linear functions. Another limiting factor is the need for relatively frequent reevaluation in order to adapt to the dynamically changing behavior of the scheduled running applications.

Heuristic algorithms are attractive due to their efficiency and effectiveness in searching complex and unknown spaces, and their computational performance can be adjusted by limiting the number of objective function evaluations at the expense of solution accuracy. In other words, heuristics can solve difficult problems reasonably well and reasonably fast, with the option of trading off one for the other. A widely used heuristic algorithm is the Genetic Algorithm, which operate by using information gathered from past searches about an unknown space to bias future searches towards more useful subspaces. The next subsections detail the Genetic Algorithm that was modified to suit our objective function and search space.

4.2.2 Integer Coded Genetic Algorithm

The Genetic Algorithm is based on the natural evolution process [35]. Solutions to the optimization problem are coded as chromosomes. A subset of the total possible chromosomes form individuals in a population that evolves towards better solutions through selection (of the fittest members), crossover (recombination of different chromosomes), and random mutation (of chromosomes). The high-level algorithm operation is shown in Figure 4.6.

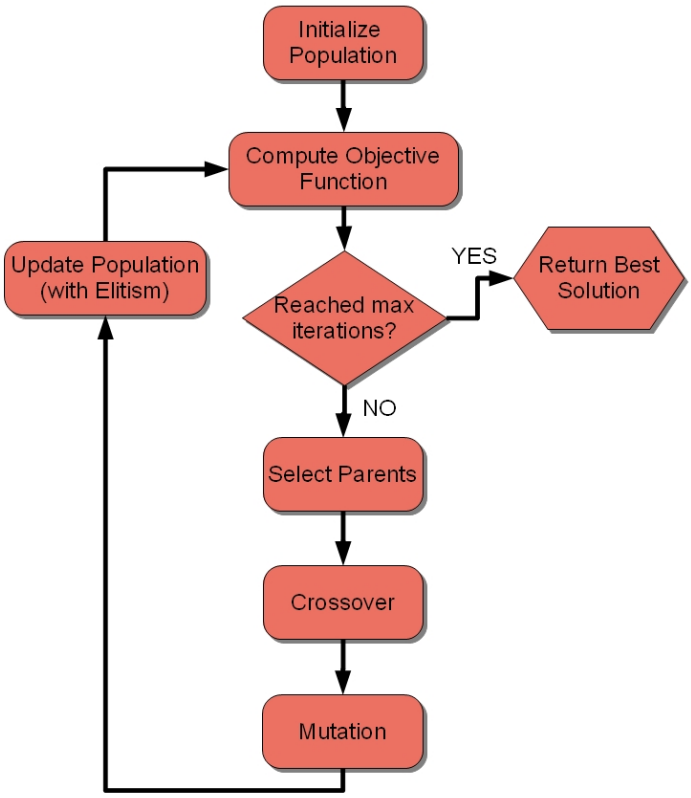


Figure 4.6: Genetic Algorithm.

Encoding: We encode each core configuration as one gene of a chromosome. Each gene can take the integer values 0 to C-1, where C is the number of possible configurations for each core. A combination of N genes form one chromosome (or individual) of a population, where N is the number of cores in the CMP.

Selection: In each generation, a new pool of individuals (children) must be created from the existing set of individuals (parents). In order to pick the mating pool, we used tournament selection with replacement. Two parents are picked at random, and the one with the higher objective function value is selected. The process is repeated again to select the second parent. One pair of parents produces one pair of children through crossover and mutation.

Crossover and Mutation: In order to create two children from two parents, we chose single point crossover at the boundary of the genes and recombined genes from both parents around the crossover point. It is important to note that this crossover mechanism cannot change the values of genes in a chromosome, which means that the configuration of a particular core cannot change from generation to generation. Therefore, we used a high mutation probability to make incremental random changes in the offspring allele values.

Elitism: Due to crossover and mutation, it is possible and quite likely that the best individuals in each generation will not be present in the new generation. In order to prevent the algorithm from losing the best solution found so far, we implement elitism by replacing a random child with the best parent.

Parameters: We empirically explored a variety of parameter values offline and built a desirability function [34, 23] to find parameters that would optimize the algorithm over a variety of power constraints. The parameters obtained were:

a population size of 25 individuals, a crossover probability of 0.9, and a mutation probability of 0.6. We run the simulation for 25 generations (which corresponds to 500 Objective Function evaluations) as a compromise between algorithm accuracy and a computation time of less than 1% of the time quantum for large CMP configurations.

4.3 Results

4.3.1 Methodology

In order to evaluate our modular adaptive technique, we use a highly modified version of the SESC [59] simulator augmented with Wattch [16], Cacti [71], and HotLeakage [78] to model both static and dynamic power consumption. We also modified the simulator to dynamically account for temperature dependent leakage power.

To ensure that the baseline processor core is appropriately sized, we performed an extensive design space study to create a balanced baseline core microarchitecture with the parameters shown in Table 4.2. Deconfiguring a portion of this baseline design results in more than a 10% performance loss for multiple benchmarks.

Note that the runtime manager may underestimate the power costs of turning lanes on, which may cause overshooting the chip-wide power budget. In these cases, which are rare, we model a Global Power Manager that engages DVFS down to meet the power budget.

We use this baseline to model 4, 8, 16, and 32 core CMPs, where each core

Front End	Branch Predictor: gshare + bimodal, 64 entry RAS, 2KB BTB 128 entry ROB, fetch/decode/rename/retire 4-wide
Execution Core	Out-of-order, issue/execute 4-wide 80 Integer Registers, 80 FP Registers, 32 entry Integer Queue 24 entry FP Queue, 32 entry Load Queue, 16 entry Store Queue 4 Integer ALUs, 1 Integer Mult/Div Unit, 1 FP ALU, 1 FP Mult/Div Unit
On-chip Caches	L1 Instruction Cache: 8KB, 2-way, 2 cycle access latency L1 Data Cache: 8KB, 2-way, 2 cycle access latency L2 Cache: 1MB, private, 8-way, 15 cycle latency
Memory	200 cycle latency
Operating Parameters	1V Vdd 4.0 GHz frequency

Table 4.2: Architectural parameters.

runs one of 13 SPEC CPU 2000 benchmarks. We fast-forward each benchmark five billion instructions and run for a total time of 100ms, which models the operating system time quantum. We generate 20 randomly chosen four-benchmark workloads to run on the 4-core CMP, without repeating benchmarks in any given workload. For 8-, 16-, and 32-core CMPs, we randomly repeat some of the benchmarks.

We choose a large number of workloads with random benchmark assignments out of the entire spectrum of applications rather than create a few combinations with one representative benchmark from each category (memory bound, cache bound, computation-intensive) in order to show the actual benefit of our approach under a realistic variety of scenarios.

The Genetic Algorithm was written in C++ and compiled with the "-O3" flag. Due to the stochastic nature of the algorithm, it was run 20 times for each configuration and the results averaged.

In order to evaluate our system under a variety of power cap scenarios, we start with a nominal power of 34.4 V for the four core CMP. This represents the average power consumption of the benchmarks we studied run on a fully provisioned 4 wide

core, multiplied by the number of cores. As the number of cores is scaled up, the nominal power is scaled accordingly. We evaluate our system at 8 different power constraints, corresponding to the range of 90% to 55% of the nominal power.

We compare our system to four different baselines that have similar area and power consumption. We model area for most structures in a core using CACTI 5.3. For fetch and decode logic, we use the transistor count and area estimation tool created by Steinhaus et al. [69]. Area estimates for integer and floating point simple and complex units are estimated based on Gupta et al. [32]. Our results are in line with previous estimations like those presented by Burns et al. [17], who find that a four wide core requires roughly 1.9 times the area of a two wide core. For the 4-core CMP, the four area-matched baselines are:

- A 4-core 4-wide CMP system that shuts down entire cores to meet the power budget.
- An 8-core 2-wide CMP system that shuts down entire cores to meet the power budget. We chose 8 2-wide cores because they have the same area as 4 4-wide cores. However, depending on the applications that are running and the power budget, only a portion of those cores can be turned on.
- A 4-core 4-wide CMP system that engages DVFS to scale down the power consumption until it meets the power budget.
- A 8-core 2-wide CMP system that shuts down half of the cores and engages DVFS by increasing voltage and frequency to boost the single threaded performance of the applications running on the remaining active cores as much as the power budget allows.

We assume three possible schemes for voltage scaling. In the first scheme, the

voltage can be aggresively scaled up and down by at most 40%, from 1V/4GHz to 1.4V/7GHz and 1V/4GHz to 0.6V/1GHz, respectively. However, due to the high power/performance tradeoff of DVFS, we find that our system can scale voltage by at most 30 percent without violated the power budget. In the second scheme, the voltage can be moderately scaled up and down by at most 20%. The third scheme is the most conservative, and assumes that the voltage can be scaled by at most ±15%.

4.3.2 Sampling Accuracy

4.3.2.1 Reducing High Frequency Noise

Effective CMP level optimization depends on the accuracy of the samples, of the surrogate surface, and of the global optimization algorithm. The sampling accuracy is affected by both low frequency and high frequency noise as discussed in Section 4.1.2.3. We identified sample compression and replication as a potential technique for reducing the underlying high frequency noise and extracting the factor level effects on the response of the system. We consider splitting each sample into 1, 2, 4, or 8 groups, reducing the individual sample size in order to maintain the same total sampling stage time. For example, if each sample is 1 ms, replicating the samples twice compresses the individual sample to 0.5 ms. Similarly, replicating the samples eight times reduces the individual sample time to 0.125ms. The total sample phase time for the Box-Behnken design is thus 13 ms, and 9ms for the Fractional Factorial (3MM3) design. Power and throughput responses are measured for each replicate and are combined into one response per sample by averaging their values. We do not consider replicating samples more than eight times

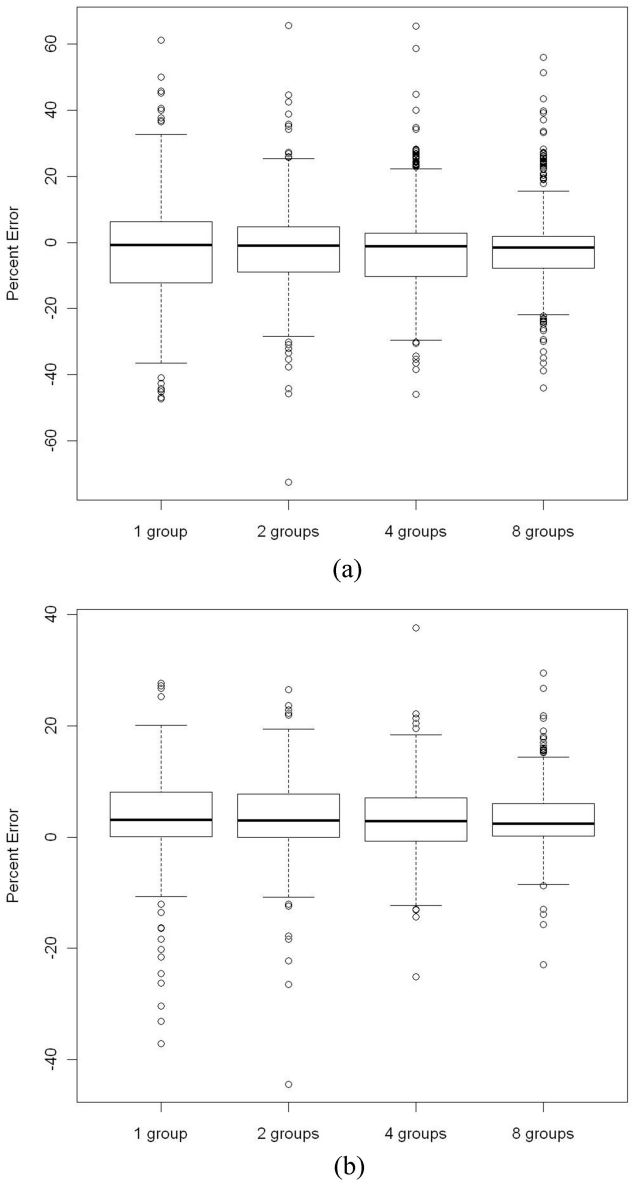


Figure 4.7: Percent error between the real system response (100ms) and the sampled response (1ms) for (a) Throughput and (b) Power. Each 1ms treatment sample is split into 1, 2, 4, or 8 smaller replicates. Statistics are collected across all 17 SPEC CPU2000 benchmarks for the Box-Behnken design.

because the overheads for reconfiguration and response combination become noticeable. Figure 4.7 shows the percent error between the real system response and the sampled response for the range of replicates considered. For each configuration

(treatment) in the design, the error is measured as the percent difference between the response of the treatment if run for the entire operating system time quantum (100ms) and the average response of the treatment if the sum of its replicates is run for a total of 1ms.

There are two opposing effects that must be balanced in order to obtain accurate samples. First, a higher number of replicates reduces low frequency noise by obtaining responses at different points of the benchmark execution. Second, in our infrastructure, many replicates imply smaller samples that are more susceptible to high frequency noise than longer samples. This is due to the fact that longer samples report the response of the system averaged over a longer time period, which is less affected by temporary microarchitectural and software events (such misses, mispredictions, or small software loops). The study in Figure 4.7 shows that for both responses (throughput and power), samples become more accurate with increasing the number of replicates, as shown by the reduced spread of the 25^{th} and 75^{th} percentiles and of the outliers. The mean error across all four replicate options hovers around zero, indicating that, on average, samples are accurately capturing the true response of the system. Unless otherwise noted, we use sampling with eight replicates throughout the remainder of the system analysis.

4.3.2.2 Sample Interval Size

The individual sample size determines the duration of the sampling phase, which in turn determines the size of the steady interval, or the amount of time for which the optimal solution is run. Ideally, samples should be as short as possible, to reduce the time spent sampling suboptimal configurations and increase the time spent running the optimal configuration. However, smaller samples are more susceptible

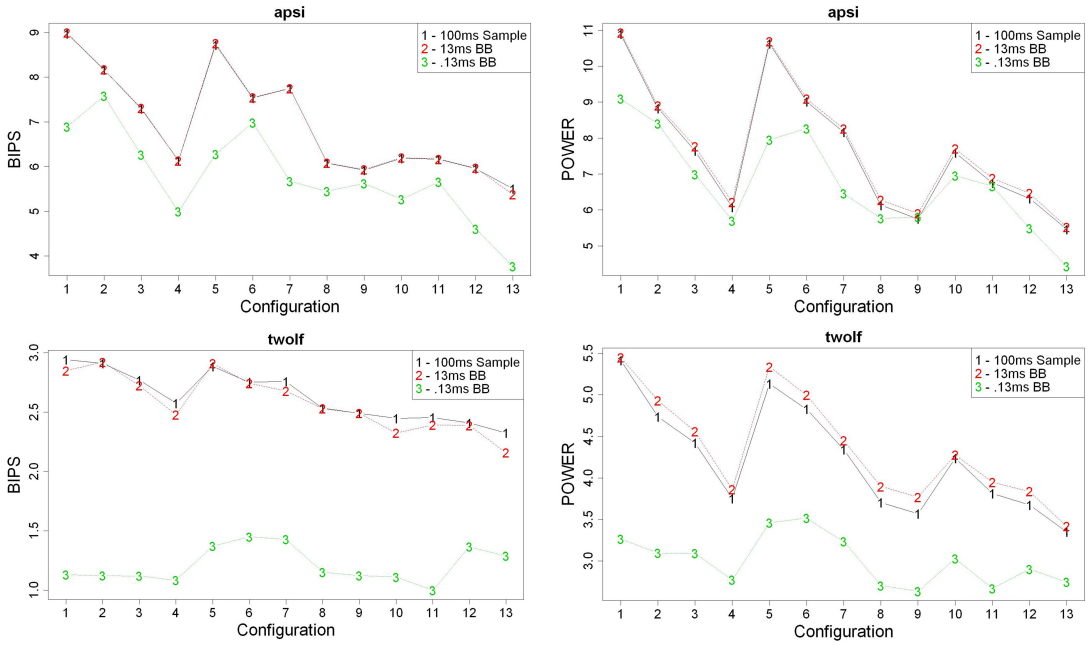


Figure 4.8: Effect of sample size on system responses (13 Box-Behnken treatments) for apsi and twolf benchmarks. The black line labeled "1" is the real system response. The red line labeled "2" is the sampled response for 1 ms samples split into 8 replicates. The green line labeled "3" is the sampled response for 0.1ms samples split into 8 replicates.

to temporary hardware and software events as described above. We perform a study to measure the sensitivity of the system responses to sample size, and show the results for the maximum (13 ms) and minimum (0.13 ms) sampling phase durations considered (Figures 4.8 and 4.9).

The vast majority of the benchmarks we studied behave similarly to the two benchmarks presented in Figure 4.8, which shows the real (100ms) and sampled responses for the Box-Behnken design for apsi (top) and twolf (bottom). The real response is labeled "1" and the sample responses are labeled "2" and "3" for 1ms and 0.1 ms individual samples, respectively. The y-axis represents the measured response in billion instructions per second for throughput (left), and the measured response in Watts for power (right). The x-asis represents the 13

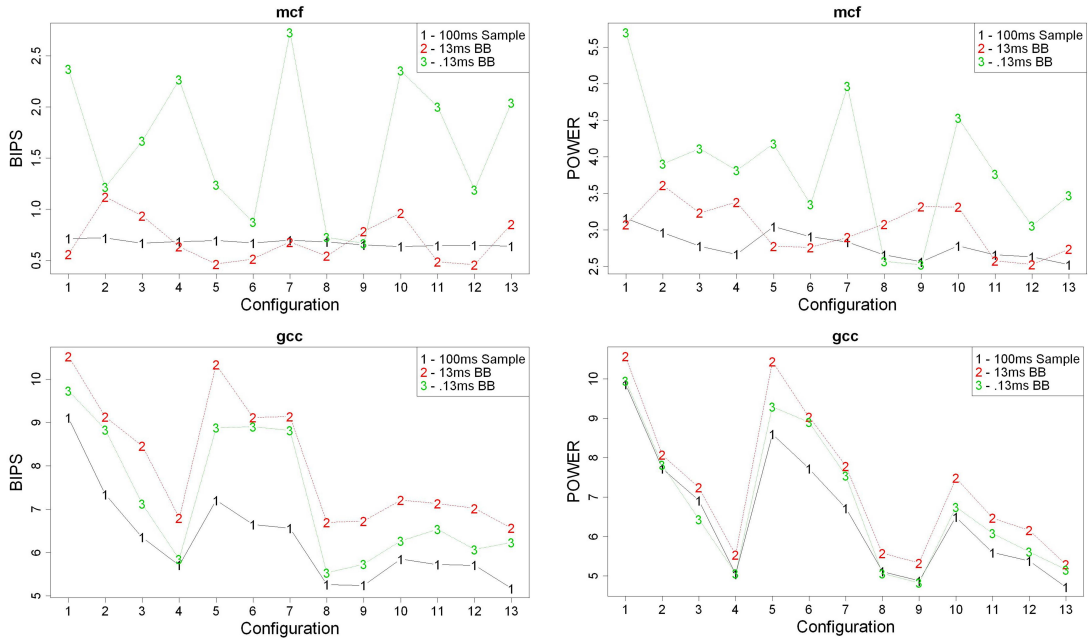


Figure 4.9: Effect of sample size on system responses (13 Box-Behnken treatments) for mcf and gcc benchmarks. The black line labeled "1" is the real system response. The red line labeled "2" is the sampled response for 1 ms samples split into 8 replicates. The green line labeled "3" is the sampled response for 0.1ms samples split into 8 replicates.

configurations sampled with the Box-Behnken design. The longer 1 ms samples consistently outperform the small samples. For benchmarks like applu, the 1 ms samples perfectly match the real system response, and the 0.1 ms samples have tolerable errors. However, many benchmarks exhibit behavior similar to twolf. The 1ms samples almost perfectly match the real system responses, but the 0.1 ms samples grossly underestimate the real responses.

Figure 4.9 shows two atypical but interesting special cases, corresponding to benchmarks mcf (top) and gcc (bottom). For mcf, neither the 13 ms, nor the 0.13 ms sampling phase is representative of the real system behavior. The small samples are so inaccurate because they are greatly susceptible to mcf's memory behavior. The longer samples are less susceptible because they average the behavior over

longer intervals, and look like they match the data relatively well. In reality, the longer samples are almost as impractical as the small samples due to the fact that the samples do not preserve the relative response trend between different configurations. For example, the configuration with the best real response is shown to produce only the fourth best sampled response, and the configuration with the worst real response produces one of the best sampled responses. We discovered that samples that do not preserve trends between configurations greatly reduce the efficiency of the optimization algorithm, even if their error seems manageable.

Another atypical behavior is that of gcc (Figure 4.9 bottom). In this case, both sample interval sizes are inaccurate, but they both preserve the relationship between the different configurations well. Moreover, they both overestimate throughput and power, which we discover is not that problematic. The bigger problem is underestimation, as it leads to power violations. The most interesting point is that the small 0.13ms sampling phase performs better than the longer 13ms sampling phase. Gcc is well known for having an erratic performance behavior for the reference inputs. Put another way, gcc exhibits very fine grained phase changes. Small samples are more prone to fall within these fine grained phases than the longer 1ms samples, which are likely straddling across phase changes.

The value of the response surface methodology is limited by the inaccuracy of the samples, since they are the basis for response prediction at points that are not sampled. The error of small samples is perpetuated in the response surface fitting interval, resulting in poor optimization choices. To demonstrate this point, we compiled the error of the predicted responses across all 17 studied benchmarks for both the quadratic and RBF response surfaces (Figure 4.10). Small 0.1ms samples (0.13ms total sampling interval) result in prediction errors in excess of

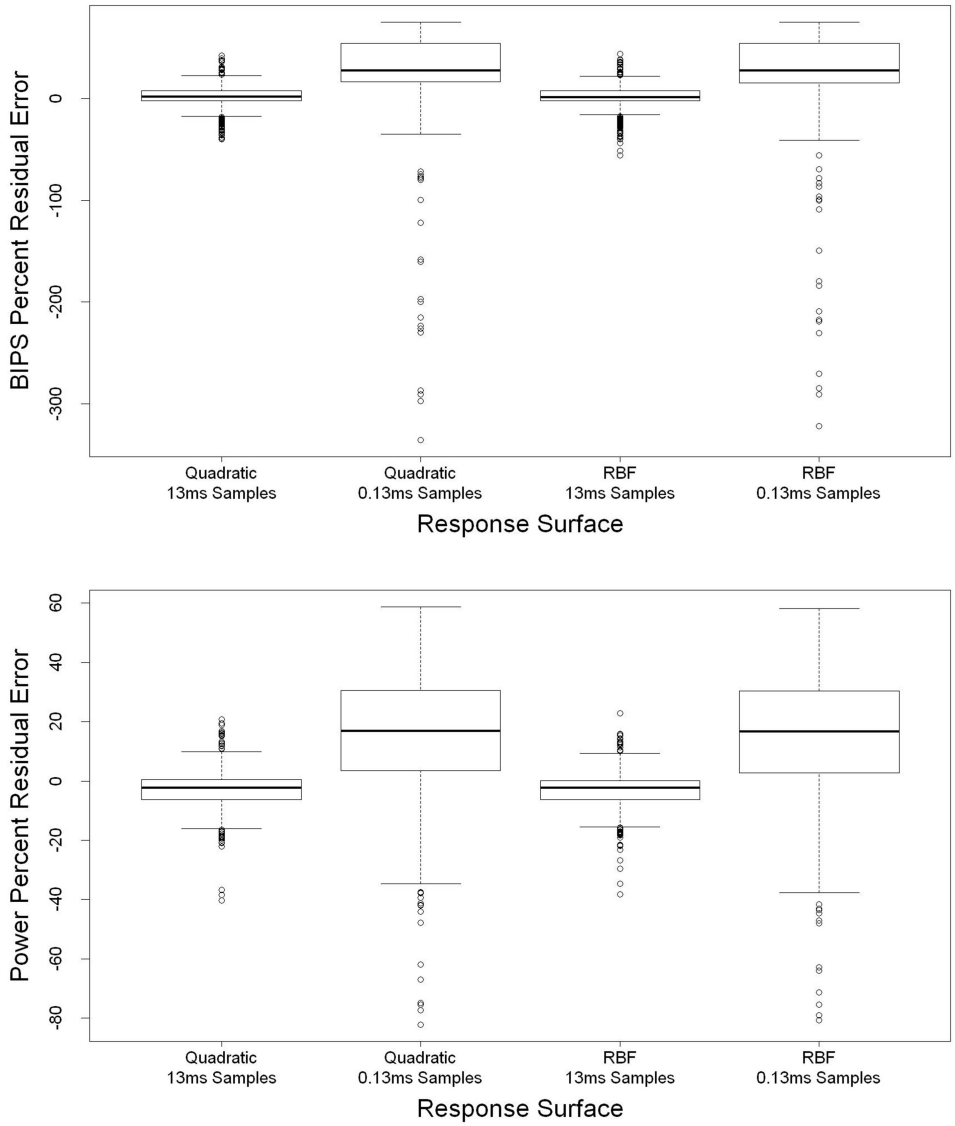


Figure 4.10: Effect of sample size on system responses across all 17 benchmarks, showing statistics for the percent error between the real system response and the predicted responses based on small (0.1ms) and long (1ms) samples. The surrogate models are built on the Box-Behnken design, resulting in a total sampling interval of 13ms, and 0.13ms, respectively.

300%, with the average hovering around 10% for throughput and 20% for power, while 1ms samples have a much tighter distribution and significantly fewer outliers. Longer sampling intervals are more accurate, but reduce the steady interval size to

unacceptable levels. For example, 2 ms individual samples result in an sampling interval of 26 ms for the Box-Behnken design, or almost 30% of the OS time quantum. 4 ms samples reduce the steady interval to a little over 50% of the OS time quantum. To strike a balance between sampling interval size and accuracy, we use 1ms samples for the remainder of our analysis.

4.3.3 Response Surfaces

Figure 4.11 shows the accuracy of the three surrogate surfaces considered for characterizing application throughput (top) and power (bottom). The y-axis represents the percentage by which the predicted responses deviate from the real responses, and each box plot depicts statistics collected across the 17 benchmarks studied. For each application, a response surface was built on the full factorial sampling design (27 possible combinations of active lanes in the three pipeline regions), on the Box-Behnken design (13 samples according to Figure 4.2), or on the Mod 3 based Fractional Factorial design (9 samples, denoted as 3MM3). We generated the figure using real responses (as opposed to sampled responses) for the observation points, in order to extract the underlying shape of the response functions and separate sampling effects from surrogate surface fitting effects.

The linear model fits the data the worst, despite being built using the full factorial configurations. The residual percent error can be as high as 20% in either direction, meaning that responses are both over and under estimated. As expected, this suggests that the throughput and power follow a non linear relationship with hardware configurations.

The second data point represents the distribution of the errors associated with

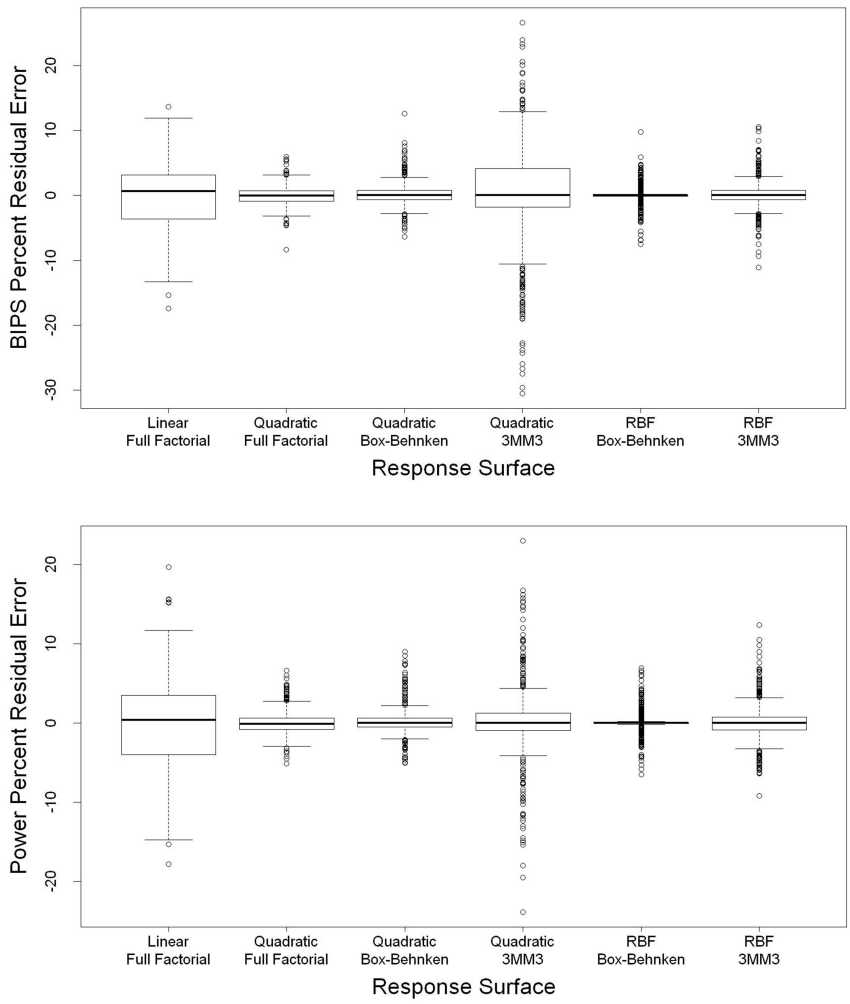


Figure 4.11: Surrogate model accuracy measured as percent residual error between the predicted and real (100ms) system responses: throughput (top), and power (bottom).

fitting a quadratic surface to the full factorial observations. The error is drastically reduced for both throughput and power, as shown by the 25^{th} and 75^{th} percentiles having less than 2% error, and the outliers in the 5% error range. Even for a reduced number of observations (Box-Behnken design), a second order polynomial matches the responses well (third data point). The bulk of the prediction errors stay the same, with a slight increase in the number of outliers. The prediction accuracy drops dramatically when building the quadratic surface on only nine

observation points (3MM3 Design). Recall that a quadratic response surface for three variables has the form

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \beta_{33} x_3^2 \quad , \tag{4.30}$$

thus requiring 10 coefficients, which cannot be obtained using only nine samples. We can eliminate the last term of Equation 4.30 and lose the squared effects of variable x_3. However, the matrix $\mathbf{X}^T\mathbf{X}$ becomes singular, thus noninverting. As such, one more term needs to be eliminated from the quadratic formula, which effectively reduces the quadratic function to an almost linear one. We characterize applications at runtime using the quadratic surrogate surface built on the 13 Box-Behnken sampling points.

The radial basis function response surface is an interpolating model. If the full factorial design is used to build the RBF surface, the residual error is zero and as such is not shown in Figure 4.11. Using only 13 Box-Behnken observation points to create an RBF surrogate surface still results in extremely accurate results, with almost no spread and a relatively small number of outliers. Reducing the number of observation points even further (3MM3) degrades the accuracy slightly, but still maintains results as good as the quadratic surface built on 13 observations.

The effectiveness of runtime application characterization is contingent on the combination of an accurate response surface and accurate observation points (samples). Figure 4.12 shows two typical online system responses compared to the actual behavior of those configurations on the entire 100ms time quantum (the points labeled "1 - 100ms Sample"). The data series labeled "2 - 13ms BB, Quad"

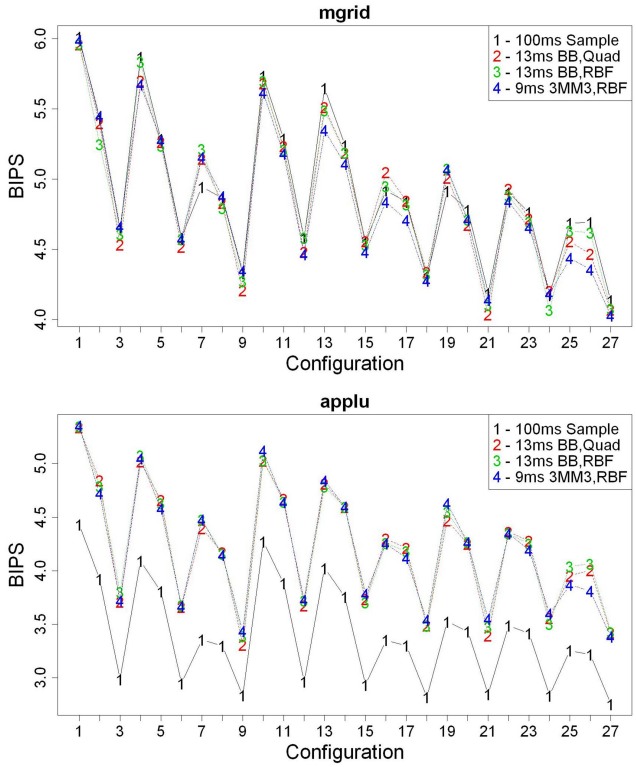


Figure 4.12: Surrogate surface predictions for benchmarks mgrid (top) and applu (bottom). The line labeled "1" is the actual response of the system; the lines labeled "2" and "3" correspond to predictions based on the quadratic and RBF surfaces using the Box-Behnken design, respectively; the line labeled "4" corresponds to predictions based on the RBF surface that uses the fractional factorial 3MM3 design.

corresponds to a quadratic surface fit on 13 online samples, with the other 14 responses predicted by the surrogate. Similarly, the data series labeled "3 - 13ms BB, RBF" shows the samples and the predictions obtained from an RBF surface fit on Box-Behnken samples. Finally, the series denoted as "4 - 9ms 3MM3, RBF" shows nine Fractional Factorial samples and 18 predictions of the corresponding RBF surrogate surface. For most of the benchmarks (nine out of 17 benchmarks), all three surrogate surface methodologies are very accurate as shown by the BIPS response for mgrid. For other benchmarks such as applu, the surrogate surfaces

accurately predict the system response based on the information obtained from the sampling period, but the samples themselves are noisy. Note that the relationship between the different configurations is preserved correctly, even though the absolute values are predicted slightly higher.

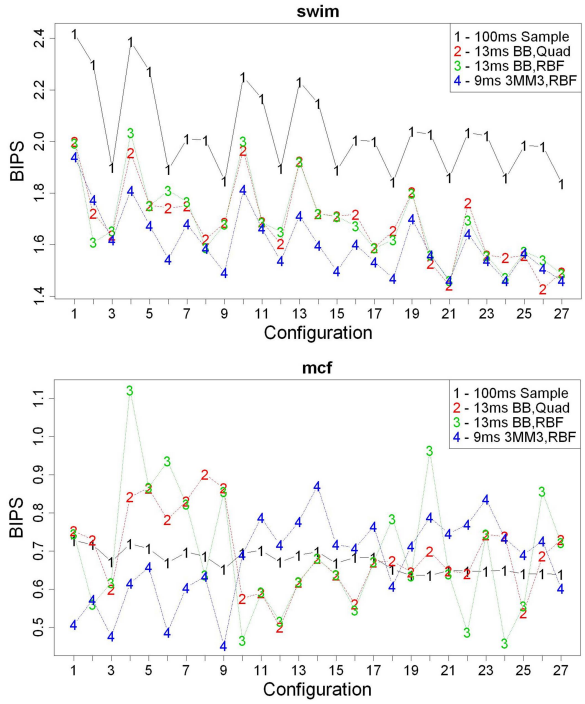


Figure 4.13: Surrogate surface predictions for benchmarks swim (top) and mcf (bottom). The line labeled "1" is the actual response of the system; the lines labeled "2" and "3" correspond to predictions based on the quadratic and RBF surfaces using the Box-Behnken design, respectively; the line labeled "4" corresponds to predictions based on the RBF surface that uses the fractional factorial 3MM3 design.

Similarly, some benchmark behavior is predicted lower than the real system response as depicted for swim in Figure 4.13 (top). Note that the RBF surface built on the fractional factorial design preserves the relationship between configurations better than the surfaces built on the Box-Behnken design, even though the absolute predicted values are slightly more inaccurate. Both surfaces build

on the Box-Behnken design behave almost the same because the difference in response surface accuracy is masked by the larger sampling inaccuracy. From the six benchmarks that fall in this category, applu, swim, and gcc exhibit the biggest discrepancy between the predicted and real response. Lastly, the behavior of only two benchmarks (mcf and parser) is poorly characterized using the combined sampling and surrogate surface technique (Figure 4.13 (bottom)).

4.3.4 Optimization Results

For a four core CMP, implementing an algorithm that exhaustively searches the entire combinatorial space and picks the best performing configuration under the power constraint is computationally feasible. Such an algorithm executes in roughly 80ms, which makes it impractical for deployment in an online system, but provides an upper bound on the expected system optimization results. We use the Genetic Algorithm discussed in Section 4.2.2 run on the surrogate surface predictions to search the global, CMP-level combinations of N-core configurations and limit its runtime by setting the number of generations to 25. The best core-level configurations (active lanes) that maximize the global objective function found in the 25 generations are run during the remaining steady interval.

Figure 4.14 shows how close the solution found by the online Genetic Algorithm (blue bars) is to the global maximum found by the exhaustive algorithm. The results are normalized to the best solution found by the exhaustive search at each of the eight power constraints shown on the x-axis. The best solution found by the Genetic Algorithm is within at least 4% of the global maximum across all power constraints. The online solution gets progressively closer to the global maximum as the power constraints are relaxed from 55% to 90% of the nominal power.

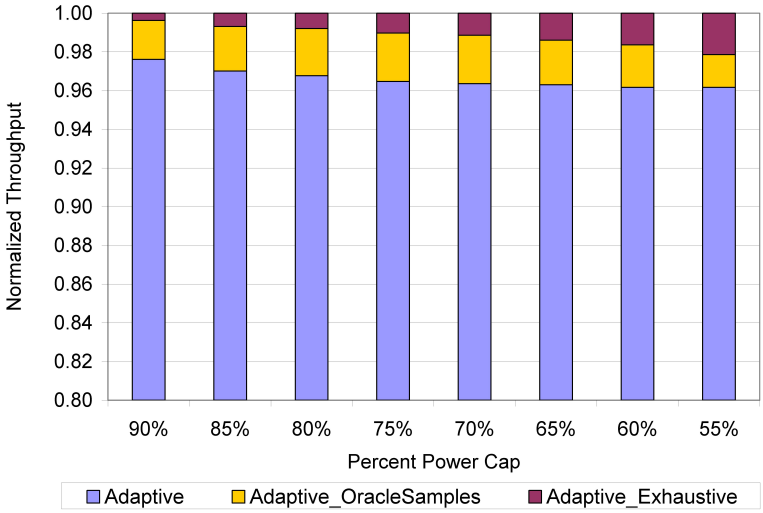


Figure 4.14: Comparison of exhaustive optimization algorithm, Genetic Algorithm based on oracle samples, and Genetic Algorithm running on top of the quadratic response surface with Box-Behnken samples.

This is due to the fact that at very strict power constraints, most individuals in a generation's population are infeasible, making it difficult to select good parents to create a new generation. At relaxed power constraints, most individuals in each generation are feasible, and the fitness value can be effectively used to generate parents that are likely to produce good offspring. Also shown in the graph are results obtained by a Genetic Algorithm run on the true 100ms power and throughput responses (denoted as Adaptive_OracleSamples) to see how much potential performance benefits are lost through sampling and response predictions. The results are within at least 2% of the global maximum across all power constraints, which implies that our online approach of characterizing applications is very effective. Note that the difference between GA using online and oracle samples decreases with stricter power constraints, implying that the Genetic Algorithm itself and not the online sampling and response approximations are at fault.

As the number of cores in the CMP system is increased, it is no longer compu-

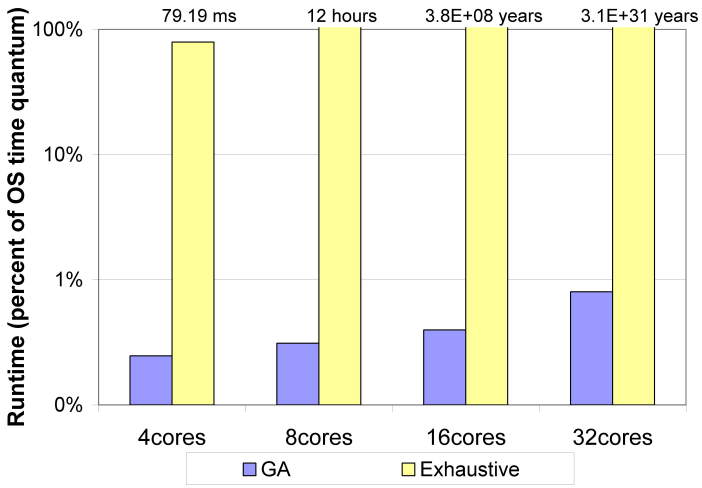


Figure 4.15: Runtime of the Genetic Algorithm with 25 generations expressed as a percentage of the OS time quantum (logarithmic scale). Runtime of the exhaustive search is shown in absolute numbers.

tationally feasible to compute an offline exhaustive search algorithm. Figure 4.15 shows the runtimes of the exhaustive algorithms and of the online Genetic Algorithm limited to 25 iterations. As the number of cores is increased, the combinatorial exploration space explodes, making it impossible to compute the real global maximum. In order to obtain an upper bound for 8, 16, and 32 cores, we implement an offline Genetic Algorithm that uses oracle "sample" values and is run for 200 iterations as discussed in more detail in Section 4.3.5.3. The runtime of the online Genetic Algorithm is managed by restricting the number of iterations to 25, which limits the execution time to at most 1% of the OS time quantum in a 32 core CMP system. The runtime of GA increases with the number of cores even though the number of generations remains constant, because the chip-wide throughput and power are calculated for each individual of a population, and they depend on the number of cores in the system.

For the remainder of this chapter, we report the performance and power of the

CMP during the entire OS time quantum (including the sampling and optimization intervals). We conservatively assume a "dead" time of 1ms (or 1% of the OS time slice) where no useful instructions are executed to account for the optimization time.

4.3.5 System Level Results

4.3.5.1 Single Threaded Performance

In order to evaluate the single-threaded performance of our technique, we compare a modular lane-based core with two static designs, a 4-wide core and a 2-wide core, that employ DVFS (either up or down) to match a certain power budget. A 2-wide core has DVFS enabled and operates at a higher voltage and frequency to match the 4-wide power. Figures 4.16, 4.17, and 4.18 show the throughput improvement of our adaptive core over the 2-wide core under the three dynamic voltage and frequency scenarios described in the Methodology section at three power budgets. The power budgets represent 90%, 75%, and 55% of the total power consumed by each application running on a fully provisioned 4-wide core. At the 90% power budget, an adaptive lane-based core obtains average improvements of 14.4%, 19.1%, and 25.1% over 2-wide cores employing aggressive, moderate, and conservative DVFS, respectively. This is due to the fact that for most sequential applications, issue width and implicit ILP exploitation is more power efficient than increasing frequency and voltage, which has an exponential effect on power.

As the power budget is reduced, a 2-wide core employing DVFS up becomes more efficient because voltage and frequency need not be scaled up by large amounts. At the lowest power budget (55% power cap), a 2-wide core without

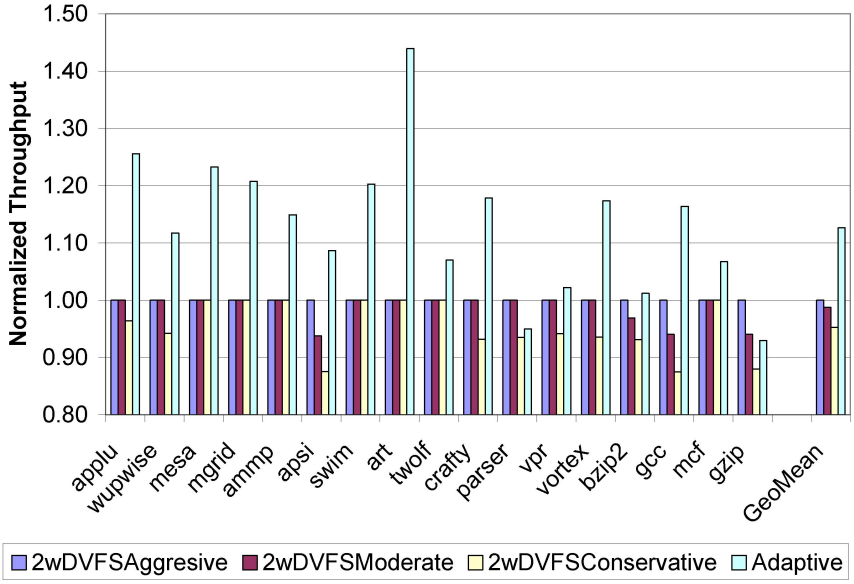


Figure 4.16: Throughput improvement over a single 2-wide core with aggressive, moderate, and conservative DVFS at 90% power cap. Throughput normalized with respect to a 2-wide core with aggressive DVFS.

voltage and frequency scaling (as well as an adaptive core with two lanes active in every pipeline region) consumes more power than the power cap for some applications. In those cases, a 2-wide core is forced to employ DVFS down, and an adaptive core is forced to shut off additional lanes, becoming scalar in some pipeline regions. Because voltage and frequency can be scaled down slightly for high power savings, the 2-wide core performs better than adaptive when running applications that had to scale down to one lane in some pipeline regions (see mgrid, ammp, art, swim in Figure 4.18). One advantage of adaptive cores is that when they are deployed in a system with other adaptive cores, they do not need to scale down the hardware to one active lane (which incurs a high performance penalty) because they effectively "steal" power from other cores that have a worse power-performance tradeoff. Adaptive cores are not meant to replace existing go-to techniques when they are effective. Rather, they can be used to complement

any existing technique when it fails to provide benefits. Depending on what applications are running and the power constraints, one can choose whether to engage DVFS or shut down lanes in order to obtain the best performance.

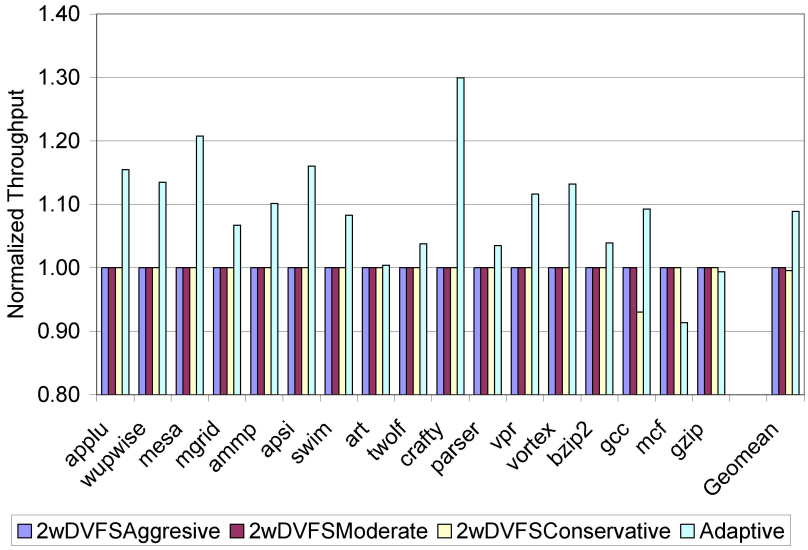


Figure 4.17: Throughput improvement over a single 2-wide core with aggressive, moderate, and conservative DVFS at 75% power cap. Throughput normalized with respect to a 2-wide core with aggressive DVFS.

We now compare an adaptive lane-based core with a static 4-wide core. As the power budget is reduced, an adaptive lane-based core shuts down lanes, while a static 4-wide core must have DVFS enabled and thus operates at a lower voltage and frequency to consume the same reduced power. Figures 4.19, 4.20, and 4.21 show the improvement in performance of the adaptive core over the static 4-wide core employing DVFS down for three power constraints (90%, 75%, and 55%). All the results are normalized with respect to the throughput of the static 4-wide core with aggressive dynamic voltage and frequency scaling. At relaxed power budgets (90% power cap), adaptive cores perform on average about the same as 4-wide cores (Figure 4.19).

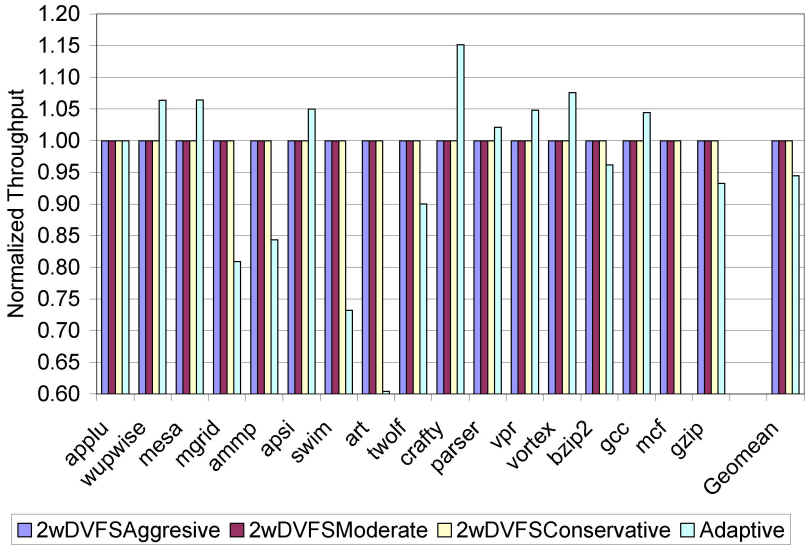


Figure 4.18: Throughput improvement over a single 2-wide core with aggressive, moderate, and conservative DVFS at 55% power cap. Throughput normalized with respect to a 2-wide core with aggressive DVFS.

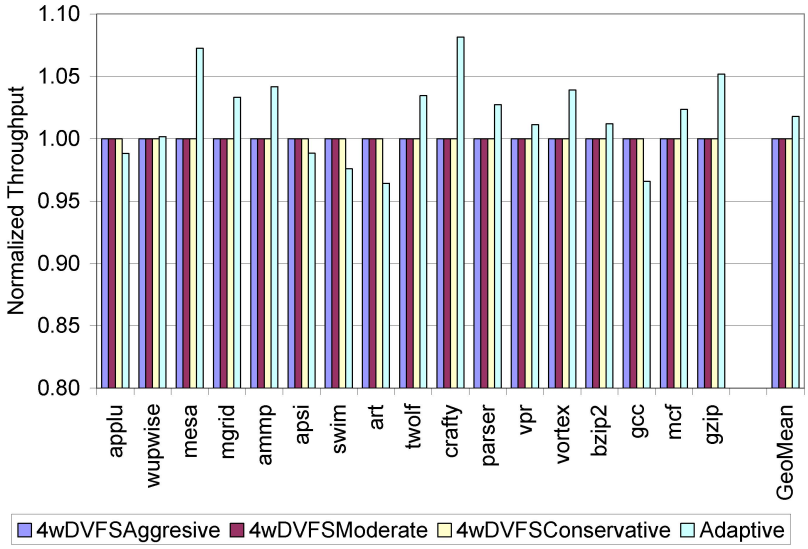


Figure 4.19: Throughput improvement over a single 4-wide core with aggressive DVFS at 90% power cap. Throughput normalized with respect to a 4-wide core with aggressive DVFS.

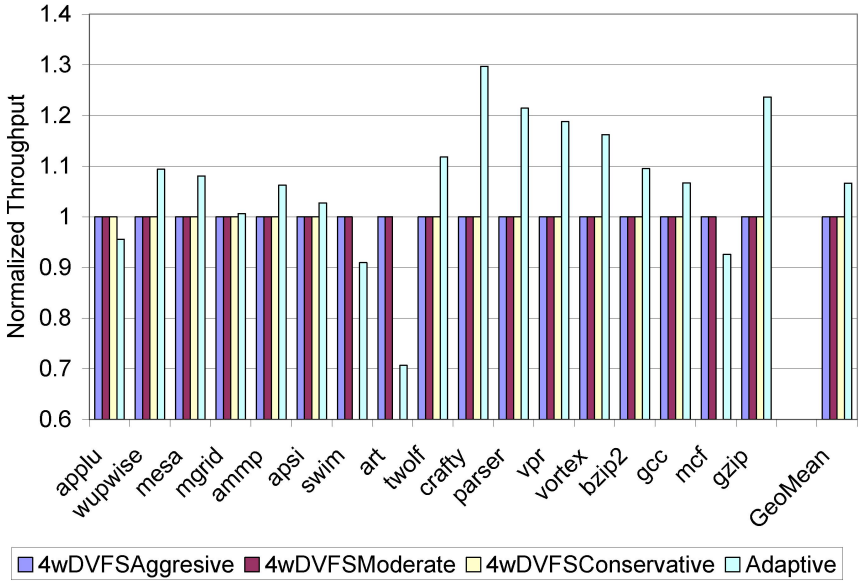


Figure 4.20: Throughput improvement over a single 4-wide core with aggressive DVFS at 75% power cap.

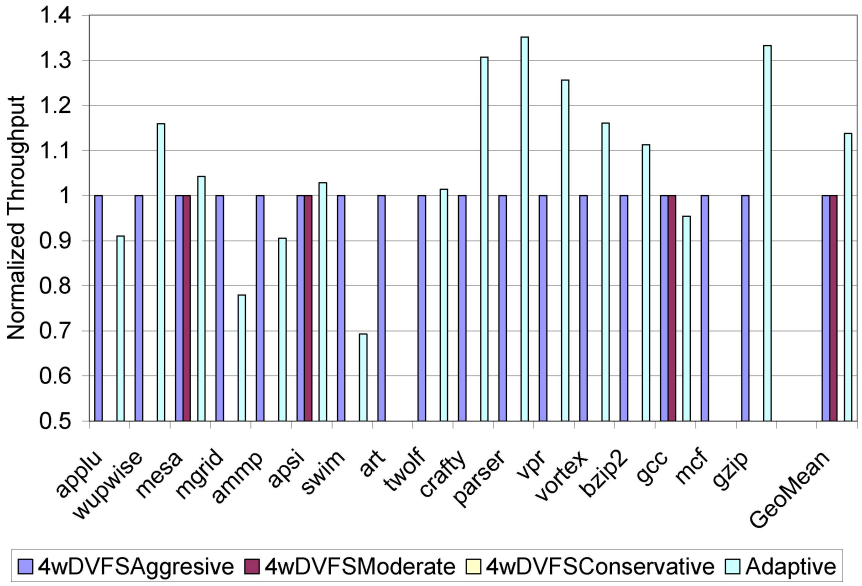


Figure 4.21: Throughput improvement over a single 4-wide core with aggressive DVFS at 55% power cap.

If power is constrained further to 75%, our technique shows modest average improvements of 6.7% when the 4-wide cores uses the aggressive voltage scaling

technique, with the static 4-wide core performing better for almost 30% of the workloads. When voltage and frequency are scaled down even slightly, tremendous amounts of both static and dynamic power are saved across the entire chip. Static power is saved because it is dependent on the difference between Vdd and Vth (the transistor threshold voltage) and dynamic power is saved proportionally to $f \cdot Vdd^2$. In contrast, our adaptive technique can only linearly save the static and dynamic power of the lanes that have been deconfigured. However, a 4-wide core cannot meet the lowest power budget for three out of the four workloads where aggressive DVFS performed better if the voltage can be dynamically scaled conservatively. At very stringent power constraints (55% power cap), aggressive DVFS outperforms adaptive cores for a little under half the applications, but overall is outperformed by 13.7%. In reality, it is predicted that aggressive voltage and frequency scaling will not be feasible. A more realistic scenario is modeled by the moderate DVFS, which cannot meet the lowest power budget for three quarters of the applications. If voltage and frequency can only be conservatively scaled, a static 4-wide core cannot meet the lowest power budget for any of the benchmarks. Moreover, DVFS up or down can be applied at any point on top of our adaptive core to push the power limits much further than DVFS alone.

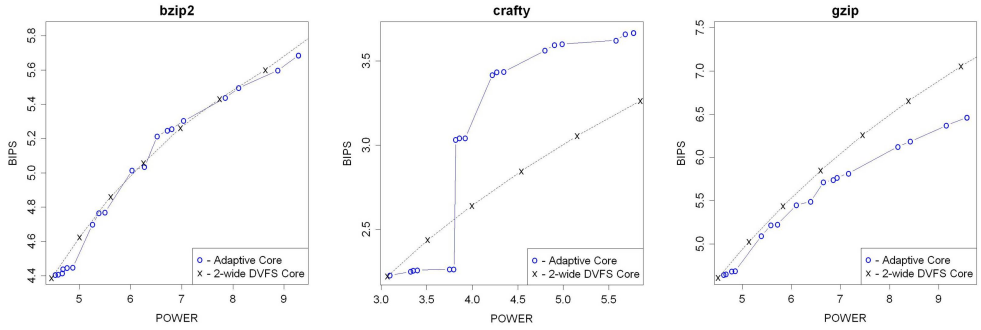


Figure 4.22: Power-performance tradeoffs for an adaptive core versus a 2-wide core with DVFS up.

Figures 4.22 and 4.23 show typical power-performance tradeoff curves for adaptive and DVFS on a 4-wide and 2-wide core. The "Adaptive Core" line was obtained by eliminating the pareto-dominated configurations out of the total 27, then sorting the remaining configurations in decreasing throughput order. The steeper the slope, the better, since that implies large improvements in throughput for very small increases in power consumption. In both figures, the leftmost application is a typical application that benefits the same from lane deconfiguration and from DVFS. The middle application shows typical behavior for applications where DVFS is much less cost effective than adaptive, and the rightmost figure shows applications for which DVFS is more attractive. Since there is a lot of variation among benchmarks, an interesting direction for future research is to dynamically choose between engaging lane-based optimization and DVFS. Depending on the slope of the power-performance tradeoff curves, DVFS can be engaged up on a lane-based core with the minimum amount of active lanes, or engaged down on a lane-based core with the maximum amount of active lanes. As a matter of fact, rather than arbitrarily choosing the biggest or smallest lane-base configuration, DVFS could be engaged on the most power-performance efficient lane-based configuration for the running application.

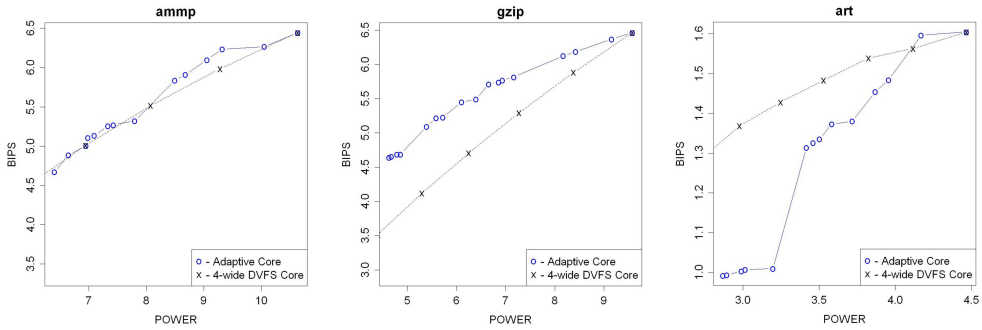


Figure 4.23: Power-performance tradeoffs for an adaptive core versus a 4-wide core with DVFS down.

4.3.5.2 Multiprogrammed Workload Performance

While our approach shows good improvements in single-threaded performance, its full potential is realized in systems with multiple cores. For such systems, the adaptive lane-based approach is able to effectively "steal" power from the cores that do not need their fully provisioned resources while maintaining acceptable performance levels, and redistribute it to resource hungry cores to increase their performance well beyond their maximum level in static designs.

In order to evaluate the system level performance of our adaptive technique against the static baselines, we create 20 multiprogrammed workloads from the suite of SPEC CPU2000 benchmarks. Each multiprogrammed workload consists of four randomly chosen sequential applications, and the global, or system-level throughput is calculated by taking the geometric mean of the cores' BIPS metric. This approach ensures that low-IPC applications are not penalized in favor of high-IPC ones.

Turning off entire cores

Standard static CMPs are not inherently designed to operate at different power constraints, but one approach is to turn off entire cores until the chip-wide power budget is met. One significant downside to this technique is the fact that the operating system needs to change its scheduling policies to only schedule as many applications as there are cores. It is also difficult to compare the performance of two systems that do not have the same number of cores. One approach is to compute the sum of throughputs as if there were enough cores to run all applications, and scale that sum by the number of cores that are actually on. However, this approach does not take into consideration the single-threaded performance of the running

applications, and at the same time does not insure that all benchmarks are weighed fairly in the reported system throughput. We choose to normalize the throughput of each of the applications to their individual performance on a fully provisioned 4-wide core, and scale the average system throughput by the number of active cores. This ensures fairness to some degree, since all the normalized throughputs are in the same scale, but does not capture the latency degradation in single-threaded performance. For example, let's assume there are two designs, A and B, the first with N cores and the second with N/2 cores. If design A degrades the performance of every application by a half, then the normalized throughput average for each core is 0.5. At the system level, the throughput of system A is $N \cdot 0.5$ and that of system B is $\frac{N}{2} * 1$, resulting in the same throughput for both systems, even though every application on system A will finish in double the amount of time it does on system B.

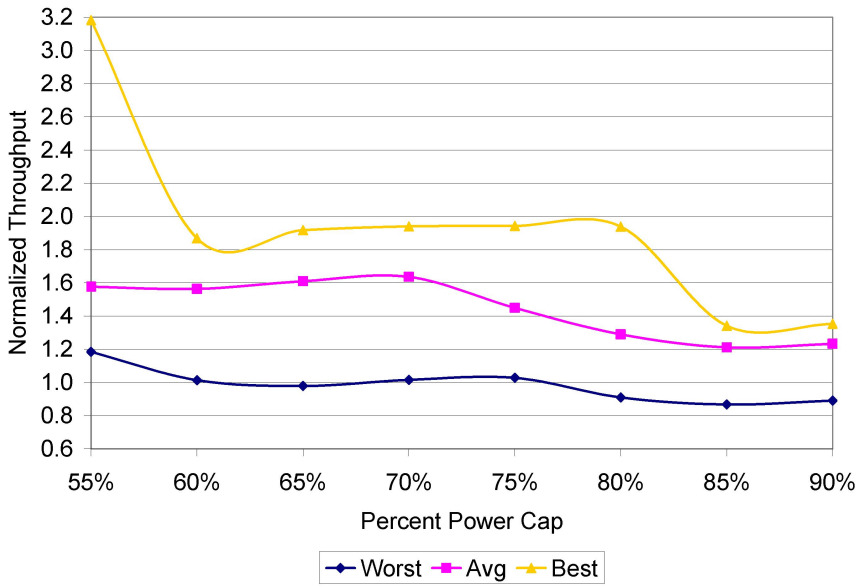


Figure 4.24: Best, average, and worst global throughput improvement over a CMP system with 4 4-wide cores that shuts down cores to meet the power budget.

Figure 4.24 shows the improvement of a 4-core lane-based adaptive CMP over a system that employs four 4-wide cores and sequentially turns off cores until the power budget is met. The baseline must turn off resources at a very coarse granularity, which negatively impacts its performance. We show the performance of the best, average, and worst of the 20 multiprogrammed workloads across a wide range of power constraints. For low power constraints, two of the four cores must be turned off in order to meet the power budget, effectively halving the system performance. In one of the 20 workloads comprising all high-IPC, power-hungry benchmarks, the system had to shut down three of its four cores, drastically reducing its performance. On the other hand, our adaptive CMP with the smallest number of lanes turned on was able to sustain a performance more than 3 times higher than the statically designed system. At high power levels, it is possible and likely that workloads consisting entirely of low-power applications can fit well below the power budget, eliminating the need to shut cores off. In such cases, the adaptive technique performs worse than the 4-wide system due to the overhead associated with sampling and application characterization. On average, our technique performs significantly better than the baseline across all power budgets, with average improvements ranging from 23.3% to 63.7%.

An analogous analysis comparing a lane-based 4 core CMP to a system with eight 2-wide cores that shuts down cores to meet the power target is shown in Figure 4.25. As opposed to the previous results, the static baseline outperforms our architecture according to the metric selected for evaluation by an average of 13% at relaxed power constraints. Since the metric does not capture single threaded performance, the sheer volume of small cores makes up for the reduced functionality in each core. However, as the power goal becomes more restrictive, the adaptive architecture starts outperforming the 2-wide CMP system by as much

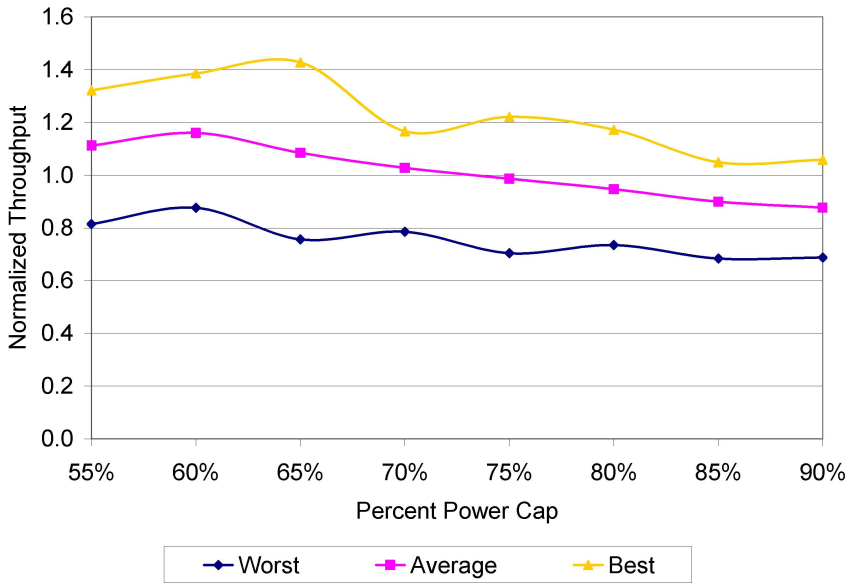


Figure 4.25: Best, average, and worst global throughput improvement over a CMP system with 8 2-wide cores that shuts down cores to meet the power budget.

as 16% at the 60% power cap across all 20 workloads. In the best case, performance improves by up to 40%.

Note that the performance improvement drops when moving from a power cap of 60% to 55%. At 55% power cap, the power constraint is so stringent that most adaptive cores go to the lowest hardware level, eliminating the opportunity to redistribute power from one core to the other. However, an interesting direction of future research is to apply DVFS in select cores to save power to redistribute to other portions of the chip that can make better use of it.

Cores with dynamic voltage and freqency scaling

In the previous section, we discussed the difficulties in evaluating sequential workloads on systems that have different numbers of cores. A more appropriate evaluation looks at current-practice architectures that have the same number of

cores. For example, instead of shutting down cores, a 4-wide CMP can adjust its power by dynamically engaging voltage and frequency scaling. Similarly, in order to maximize single-threaded performance, an 8 core 2-wide CMP can shut down half of its cores and boost voltage and frequency on the remaining four cores.

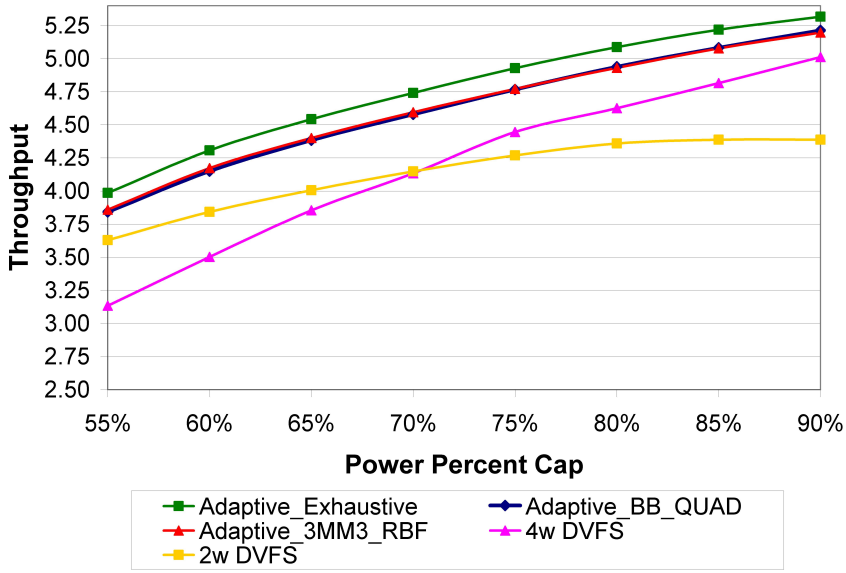


Figure 4.26: 4 core CMP system level improvement over a CMP system with 2-wide cores with DVFS Up and over a CMP system with 4-wide cores with DVFS Down, assuming conservative voltage scaling.

Figure 4.26 shows the fair throughput (geometric mean) of the lane-based adaptive architecture and the two CMPs described above with conservative voltage scaling. For the lane-based architecture (denoted as "Adaptive" in the figure), we show three results. The green data series denoted as "Adaptive_Exhaustive" represents an exhaustive search on the entire combinatorial space to select the best configuration within the corresponding power budget, and cannot be deployed online due to its long runtime. The red series denoted as "Adaptive_3MM3_RBF" represents the fractional factorial design that selects nine sample points, builds a per-core RBF surrogate surface to obtain application response predictions, and

then applies the Genetic Alogrithm limited to 25 iterations to pick and run the best found configuration during the steady interval. The blue line represents the same Genetic Algorithm run on a quadratic surface built on 13 Box-Behnken samples. Lastly, the series denoted as "4w DVFS" and "2w DVFS" show the performance of the two baselines. There are a number of interesting trends captured by this figure. First, both online implementations for the lane-based adaptive architecture exhibit very similar results. Recall from Figure 4.11 that a quadratic surface build on Box-Behnken samples is slightly more accurate than an RBF surface build on 3MM3 samples. However, the 3MM3 design requires four less samples than the Box-Behnken design, effectively increasing the steady interval by 4 ms. Since Adaptive_3MM3_RBF runs a good configuration slightly longer than Adaptive_BB_QUAD, the two contradictory effects cancel out and the performance of the two implementations is almost identical. Second, the online implementation of the adaptive architecture is extremely efficient, losing very little performance over the oracle, offline, Adaptive_Exhaustive approach. Lastly, the lane-based adaptive CMP outperforms both baselines under any power budget, and can be further extended in either direction by the addition of DVFS. The other two baselines cannot meet power budgets either higher (2-wide with DVFS up) or lower (4-wide with DVFS down) than depicted in the graph.

Figure 4.27 and Figure 4.28 compare the adaptive technique to the baselines that employ moderate DVFS. The figures show the percent improvement of the adaptive technique (for the best, worst, and averaged over 20 configurations cases) over the 2-wide with DVFS up baseline, and over the 4-wide with DVFS down baseline, respectively. In the best case scenario, our proposed technique outperforms 4w by 66% and 2-wide by 22%. On average, we improve over the 2-wide CMP between 5.5% and 12.8% depending on the power cap, and between 4% and

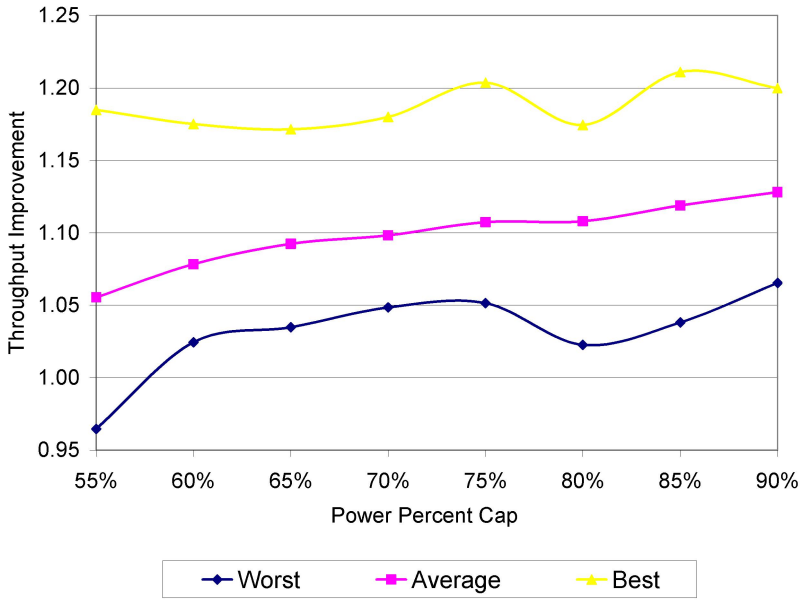


Figure 4.27: Best, average, and worst global throughput improvement over a CMP system with 2-wide cores with DVFS up.

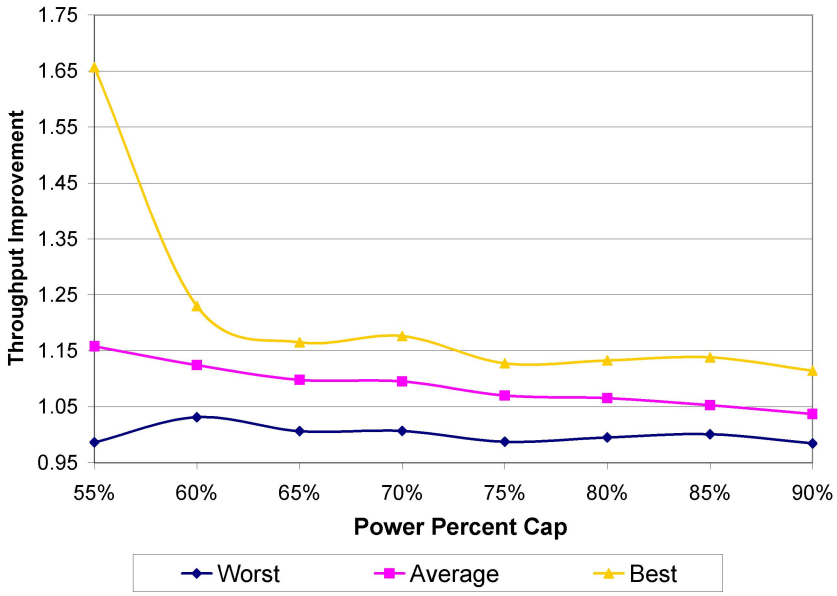


Figure 4.28: Best, average, and worst global throughput improvement over a CMP system with 4 wide cores with DVFS down.

16% over the 4-wide CMP depending on the power constraint. In the worst case scenario, Adaptive is able to at least match the performance of the 2-wide baseline

with the exception of the 55% power cap, where performance is degraded by less than 5%. Our technique also matches the performance of the 4-wide CMP within ±3% for the worst performing workload.

The previous results show averages for the 20 randomly created workloads since we believe they are an accurate representation of real system behavior where the Operating System schedules applications from a ready queue. However, the randomness of the workloads makes it difficult to evaluate the type of tasks the adaptive architecture is suited or unsuited for, and to extract key insights about the potential benefits of our technique. We classify the SPEC CPU2000 benchmarks into three categories (CPU, cache, and memory bound) and create six workloads that explore the possible combinations of these categories. The workloads are shown in Table 4.3, and correspond to: a workload comprised solely of CPU bound applications, a workload comprised solely of cache bound applications, one comprised only of memory bound applications, and three workloads that explore a combination of two benchmark classes: CPU and memory bound, CPU and cache bound, and cache and memory bound.

CPU	apsi	gcc	wupwise	gcc
MEM	swim	art	art	mcf
CACHE	vpr	twolf	twolf	crafty
CPU+MEM	apsi	art	wupwise	art
CPU+CACHE	gcc	crafty	wupwise	crafty
CACHE+MEM	twolf	art	crafty	swim

Table 4.3: 4-benchmark workloads created from SPEC CPU 2000 benchmarks classified as CPU, cache, and memory bound.

Figures 4.29 and 4.30 show the system throughput improvement of the adaptive scheme over a CMP system with 4-wide cores with DVFS down to match the power budget and over a CMP system with 2-wide cores with DVFS up, respec-

tively. Results are shown for power caps of 90%, 70%, and 55%. As expected, adaptive greatly outperforms the 4-wide CMP with DVFS down if all the applications scheduled on the CMP are CPU-bound, and performs worse if all applications scheduled are memory bound. For CPU bound applications, the improvements peak at 56% around a power cap of 70%, and decrease to 43% at a power cap of 55%, rather than increasing with more stringent power constraints as is the case for the other workloads. This is due to the fact that the performance of CPU bound applications dramatically drops as the number of active lanes drops to 2, reducing the difference between a low frequency design (DVFS) and a weak core design (2 lanes turned off). In general, if memory bound applications are present, employing DVFS down to manage power is more effective than the lane-based architecture, because a significantly higher portion of power can be saved with little performance loss.

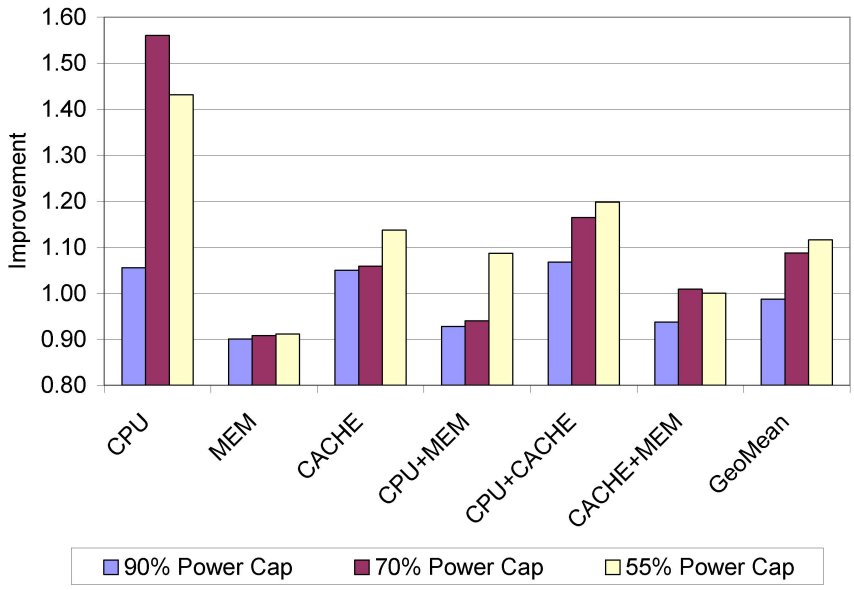


Figure 4.29: 4 core CMP system level improvement over a CMP system with 4 wide cores with DVFS down for select workloads.

When comparing the adaptive technique to a CMP consisting of 2-wide cores

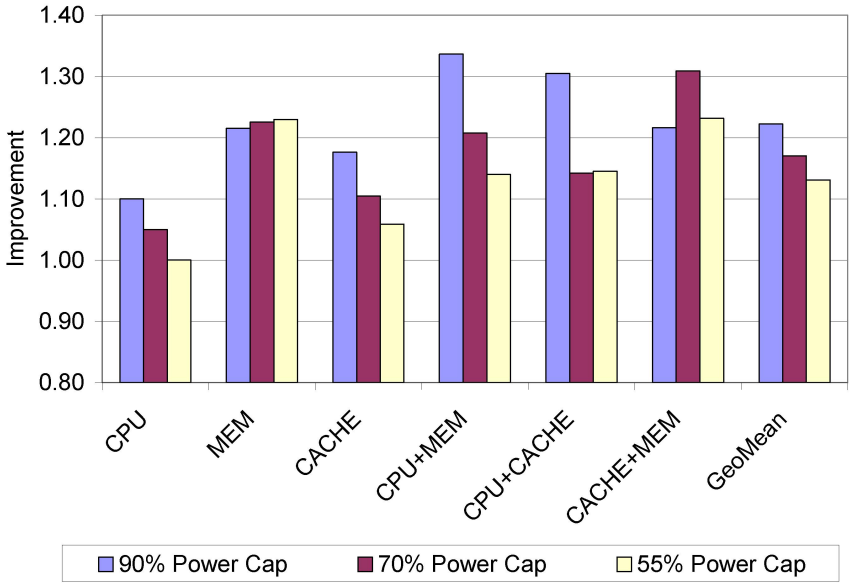


Figure 4.30: 4 core CMP system level improvement over a CMP system with 2-wide cores with DVFS up for select workloads.

employing DVFS up to match the same power budget (Figure 4.30), the trends are the opposite: better gains are obtained when running memory or cache bound applications, because increasing frequency does not benefit these applications. Excluding the corner cases (all CPU bound or all memory bound applications in one workload), the adaptive lane-based architecture is most effective when the running applications exhibit variety in their power and throughput responses. In such situations, a subset of cores effectively borrow power from the others, creating a dynamically heterogeneous architecture and realizing much bigger gains for the same chip-wide power. Conversely, when the workload exhibits no software variety, our adaptive technique matches, but is not able to greatly outperform, homogeneous decisions and architectures employing DVFS up or down.

Lastly, we present the average improvements over 2-wide and 4-wide CMPs with DVFS under all three scaling assumptions: aggressive, where voltage can be scaled by ±40%; moderate, where voltage can be scaled by ±20%; and conservative,

where voltage can be scaled by ±15% (Figure 4.31). The 4w (aggressive) and 2w (aggressive) techniques can apply voltage and frequency scaling for all power constraints up to 55% and 90%, respectively, which is why they perform the best out of the three DVFS scaling assumptions. For moderate DVFS, neither the 4w or the 2w can engage DVFS to match the power budget at the endpoints of the ranges we consider. 4w has to resort to shutting down cores in addition to DVFS to meet the power constraint, while 2w must stop scaling voltage up when it is still below the power constraint, missing the opportunity to increase performance. Conservative DVFS pushes these points to the left for 4w and to the right for 2w.

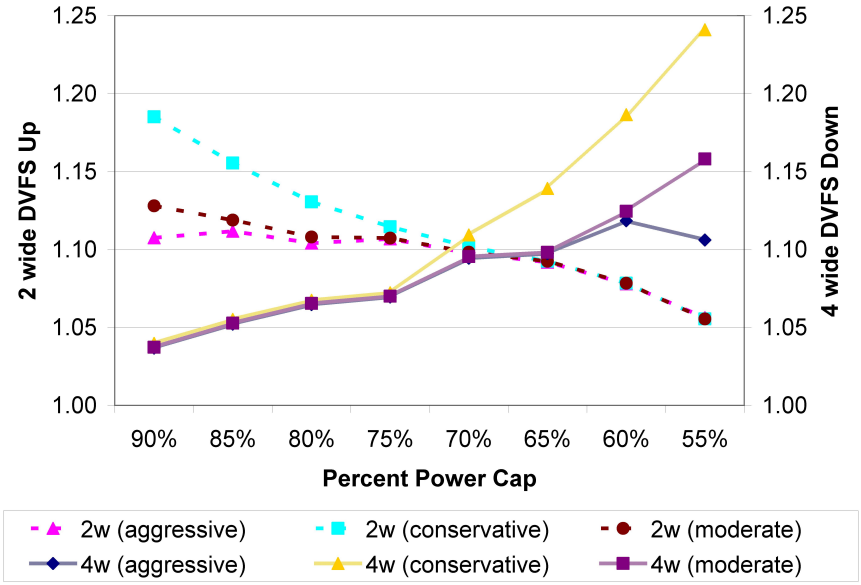


Figure 4.31: 4 core CMP system level improvement over a CMP system with 2-wide cores with DVFS Up (left y-axis) and over a CMP system with 4-wide cores with DVFS Down (right y-axis), assuming aggressive, moderate, and conservative voltage scaling.

4.3.5.3 Scaling to Many Cores

The analysis in the previous sections has focused on 4-core CMPs. Figure 4.32 shows that the online optimization methodology we presented scales very well even for CMPs with 32 cores. Since the runtime for exhaustive algorithms becomes prohibitive as the number of cores is increased, we use the proxy described in Section 4.3.4 to obtain an upper bound on the CMP throughput we can expect.

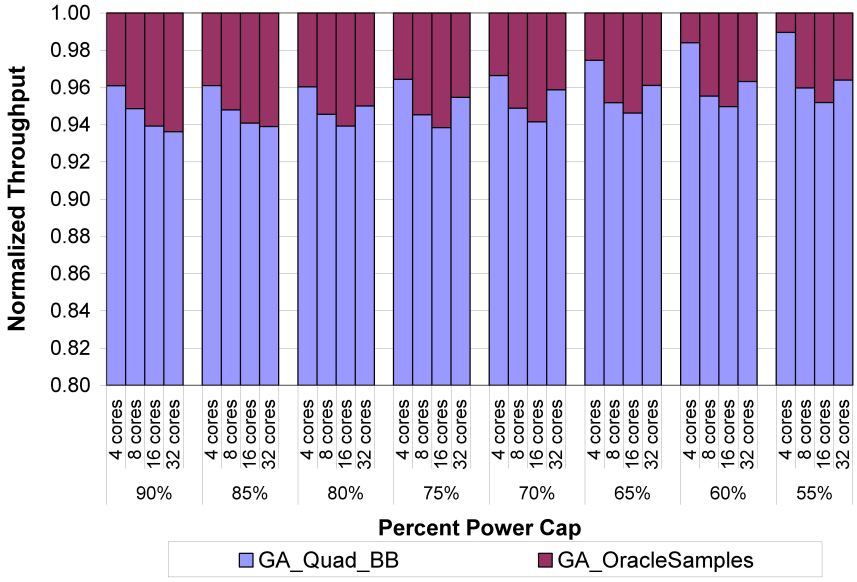


Figure 4.32: Performance of the lane-based adaptive technique for CMPs with increasing number of cores. Throughput is normalized with respect to a Genetic Algorithm run on oracle samples for 200 generations.

In general, across the range of power constraints considered, the online Genetic Algorithm loses some accuracy as the number of cores is increased, but still finds solutions that capture more than 94% of the potential performance. An interesting exception is the 32 core case, for which solutions are found that are more accurate than for 8 or 16 cores. There are two possible explanations for this behavior. First, the 20 8-, 16-, and 32-application workloads are randomly generated, which means

that one should not attempt a direct comparison between CMPs with differing number of cores. It is possible that for 32 cores, the workloads happen to have more "reasonably good" solutions. Second, it is possible that with many cores, more feasible solutions actually exist in the 32 variable space under the very strict power constraint, since every workload most likely has a number of low power benchmarks. As such, it is easier for the Genetic Algorithm to pick feasible parents for mutation and crossover which in turn are likely to produce good offspring.

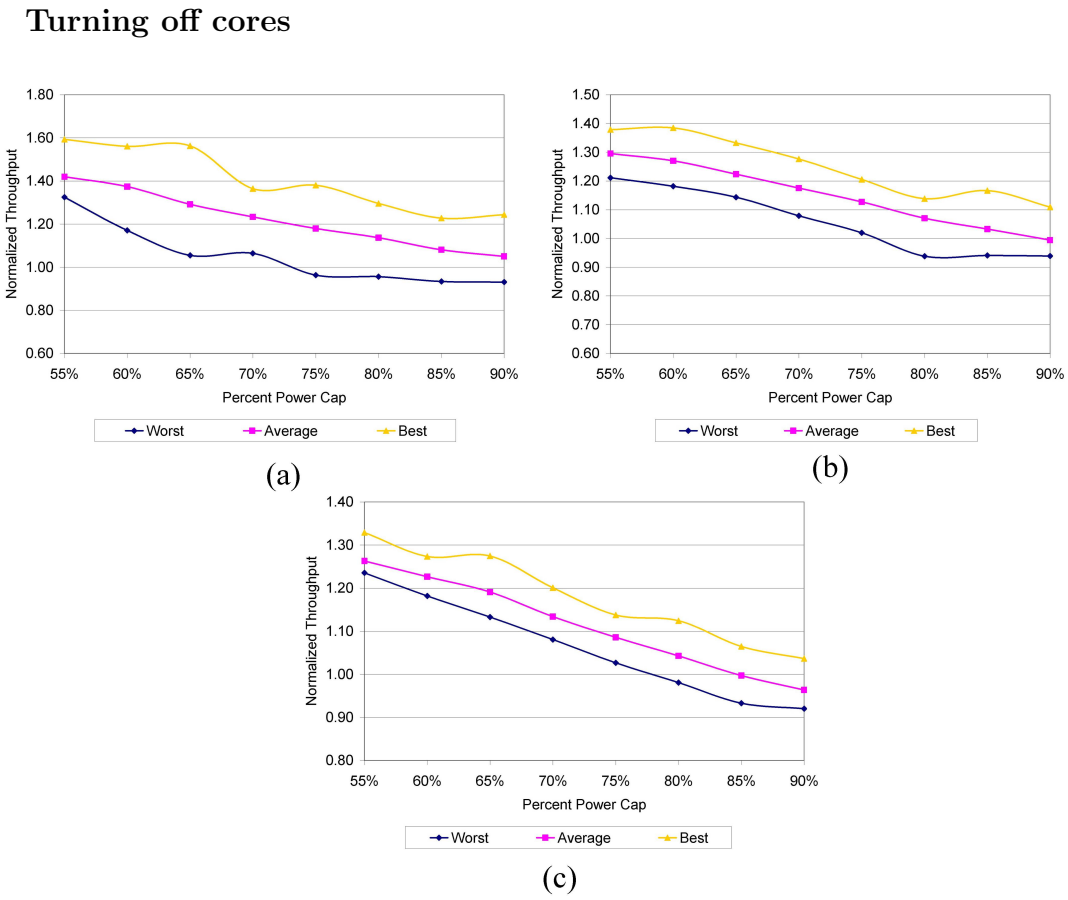


Figure 4.33: Best, average, and worst global throughput improvement of an N-core lane-based architecture over a CMP system with N 4-wide cores that shuts down cores to meet the power budget. (a) 8 core CMP; (b) 16 core CMP; (c) 32 core CMP

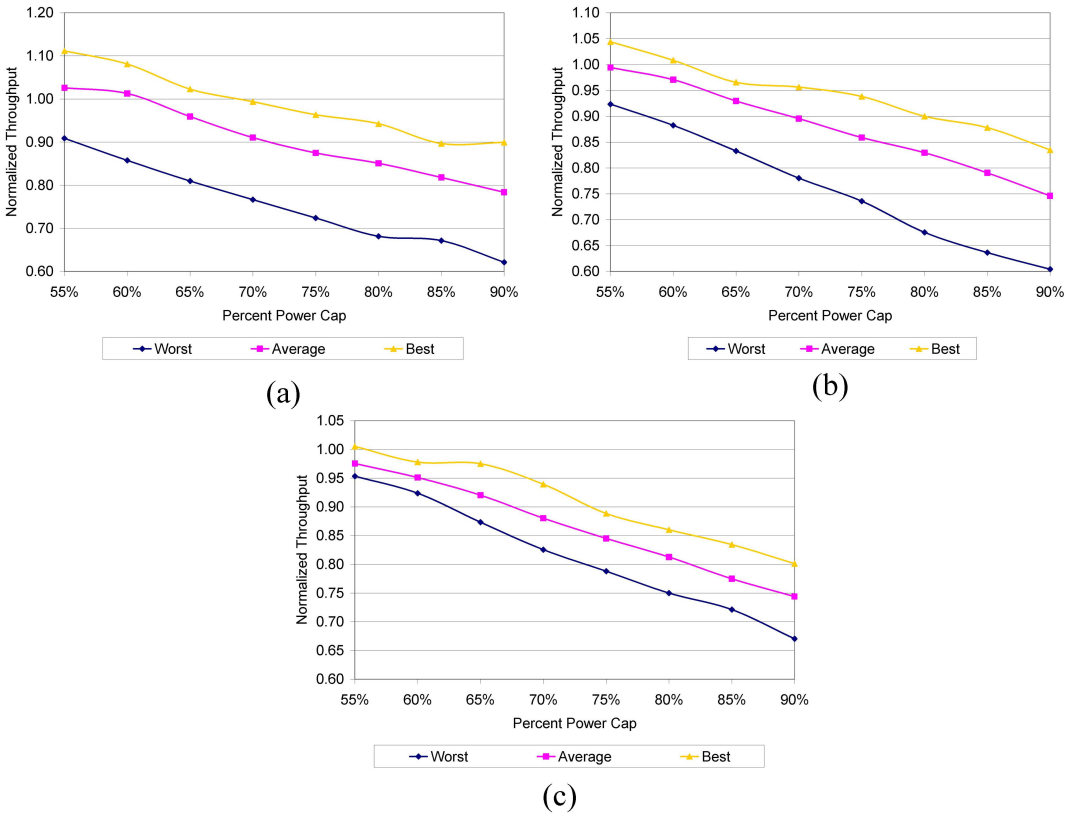


Figure 4.34: Best, average, and worst global throughput improvement of an N-core lane-based architecture over a CMP system with 2N 2-wide cores that shuts down cores to meet the power budget. (a) 8 core CMP; (b) 16 core CMP; (c) 32 core CMP

Figures 4.33 and 4.34 show how the adaptive lane-based technique compares against static designs that disable cores to meet the power budget as the baseline number of cores is scaled up to 8, 16, and 32. First we compare an N core adaptive lane-based CMP against an N core 4-wide CMP in Figure 4.33. As the number of cores is scaled up, the power budget expressed as a percentage of the total power consumed when all transistors are on is expected to be in the 50% range based on [26]. The adaptive technique presented in this work outperforms the static 4-wide CMP at low power budgets (55% power cap) by 40%, 30%, and 27% as the number of integrated cores increases to 8, 16, and 32, respectively. It is

important to note two effects: first, as the number of integrated cores is increased to 32, the difference between the best, average, and worst performing workload is reduced; second, the system level improvement in throughput decreases with increasing number of cores. Both of these effects are due to the fact that we have a limited number of diverse applications. Therefore, when the number of cores outnumbers the available applications, the 32-core workloads that we can produce are alike, individual application behavior is assimilated and lost in the average SPEC CPU benchmark behavior, and the workload behavior becomes similar to a 4 core workload that has four average applications. Even at unrealistically high power budgets (90% power cap), the adaptive technique is able to match on average the performance of a static design that disables cores, with the exception of the 32 core CMP, where adaptive performs less than 5% worse than the static design. This trend shows that when power is not constrained, simple techniques like DVFS engaged on static designs provide good performance. However, there is a clear need for more sophisticated architectures and control algorithms as the power is increasingly constrained according to scaling predictions.

Figure 4.34 shows that if pure throughput is the only metric of interest to microprocessor designers, integrating many weaker cores is hard to beat, regardless of whether an adaptive technique based on 4-wide cores or a static 4-wide CMP are employed. However, our adaptive lane-based technique can be integrated within weaker cores as well to reduce them to scalar cores if necessary, or applied at a finer grain to individual structures rather than pipeline regions.

Static cores with DVFS

Figures 4.35, 4.36, and 4.37 show the improvements in performance of the adaptive lane-based technique over 4-wide static cores that scale voltage and frequency

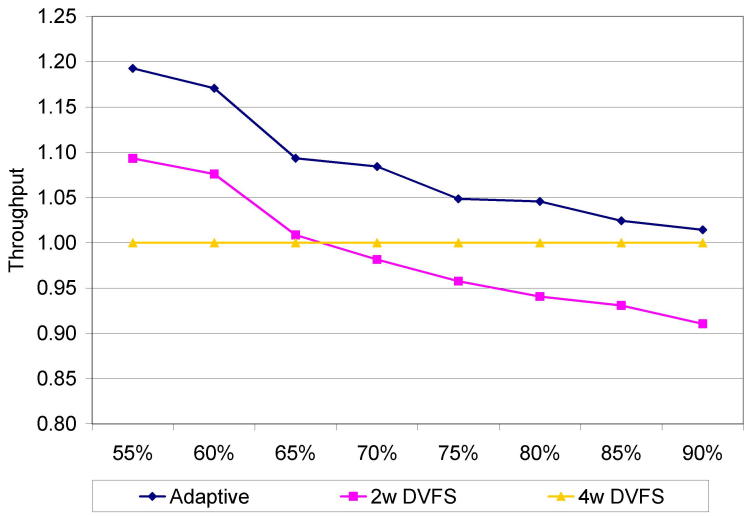


Figure 4.35: Adaptive lane-based 8-core CMP improvement over a CMP system with 8 active 2-wide cores with DVFS Up and over a CMP system with 8 4-wide cores with DVFS Down, assuming conservative voltage scaling. All results are normalized with respect to the static 4-wide CMP with DVFS Down.

down (yellow data series), and over 2-wide static cores that scale voltage and frequency up (pink data series) to meet the power budget as the number of integrated cores increases to 8, 16, and 32. Our technique consistently outperforms the 2-wide cores with DVFS up by at least 7% on average across 20 random workloads as the number of cores in the CMP is scaled up. Similar to the results for core disabling, if power is not a concern (90% power cap), then 4-wide cores that engage DVFS outperform our adaptive technique. The adaptive technique overheads are not warranted if the system is not constrained, and simple methods such as DVFS are more adequate to optimize system performance. However, as the power budget is reduced as expected due to device scaling, the more refined lane-based technique is a superior choice.

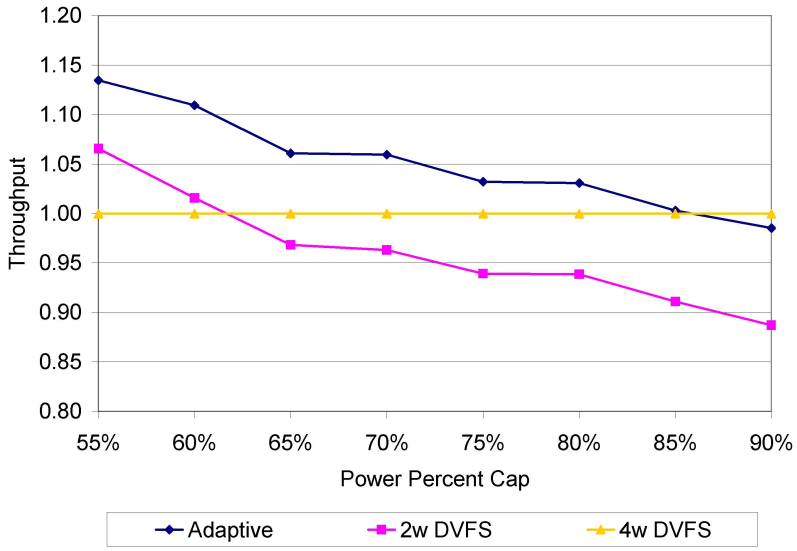


Figure 4.36: Adaptive lane-based 16-core CMP improvement over a CMP system with 16 2-wide cores with DVFS Up and over a CMP system with 16 4-wide cores with DVFS Down, assuming conservative voltage scaling. All results are normalized with respect to the static 4-wide CMP with DVFS Down.

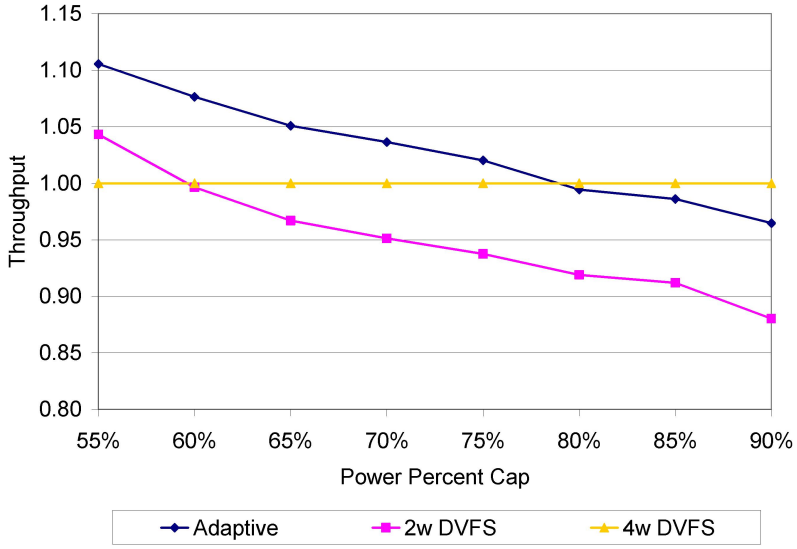


Figure 4.37: Adaptive lane-based 32-core CMP improvement over a 32-core CMP system with 2 wide cores with DVFS Up and over a 32-core CMP system with 4-wide cores with DVFS Down, assuming conservative voltage scaling. All results are normalized with respect to the static 4-wide CMP with DVFS Down.

4.3.5.4 Parallel Workloads

Global decisions for a system running parallel workloads can be simplified by making the observation that decisions for threads belonging to the same application should be made identically. Since efficient parallelization assigns a similar amount of work to each thread through load balancing and the threads are periodically synchronized for global communication, it is most efficient that all threads run on the same hardware configuration in order to guarantee similar execution times. As such, global decisions in a system that runs multiple applications with multiple threads needs only consider a system that runs one thread of each application. For example, a 32 core CMP running 4 applications with 8 threads makes decisions in a similar fashion to a 4 core CMP running 4 applications with one thread each. Once the best configuration is chosen for one of the threads, the remaining threads are run on cores reconfigured with the same hardware configuration. This implies that the decision space for parallel workloads is reduced to the decision space of the number of independent applications, making the search more efficient than for multiprogrammed workloads running on the same number of cores.

Our current simulation infrastructure does not support parallel workloads, but we approximate their behavior on our system by simultaneously running multiple copies of single threaded workloads. Figure 4.38 shows results for a 32 core CMP running 20 random workloads. Each of the 20 workloads are obtained by randomly choosing 4 SPEC CPU2000 benchmarks and running 8 copies of each of them. The figure shows the fair throughput of the adaptive lane-based architecture and of a static 4-wide 32 core CMP that scales voltage and frequency down to match a 55% power percent cap. All results are normalized with respect to the 32 core CMP with DVFS down. Our technique improves system throughput by up to 30% and

at least 2%, with an average improvement of 13% for the workloads we considered.

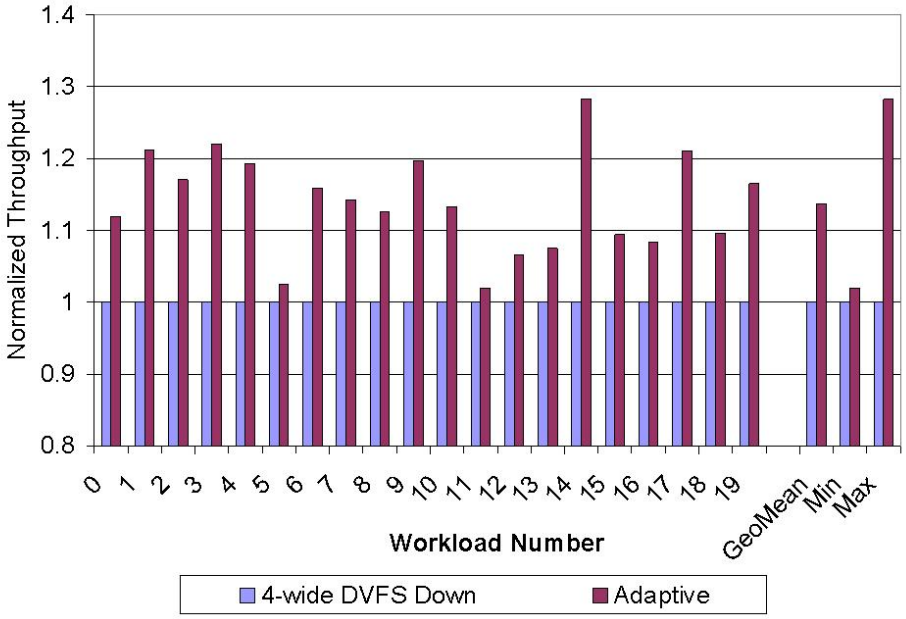


Figure 4.38: Adaptive lane-based 32-core CMP improvement at 55% power cap over a CMP system with 32 4-wide cores with DVFS down, assuming conservative voltage scaling. All results are normalized with respect to the static 4-wide CMP with DVFS down.

4.3.5.5 Pareto Optimality

In this section, we show that the granularity and complexity at which the adaptive lane-based architecture operates is justified over simpler architectures and optimization techniques. In particular, we show how two simple architectures are inferior to ours because they cannot efficiently adapt to a range of power constraints. The first architecture is an asymmetric chip multiprocessor that has N groups of cores, each with one 2-wide core, one 3-wide core, and one 4-wide core. This architecture enables only one of the three different cores from each group. The second architecture is based on our adaptive architecture but attempts to

reduce the complexity of sampling and optimization by varying the number of active lanes homogeneously across all pipeline regions. Each core in this simplified lane-based architecture can be configured in one of three ways: four, three, or two active lanes in all pipeline regions. This architecture effectively captures all the benefits of an asymmetric CMP with a reduced area overhead. As such, we will focus on the latter architecture. For clarity, we present results for two-core CMPs, but the trends are similar for an increased number of cores.

Figure 4.39 shows the decision space for a 2 core homogeneous lane-based CMP (black diamonds labeled with the width of each core) and for a 2 core adaptive lane-based CMP that can have a different number of active lanes in each pipeline region (blue circles). The former CMP has 3^2 or nine chip-wide hardware combinations, while the latter has 27^2 or 729 combinations. The green diamonds correspond to the former combinations that are Pareto optimal and the red circles correspond to the latter combinations that are likewise Pareto optimal. The horizontal lines represent hypothetical power constraints. For memory bound applications such as mcf and swim, simple architectures are effective at adapting to the power budgets as shown in the top plot of Figure 4.39. In the worst case, the optimal heterogeneous lane-based configuration (red circle) that fits under the power budget performs only 4% better than the optimal homogeneous lane-based configuration that reduces the width of the core running mcf to 2-wide and the width of the core running swim to 3-wide (black diamond). However, for other applications such as gcc and apsi, the homogeneous architecture misses a large performance opportunity as shown in Figure 4.39 (bottom) across a variety of power constraints. For example, at a power constraint of 18 Watts (red horizontal line), the best configuration that the homogeneous architecture can employ is reducing both cores to 3-wide (black diamond labeled "(3w3w)"). The best configuration that the heterogeneous lane-

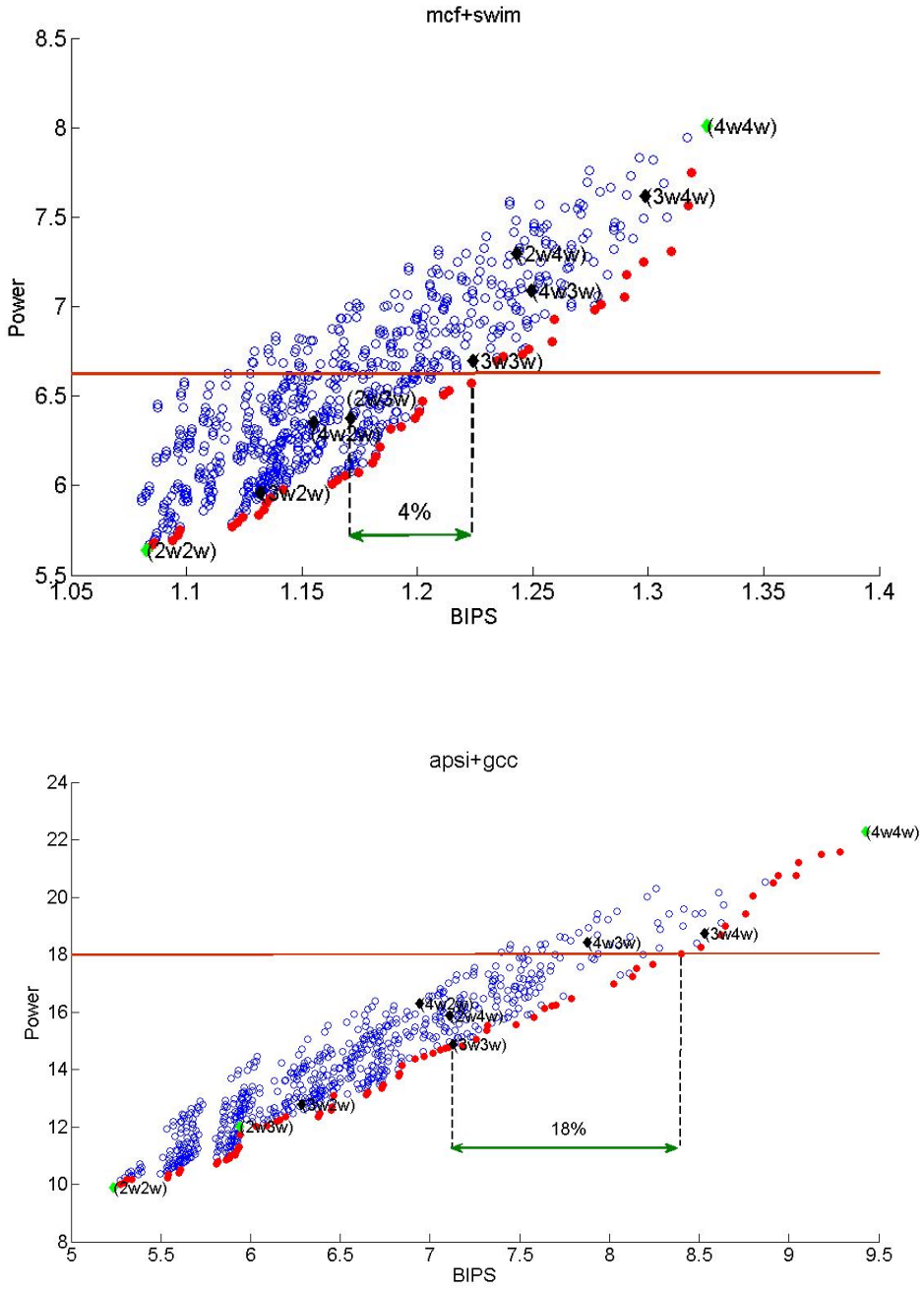


Figure 4.39: Power-performance Pareto fronts for 2-core CMPs running two multiprogrammed workloads: mcf and swim (top), and apsi and gcc (bottom).

based architecture can employ under the 18 Watt power constraint is (433,443), that is four active FE lanes, three active BE lanes, and three active LSQ lanes in

the core running apsi, and four active FE lanes, four active BE lanes, and three active LSQ lanes in the core running gcc. This configuration outperforms the homogeneous one by 18% at this power constraint.

CHAPTER 5

OPTIMIZATION TECHNIQUES FOR PERFORMANCE RECOVERY IN FAILURE PRONE CMPS

5.1 Introduction

This chapter explores how the lane-based architecture, together with performance boosting techniques, can mitigate the performance losses associated with hard faults that occur in chip multiprocessor pipelines. The possibility of wear-out failures and manufacturing defects is forcing multicore architects to include the capability of deconfiguring various features that may become faulty, to permit the system to operate in a degraded state in the event of a hard error. The most obvious redundancy to exploit in a multicore microprocessor is at the core level, where an entire core is disabled when it encounters the first fault. With many failures possible at product shipment [11] and over the lifetime of a product, this approach is wasteful and quickly degrades the performance of the entire system. On the other hand, finer grain levels of redundancy that permit each core to operate in a degraded state provide longer processor lifetime. Examples of processor structures whose inherent redundancy have been exploited by prior researchers for fault tolerance include banked RAM structures such as caches and register files [48, 64], multiple queue entries [12], and duplicate functional units [64]. Even though this approach provides very good resiliency to multiple faults, at very fine grained levels of redundancy, the overheads associated with fault detection, isolation, and reconfiguration or spare replacement can be prohibitively large.

Arguably, the most difficult challenge of defects and wear-out faults is providing reconfiguration mechanisms that have a reasonably low built-in cost and that,

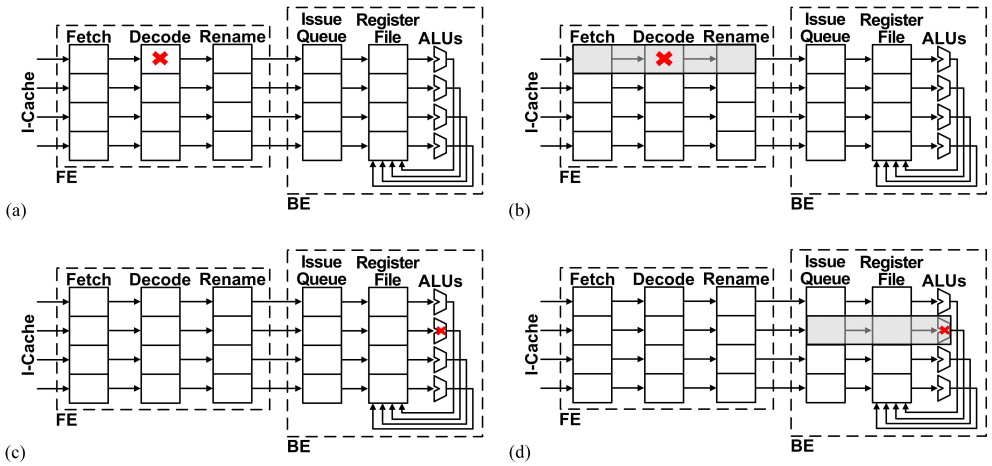


Figure 5.1: Lane-based fault tolerant pipeline microarchitecture.

when activated, have a high probability of making the loss of chip functionality imperceptible to the user from a performance perspective. One recently proposed approach is to salvage partially functional cores by stitching working parts together to form a fully functional core. For example, StageWeb [29] implements a sea of pipeline stages microarchitecture in which individual stages can be combined via a set of interconnection networks to form fully-functional pipelines. Should a stage become defective due to a wear-out failure, the interconnect is reconfigured to make best use of the remaining fully-functional stages. The advantage of this approach of stitching together fine-grain vertical slices of a conventional pipeline is that it can make good use of the fully functional stages within the chip, so long as defects are distributed in a way that permits nearby stages to be suitably combined. A potential disadvantage is the built-in area, power, and performance costs of the network that is required to combine stages.

An alternative to this "vertical slicing" approach is to slice the pipeline *horizontally*, using the modular CMP design presented in Chapter 3. Figure 5.1(a) shows a superscalar pipeline with a fault that causes one of the instruction decoders to

be unusable. The fault reduces the decode width by one instruction, which in turn decreases the front-end (FE) width by one instruction due to the tightly-coupled nature of the front-end pipeline stages. Thus, if the FE is architected as individual instruction lanes – horizontal slices through fetch, decode, rename, and dispatch that can be independently deconfigured – the FE can be reconfigured as a functional, but narrower, pipeline (Figure 5.1(b)). Moreover, since the issue queue decouples the front and back ends of the pipeline, the back-end (BE) can remain operational at its full width.

Similarly, a fault in one of the execution units (Figure 5.1(c)) reduces the execute width by one instruction, which in turn decreases the BE execution width by one instruction. With the BE pipeline architected as lanes similar to the FE, the affected lane can be deconfigured to permit the BE to operate as a narrower pipeline while the FE remains fully functional (Figure 5.1(d)).

A significant advantage of this approach over a vertically sliced microarchitecture is that it obviates the need for a complex interconnection network between adjacent pipeline stages. A laned microarchitecture can leverage common built-in pipeline mechanisms – such as the rotation of instructions from the cache into the proper pipeline position [76], and partial instruction issue to a subset of the functional units – to provide low-cost deconfiguration in the presence of a pipeline fault.

Despite these advantages, a laned microarchitecture may introduce *pipeline imbalance*. A modern pipeline is highly tuned to match its instruction fetch, load/store, and execution bandwidths according to the overall characteristics of the workload. When a fault causes a lane of the FE or BE to be deconfigured, the unaffected part of the pipeline may now become *overprovisioned*. That is, the

full width of the unaffected part of the pipeline may become unnecessary given the reduced bandwidth in the faulty pipeline section. This mismatch of the front and back end bandwidths leads to inefficient use of chip power.

As we show in Section 5.3.1, the deconfiguration of a lane due to a fault can lead to significant pipeline imbalance in some applications. We propose to *rebalance* the pipeline in these situations through *symbiotic deconfiguration* of a lane of fully-operational, but now overprovisioned, pipeline regions. This approach takes advantage of the lane-based deconfiguration capability built into the fully functional pipeline for fault tolerance. For some applications, additional deconfiguration of a lane of a fully-functional pipeline region results in significant performance loss. For those applications, only the lane with the fault is deconfigured while the other regions remain fully operational. Thus, symbiotic deconfiguration is a dynamic technique that deconfigures additional lanes in fault-free pipeline regions only when those regions are overprovisioned for the currently running application.

Symbiotic deconfiguration can restore the performance lost due to pipeline faults by enabling chip-wide *power redistribution*. A key observation is that pipeline rebalancing via symbiotic deconfiguration results in little additional performance loss while recouping a comparatively larger amount of the chip-wide power margin. The power saved by improving pipeline efficiency in this manner can be more profitably used to boost chip-wide performance by *transferring* that power to other functionality. An obvious choice is to boost the frequency of the affected pipeline via DVFS. However, we show that the use of several performance boosting techniques, and distributing all such *harnessed power* from a chip-wide pool in a more optimal fashion among multiple cores, yields significantly higher performance.

The focus of this chapter is *PowerTransfer*, a novel fault-tolerant modular

multicore architecture that harnesses power through dynamic pipeline rebalancing and uses that power to recover the performance lost due to pipeline faults. PowerTransfer identifies power harnessing opportunities in which a now-overprovisioned pipeline region can be symbiotically deconfigured with little performance loss, and simultaneous *boosting* opportunities where applications can improve performance given the harnessed power to enable additional microarchitectural features.[1]. We develop heuristic optimization methods that permit periodic assessment of symbiotic deconfiguration and performance boosting opportunities and that achieve nearly the same performance as exhaustive techniques that cannot scale to large multicore systems. Our results for up to 32 core CMPs demonstrate that PowerTransfer can fully recoup the lost performance due to pipeline faults, thereby making these faults imperceptible to the user.

5.2 Performance Boosting Techniques

Once a margin of additional available power has been accumulated by exploiting both permanent unit deconfiguration (due to a fault) and phase-level deconfiguration (for pipeline rebalancing), this power is distributed among the chip components in order to boost performance. This is accomplished by temporarily enabling previously dormant hardware features within the limits of the global power budget and local temperature thresholds.

By definition, a performance boosting technique does not improve performance for most applications; otherwise, the technique would be built-in to the design by default. Rather, these techniques improve what might be deemed *performance cor-*

[1]Such a feature, when enabled, must not cause a violation of maximum thermal limits or excessive di/dt noise.

ner cases, snippets of particular applications. In fact, one might consider microarchitecture techniques that were discarded from consideration since they improved performance for a small subset of applications as candidate performance boosting techniques, so long as their overhead is reasonably minor. While the speedup may be significant in these situations, in most cases, the power cost exceeds the performance gain such that the technique is not enabled by default.

Thus, there are a number of important criteria in considering the adoption of a particular performance boosting technique. Since the dormant boosting techniques must be readily available for temporary use, fast power-up is necessary to ensure timely exploitation of the accumulated power. Moreover, engaging the performance boosting technique must not cause sudden power surges or trigger localized hotspots due to increased resource utilization, which would engage DVFS and counteract performance gains. Ideally, the technique should also be simple in design and have low overhead to justify its existence on-chip in the powered-down state. In the powered-up state, the technique should provide a good performance/power ratio for particular application phases. In addition, the boosting techniques should collectively cover a range of performance corner cases such that there is ideally always some worthwhile boosting technique to engage given some harnessed power no matter what mix of applications are running.

In PowerTransfer, several boosting techniques are implemented that collectively cover a range of performance corner cases such that there is ideally always some worthwhile boosting technique to engage given some harnessed power, no matter what mix of applications are running.

While there are many potential approaches, we implement three techniques that cover the spectrum of CPU, cache hierarchy, and memory performance-bound

applications: DVFS, Speculative Cache Access, and Checkpointed Early Load Retirement (Clear) [47]. This is not intended to be a complete list as there are likely dozens of potential techniques.

5.2.1 Dynamic Voltage and Frequency Scaling

As a result of a fault and subsequent symbiotic deconfiguration, the processor has narrower processing bandwidth and is not able to exploit as much ILP in the currently running application, leading to reduced IPC. To compensate, the saved power can be used to increase the voltage and frequency of that core, assuming this core was operating below its maximum settings due to power budget constraints. Thus, the lost IPC can be made up with increased frequency, hopefully negating much of the performance penalty of the hardware fault. Future microprocessors may include multiple frequency and voltage domains, and recent research has shown the merits of separate domains for each core [39, 45]. Similar to Intel's Turbo Boost [75], we increase the operating voltage and frequency within the constraints of the overall power budget.

We note that benchmarks can be divided into three rough categories: high, mid, and low IPC benchmarks. Since we only boost the frequency of the core but not of main memory, low IPC benchmarks that are already memory bound will exhibit insignificant improvement in performance even when operated at a significantly higher frequency. High IPC benchmarks are generally computation bound and could benefit from boosting given enough power harnessed from deconfiguration.

The most straightforward approach is to use the saved power locally within the core with the faulty unit to compensate for the loss in IPC performance. Instead,

PowerTransfer adds any locally saved power to the chip-wide pool of accumulated power for potential use in boosting other cores' performance. While this requires more complex chip-level power management algorithms, a performance gain closer to the global optimal can be found. For instance, it would be more worthwhile to boost the frequency and voltage of another core running a computationally intensive thread when the local core is running a memory-bound thread.

We apply DVFS by noting that memory behavior is usually well correlated with IPC. As such, benchmarks with high IPC are usually not memory bound and are considered good candidates for DVFS boosting. Benchmarks are thus sorted by IPC and DVFS is speculatively engaged in a greedy fashion starting with the highest IPC benchmark until the leftover power is exhausted. If the DVFS boost will exceed the power budget, voltage and frequency are scaled down accordingly.

We use four voltage and frequency levels above the baseline frequency and voltage, in 2.5% Vdd increments. We limit the Vdd increase to 10% above nominal to avoid overly engaging the Global Power Manager when the power budget is exceeded.

While DVFS provides good power savings relative to the performance loss when scaling frequency down, the opposite is true when scaling frequency up (boosting): a cubic increase in dynamic power is incurred for a linear increase in frequency. The other two techniques that we identify have a more favorable power-performance ratio, and are also beneficial for applications that are not CPU-bound.

5.2.2 Speculative Cache Access

For applications that are L1 miss limited, improvements in the access time of lower levels of the cache hierarchy would be more beneficial than DVFS. Speculative cache access is an effective means to boost performance at reasonable cost.

L2 caches are typically accessed in sequence after L1 lookup. To do so otherwise would greatly increase power consumption for relatively little overall performance gain. A performance boosting technique that requires little added hardware complexity is to speculatively send L1 requests to the L2 cache simultaneously in order to reduce the delay penalty in the case of an L1 miss. We expect good improvements for applications that miss in the L1 but hit in the L2.

The main drawback of this technique is the substantial additional power requirement, which amounts on average to increasing a core's power usage by 60%. There are two sources of additional power. First, speeding up the memory hierarchy increases the rate of computation performed by the core. As such, the Issue Queue, Functional Units, and Register File are exercised considerably closer to their design limits, causing them to dissipate proportionally more power. However, the biggest source of additional power (on average four fifths of the total) comes from unnecessarily accessing the L2 cache even in the presence of L1 cache hits.

In order to reduce this latter power consumption, we add a Load Miss Predictor, a two-bit saturating counter, updated with L1 hit/miss information in a similar fashion to the one used in Alpha 21264 [43] for speculative instruction issue. That is, on a load hit, we increment the counter by one, and on a load miss, we decrement the counter by two in order to bias it more heavily towards a performance benefit (a

load that is predicted as a hit in L1 but actually misses has the effect of serializing L1 and L2 accesses). As we show in Section 5.3.1, this low-overhead predictor has little performance penalty yet reduces L2 access wasted power by 90% on average.

A similar cost-effective performance boosting technique is to access the L2 tags and data in parallel. Lower level caches such as those in the Itanium II [74] and Alpha 21164 [25] access the tag and data arrays in sequence due to power concerns. Given additional harnessed power due to symbiotic deconfiguration, and an application that can achieve significant performance gains from parallel tag and data access, the L2 cache can be easily switched into parallel mode.

5.2.3 Checkpointed Early Load Retirement

Many speculative techniques have been proposed to boost the performance of memory-bound applications, but these may come at a prohibitive power cost. When there is potential benefit (i.e., many long latency loads) and sufficient power has been harnessed, we engage Clear mode [47]. In this mode, the registers are checkpointed, stores are buffered in the store queue, loads are speculatively early retired, and the predicted values are supplied to their destination registers. Through these mechanisms, dependency chains following a long latency load complete early, and processor resources are freed for use by non-dependent instructions. We implement a Prediction Queue of 48 entries, up to four checkpoints, and a checkpoint allocation threshold of seven loads.

5.2.4 Power Transfer Runtime Manager

The PowerTransfer Runtime Manager (PTRM) coordinates the chip-wide effort to re-allocate power among the cores to maximize performance. The PTRM collects online profile information on the performance and power dissipation of the applications on each core to assess the costs of possible deconfigurations and the benefits of power boosting alternatives. It then determines what lanes should be symbiotically deconfigured to save power, and allocates the harnessed power to boosting mechanisms on the different cores. In order to adapt to dynamic program behavior, the PTRM operates at a time granularity of tens to hundreds of milliseconds.

Operation at this time granularity also allows the PTRM to coordinate with the Global Power Manager (GPM), which controls per-core frequency and voltage to maintain the chip-wide power budget. The GPM acts as a fail-safe mechanism in instances where the PTRM underestimates the additional power cost of enabling a performance boosting techique. Such overshoots, which we account for in our results, occur infrequently.

However, operation at this time granularity makes an exhaustive search of symbiotic deconfigurations together with the performance boosting techniques unfeasible. Each core can be configured in 160 ways: 4 symbiotic deconfigurations, 5 DVFS configurations (one of four DVFS levels or no frequency boosting), 4 ways to employ Speculative Cache Access, and 2 ways to employ Clear (on/off). For a four core CMP, this results in a total of 160^4 chip-wide combinations, clearly showing that runtime exhaustive exploration of the space is impractical.

Therefore, the PTRM uses systematic sampling and power-performance metrics

to make symbiotic reconfiguration decisions. After this step, for a four core CMP, the PTRM can exhaustively explore the different combinations of performance boosting techniques. For larger CMPs, this approach is not scalable, and therefore the PTRM uses heuristic optimization algorithms to select the combination of boosting techniques.

There are a number of alternatives for implementing the PTRM. In order to obtain data about faulty components, deconfigure units, and obtain performance and power statistics, the PTRM requires access to low-level hardware information. Consequently, one option is to implement the PTRM as an embedded microcontroller similar to the Foxton Technology Controller included in Intel's Montecito [52]. The advantages of this approach are direct access to hardware and the fast, real-time responsiveness of an on-chip controller. However, implementing the whole manager in hardware would incur the highest die area and hardware complexity costs. Furthermore, it would be the least amenable to upgrades, which may be quite useful as the processor ages and wears out further, changing the power re-allocation tradeoffs. An alternative to a full hardware solution is to dedicate hardware to detect errors, deconfigure components, and gather statistics, and implement the re-allocation logic in software. The re-allocation algorithms could be incorporated into a low-level hypervisor (supervisor) level thread, or at the higher level as part of the operating system. The main factors dictating the best option would be the ease of implementation, the desire to expose applications to the decision process, and the desired granularity at which the PTRM should operate.

5.2.4.1 Symbiotic Deconfiguration

The PTRM first makes symbiotic deconfiguration decisions on each core by deconfiguring one lane for a sampling period in each of the fully-functional pipeline regions in turn, as well as simultaneously deconfiguring one lane in both regions, while monitoring the impact on performance and power. Symbiotic deconfiguration is performed if the Power Performance Ratio (PPR) of a deconfiguration is greater than a threshold, which was empirically determined to be 2. If multiple deconfigurations meet the PPR threshold, then the deconfiguration with the highest PPR is chosen. Since symbiotic deconfiguration decisions are local to each core, the sampling takes place in parallel on all cores. Thus, the sampling time remains constant regardless of the number of cores.

5.2.4.2 Decision Algorithms

For small (four core) CMPs, the calculation of the optimal combination of performance boosting techniques is computationally feasible, and we discuss this approach in the results section. For larger CMPs, such an approach is intractable, and we therefore rely on heuristic optimization techniques.

The Power Transfer Runtime Manager must solve the constrained integer global optimization problem of maximizing CMP performance under a given power budget. The objective function to be maximized is the performance relative to a baseline without PowerTransfer, given by equation (5.1):

$$f(\vec{x}) = \sqrt[N]{\prod_{i=0}^{N-1} \frac{BIPS(x_i)}{BIPS(baseline_i)}} \tag{5.1}$$

where N is the number of cores, $\vec{x}$ is a vector of size N consisting of the current configuration for each core, x_i is the i^{th} core on the chip running the current configuration, $BIPS(x_i)$ is the BIPS of the i^{th} core, and $BIPS(baseline_i)$ is the BIPS of the i^{th} core running on the baseline.

The objective function further has the constraint of meeting a certain power budget, so Deb's constraint handling method [22] is employed to differentiate between feasible (under power budget) and infeasible (over power budget) solutions. This type of constraint handling penalizes configurations that consume more power than allowed, thus ensuring that infeasible solutions are never chosen over feasible solutions. The final function to be maximized has the form:

$$F(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } g(\vec{x}) <= maxPower \\ 1 - g(\vec{x}) & \text{if } g(\vec{x}) > maxPower \end{cases} \tag{5.2}$$

where $g(\vec{x})$ is the constraint violation function and is defined as the current power consumption of the entire core: $g(\vec{x}) = \sum_{i=0}^{N-1} Power(x_i)$.

The solution for the objective function is the vector $\vec{x}$, the configuration of each core that results in the best global performance. With C possible configurations for each core and N cores, $\vec{x}$ can take C^N values. For a four core CMP, an exhaustive exploration of the space is computationally feasible, but infeasible for larger CMPs.

As the number of cores increases, the search space becomes extremely large due to combinatorial explosion. Moreover, the solution vector $\vec{x}$ consists of discrete rather than continuous variables, which makes it difficult to solve the objective function using classical mathematical techniques such as derivative or limit based methods. Another limiting factor is the need for relatively frequent reevaluation

in order to adapt to the dynamically changing behavior of the scheduled running applications.

Heuristic algorithms are attractive due to their efficiency and effectiveness in searching complex and unknown spaces, and their computational performance can be adjusted by limiting the number of objective function evaluations at the expense of solution accuracy. Two widely used heuristic algorithms are Simulated Annealing and the Genetic Algorithm, which operate by using information gathered from past searches about an unknown space to bias future searches towards more useful subspaces. The next subsections detail the two algorithms that were modified to suit our objective function and search space.

5.2.4.2.1 Integer Coded Genetic Algorithm The Genetic Algorithm was described previously in Section 4.2.2.

Parameters: We empirically explored a variety of parameter values and chose a population size of 20 individuals, a crossover probability of 0.9, and a mutation probability of 0.7. We run the simulation for 25 generations (which corresponds to 500 Objective Function evaluations) as a compromise between algorithm accuracy and a computation time of less than 1% of the time quantum for large CMP configurations.

5.2.4.2.2 Simulated Annealing Simulated Annealing [46] is based on a representation of the annealing technique in metallurgy, where a material is kept at a temperature for a period of time and then the temperature is dropped at distinct points. Our heuristic Simulated Annealing algorithm accepts random solutions at high 'temperatures' (beginning of simulation) in order to avoid becoming stuck in

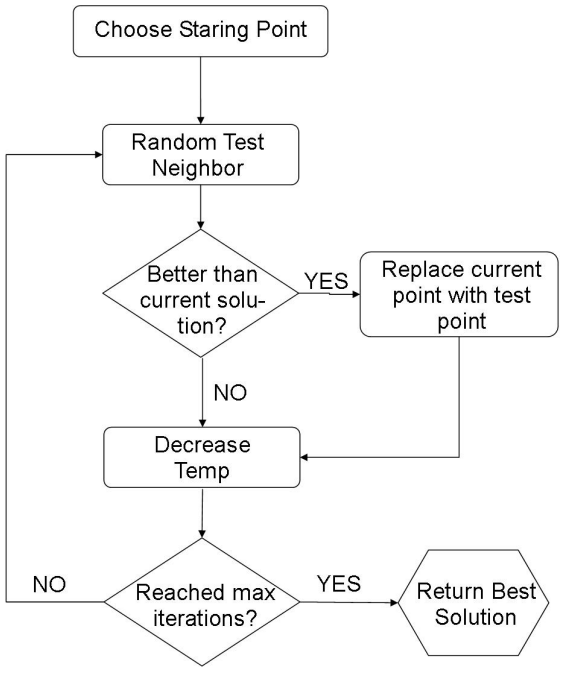


Figure 5.2: Simulated Annealing algorithm.

a local optima, and behaves more like a greedy algorithm as the temperature is decreased. The high-level algorithm operation is shown in Figure 5.2. At each iteration, a random neighbor of the current solution is chosen and evaluated. If the neighbor objective function value is higher than the current one, the move is automatically accepted and the neighbor becomes the current solution. If the neighbor objective function value is lower than the current one, the 'uphill' move is accepted with probability P as shown in equation (5.3).

$$P = e^{-\frac{F(\vec{x}_{neighbor}) - F(\vec{x}_{current})}{T}} \tag{5.3}$$

Here, $F(\vec{x})$ is the objective function value for the solution vector $\vec{x}$, and T is the current temperature value. For every iteration, the temperature is computed as $T_k = \alpha T_{k-1}$, where k is the iteration number and α is a simulation parameter that can take values between 0 and 1.

Parameters: We empirically explored a variety of parameter values through offline simulation and chose the following for online simulations: an initial temperature of 170, an α value of 0.9966, and a neighborhood size of 6. As with the Genetic Algorithm, we limited the online simulation to 500 iterations.

5.3 Results and Discussion

5.3.1 Methodology

To evaluate PowerTransfer, we use the same approach and simulation infrastructure as the one presented in Section 4.3.1.

We use this baseline presented in the same chapter to model 4, 8, 16, and 32 core CMPs, where each core runs one of 13 SPEC CPU2000 benchmarks. We fast-forward each benchmark five billion instructions and run for a total time of 100ms, the granularity at which we periodically engage PowerTransfer. We create 20 randomly chosen four-benchmark workloads that run on 20 four-core configurations, each with a random single fault chosen from the three possible coarse-grain errors (FE, BE, LSQ). For 4- and 8-core CMPs, benchmarks are not repeated for any given workload as this would tend to accentuate the benefit of our approach, but the same fault may occur in more than one core. For 16- and 32-core CMPs, we randomly repeat some benchmarks since the total number of cores is greater than the number of available benchmarks.

All Heuristic Algorithms were written in C++ and compiled with the "-O3" flag. Due to the stochastic nature of the Heuristic Algorithms, each was run 10

times for each configuration and the results averaged. All the results exhibit a relatively tight distribution, so additional trials were unnecessary.

We use a sampling interval of 4ms within a 100ms time quantum. Also, to address the possibility that the configurations deemed best in the sampling phase could exceed the power budget in a 100ms quantum, the GPM uses DVFS to reduce the frequency and voltage when the power budget might be exceeded.

To evaluate the PTRM, we assume that the power budget for any particular benchmark-core combination is the total power used by that benchmark on that core in the absence of faults. We also assume that the maximum CMP power budget is the sum of the power of all current benchmarks running on the cores in the absence of faults. This approach avoids accentuating our improvements due to an artificially high chip-wide power budget. We evaluate our PowerTransfer architecture with respect to a CMP with the initial random errors without symbiotic deconfiguration or performance boosting.

5.3.2 Comparison with Core Sparing

We qualitatively address both the benefits and disadvantages of PowerTransfer over much simpler approaches to fault tolerance such as using spares at the core level. In the latter proposal, upon detection of a fault (whether due to manufacturing or wear-out), the entire core that contains the fault is taken offline. For the following example, we look only at the performance of the microprocessor at shipment time, but it is important to note that the addition of wearout failures makes the case for PowerTransfer even stronger.

The efficacy of the core sparing technique is reliant on the availability of more

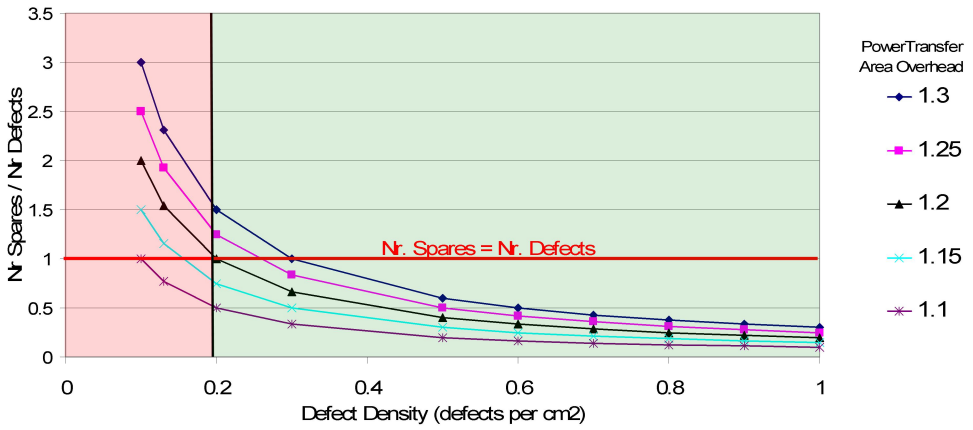


Figure 5.3: Comparison of PowerTransfer and Core Sparing with respect to manufacturing defect density. The red area (left) represents defect densities at which Core Sparing maintains peak performance at lower overhead than PowerTransfer. The green area (right) corresponds to defect densities at which Core Sparing is unable to maintain the same performance as PowerTransfer.

spares than defects. If there are more defects then available spares, the performance of core sparing decreases rapidly. If there are more spares than defects, the performance of the system remains unchanged. We show in the following sections that PowerTransfer is able to match a fault free system even in the presence of a fault in every core simultaneously. In order to compare core sparing with PowerTransfer, we assume that the total area of the two is equivalent. Therefore, core sparing will have S spares available, where S is the area overhead of PowerTransfer divided by the area of one core. Figure 5.3 shows the ratio of spare cores to number of defects for a range of defect densities and PowerTransfer area overheads ranging from 10% to 30%. If this ratio is larger than 1, core sparing is able to maintain peak performance. If the ratio is smaller than 1, core sparing incurs significant performance losses. The red horizontal line, corresponding to an equal number of spares and defects, is the breakeven point, where the two architectures (core sparing and PowerTransfer) perform the same for the same area overhead.

For example, if the area overhead of PowerTransfer is 20%, the red area represents the defect densities that can be tolerated with core sparing. As the defect densities rise (green area), only PowerTransfer is able to maintain peak performance. Since defect densities are not released from industry, the takeaway from Figure 5.3 should be trends rather than exact numbers:

- For higher defect densities, PowerTransfer provides better yield than spares;
- For lower defect densities, PowerTransfer can withstand manufacturing defects as well as wear-out defects, whereas spares may be largely used up at product shipment and fail with fewer wear-out failures; and
- The overall lifetime performance of PowerTransfer is higher than that of CMPs with spare cores that fit in the same area.

5.3.3 Performance Loss Due to Pipeline Faults

Figure 5.4 shows single-core performance relative to a fault-free core for 13 benchmarks, each with a single fault in the Front End, Back End, or Load Store Queue, sorted from high performance loss to low performance loss. As expected, the performance varies widely depending on the benchmark-error pair; gcc running on a core with a FE fault loses more than 20% of its performance, while crafty running on a core with a LSQ fault loses less than 1%. In 20% of the cases, the performance loss is greater than 10%. We later show that PowerTransfer recovers the performance lost due to these pipeline faults.

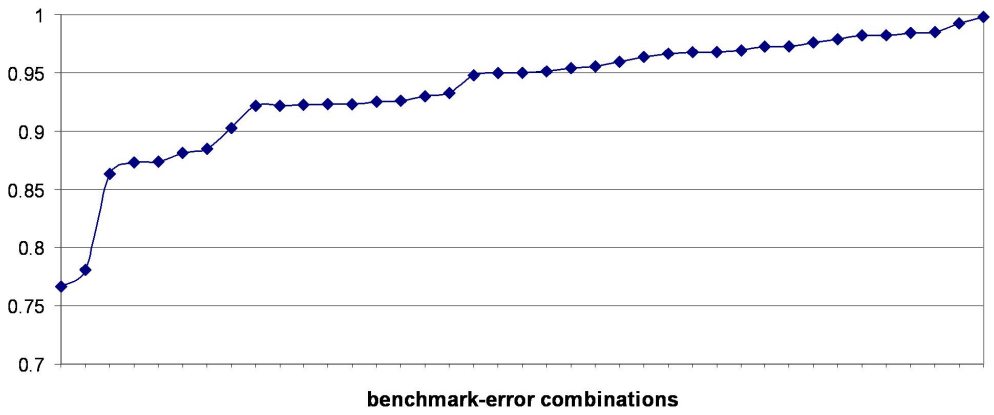


Figure 5.4: Performance relative to a pipeline with no faults for all 39 combinations of benchmarks and single pipeline faults.

5.3.4 Pipeline Imbalance and Symbiotic Deconfiguration

We first evaluate the pipeline imbalance that may occur due to a fault and subsequent deconfiguration, as well as the viability of symbiotically deconfiguring additional functionality to save power with little added performance cost. Symbiotic deconfiguration is effective if in the presence of a fault, the additional deconfiguration yields little additional performance loss relative to the added power savings; this indicates that the fault creates pipeline imbalance that, when corrected through symbiotic deconfiguration, permits significant power to be harnessed relative to the performance loss.

Figure 5.5 shows the performance loss and power savings (due to gating the lane of the affected region) for an initial fault in the LSQ, as well as the effect of symbiotically deconfiguring a lane in the FE. The left solid bars show the performance loss while the right hashed bars show the power savings. The lower subsections of the bars denote the performance cost and power savings from deconfiguring one lane within the faulty region, while the upper stacked subsections show the additional performance loss and power savings by additionally deconfiguring a lane of

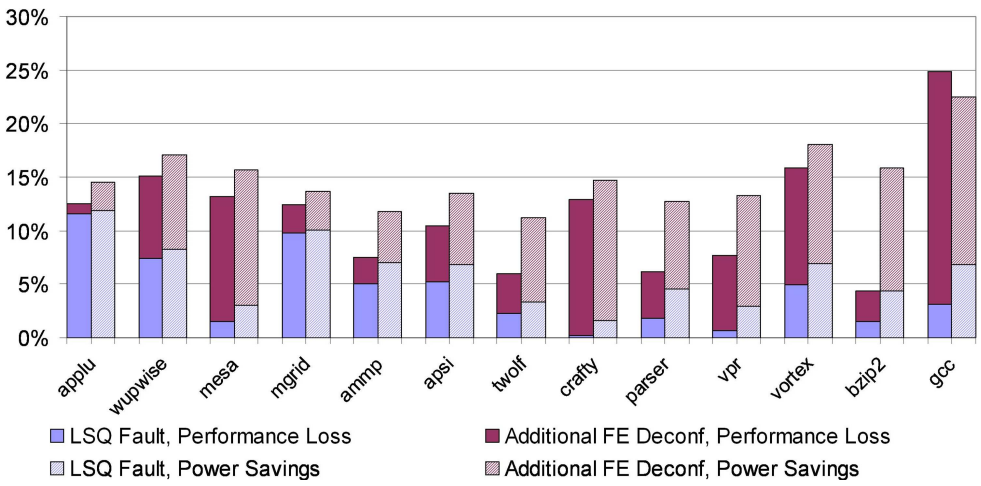


Figure 5.5: Performance loss (left bars) and power savings (right bars) due to an initial fault in the LSQ and with symbiotic deconfiguration of a FE lane.

the FE. Similar results were obtained for FE and BE faults.

We make two major observations from these results. First, the initial deconfiguration due to the faulty unit yields significant performance losses for some benchmarks, but also appreciable power savings in many cases. In most cases, the power/performance ratio is much less than two, indicating that the unit is not overprovisioned to begin with. However, given a fault, the power saved by deconfiguring the affected lane can be used to boost performance by some other means, even without symbiotic deconfiguration.

Second, additional symbiotic deconfiguration can yield a large power savings for a small additional performance loss (much greater than two to one), but for only a subset of the benchmarks (e.g., for bzip2 but not for gcc). Thus, large performance losses can be incurred by blindly deconfiguring additional units without regard for the characteristics of the running application. On the other hand, judicious

symbiotic deconfiguration in cases of pipeline imbalance can be an effective means of harnessing additional power that can be used elsewhere.

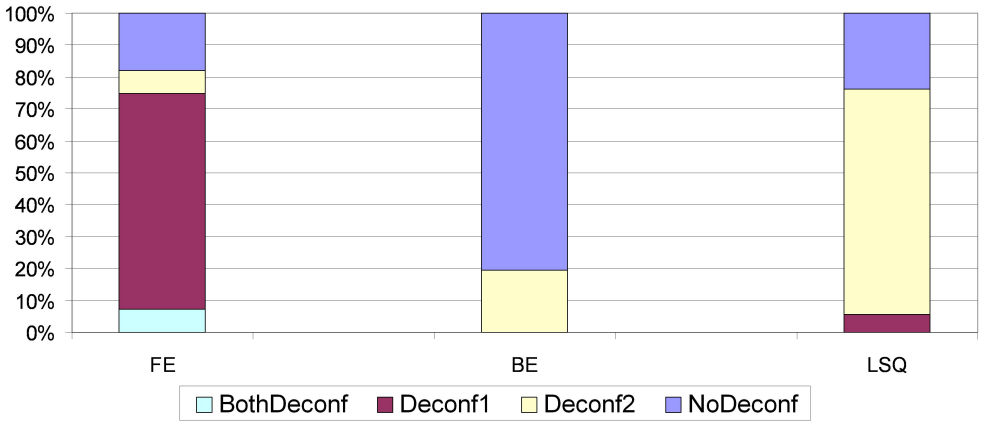


Figure 5.6: Breakdown of symbiotic deconfiguration decisions given a fault in the FE, BE, and LSQ using the Hierarchical Exhaustive algorithm discussed in Section 5.2.4.2.

Figure 5.6 shows the breakdown of the symbiotic deconfiguration decisions made by the Hierarchical Exhaustive algorithm discussed in Section 5.2.4.2 for 32 cores. The three bars correspond to faults in the FE, BE, and LSQ, with each bar showing the percentage of instances where a lane in only one of the non-faulty pipeline regions was deconfigured (Deconf1 and Deconf2), lanes in both regions were deconfigured (BothDeconf), or no symbiotic deconfiguration was performed (NoDeconf). There is no single best deconfiguration decision for any of the three initial faults, and errors in the Front End and Load Store Queue result in a dynamic set of deconfiguration decisions. In the case of an initial error in the Back End, no symbiotic deconfiguration is performed most of the time. This is due to the fact that the Back End has the most power hungry structures, accounting on average for over 50% of the total processor power, while the Front End and Load Store Queue contribute on average 10% and 6%. It is thus more difficult to meet the target power-performance threshold by deconfiguring lanes in the Front End (and

Load Store Queue to a lesser extent) without crippling processor performance.

5.3.5 Performance Boosting Techniques

In this section, we evaluate the characteristics of the three performance boosting techniques described in Section 5.2. We first evaluate the two cache hierarchy boosting techniques that use the predictor described in Section 5.2.2, and then we compare the three performance boosting techniques in terms of performance and power.

There are three possible ways to combine the two cache hierarchy boosting techniques: accessing both the L2 tag and data in parallel on reads (denoted as L1L2SeqL2Par in our graphs), speculatively sending all L1 cache requests concurrently to the L2 cache (L1L2ParL2Seq), and employing both techniques at the same time (L1L2ParL2Par).

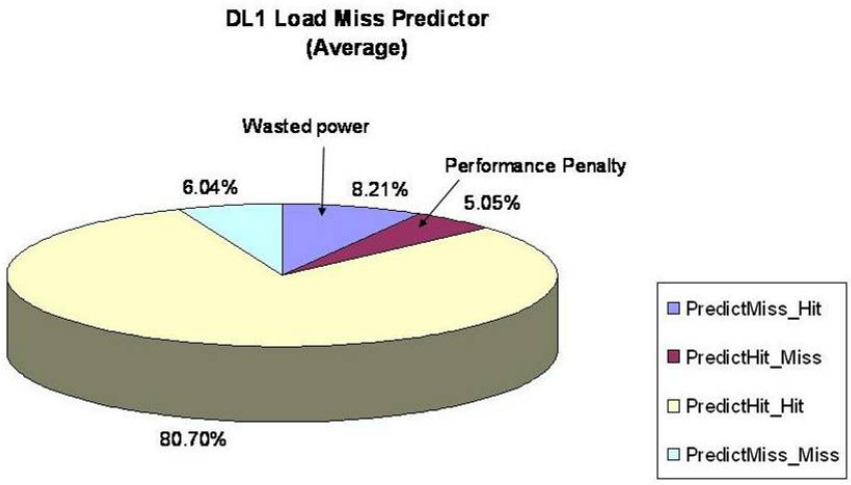


Figure 5.7: L1 cache Load Miss Predictor accuracy.

The main drawback is the substantial increase in power usage, which amounts

to increasing a core's power usage by approximately 60% on average. As explained in Section 5.2.2, we employ a simple Load Miss Predictor for each of the L1 caches that has minor area/power impact and greatly improves the power consumption. Figure 5.7 shows the performance of the L1 data cache Load Miss Predictor (the predictor in the instruction L1 Cache has an even smaller misprediction rate). 86.7% of the L1 accesses are correctly predicted. The mispredictions come in two flavors: 5.1% of the total accesses are predicted as a hit but actually miss in L1, effectively serializing the L1-L2 lookup. The other 8.2% of the L1 cache accesses are predicted as a miss but actually hit in L1, which means that the power used to perform the lookup in the L2 cache is wasted. However, employing a two bit predictor in each of the data and instruction L1 cache reduces the overall additional L2 cache power by 90%.

Figure 5.8 shows the percent performance improvement and percent power increase for the three techniques for each benchmark. The speculative cache access technique proves beneficial only for a few of the benchmarks, with crafty receiving the most gain (29%). Even with the predictors, crafty needs over 25% more power in order to implement speculation in both L1 and L2 cache levels. This is a significant amount of power, and the cores running the targeted applications would not meet the power budget if it is statically turned on at product shipment. It is important to note that no technique is superior across all benchmarks. While Clear has the greatest performance benefit for many benchmarks, Speculative Cache and DVFS are the best techniques for other benchmarks (mesa, crafty, and vortex for the former, and apsi and parser for the latter). Moreover, for some techniques and benchmarks, a large amount of the chip-wide harnessed power is necessary to engage the technique. The decision of which combination of performance boosting techniques to engage is workload dependent and cannot be based on performance

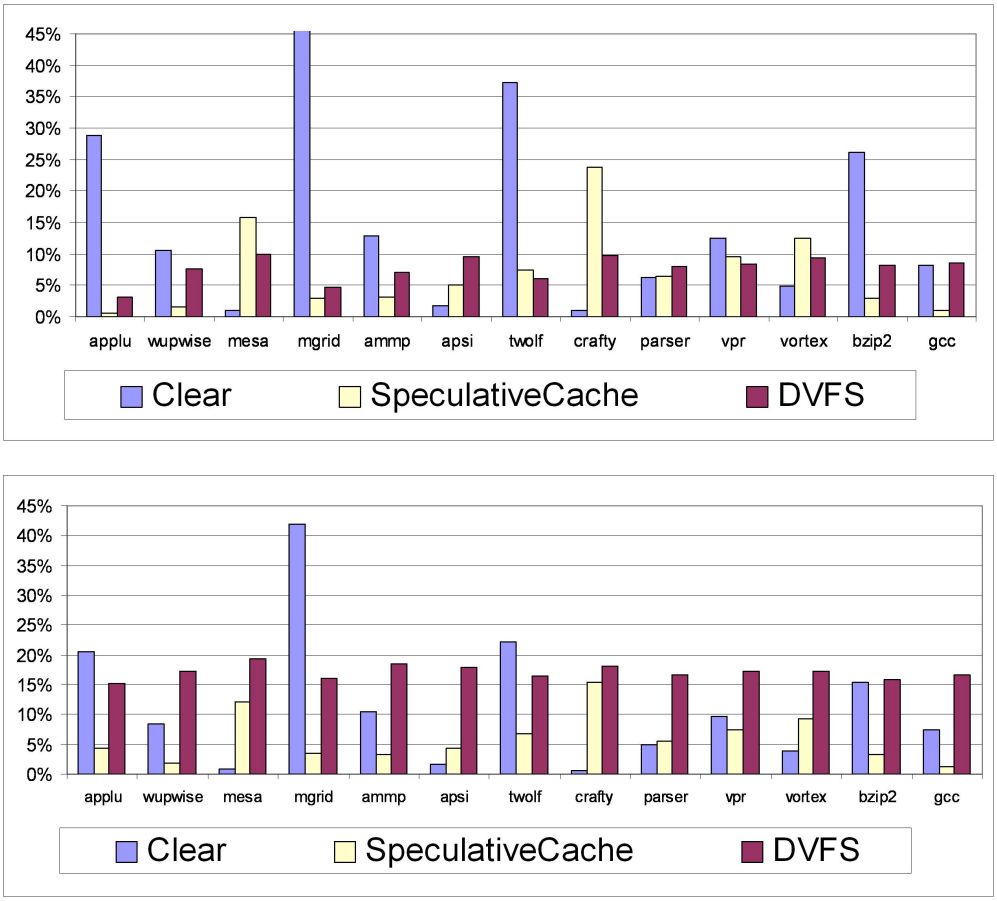


Figure 5.8: Performance improvement (top) and power cost (bottom) for the performance boosting techniques.

alone, but rather the combination of techniques that will yield the largest chip-wide performance gain given the available harnessed power.

However, the decision cannot be based on performance alone, but rather which techniques will yield the highest chip-wide performance gain given the available power. The Power Performance Ratio (PPR), the ratio of the percent power increase to percent performance gain (relative to no boosting) is an effective metric for making this decision. The smaller the PPR, the more efficiently the technique uses the accumulated power. Figure 5.9 shows PPR ratios for the three proposed boosting techniques. As mentioned previously, DVFS has a high power cost relative

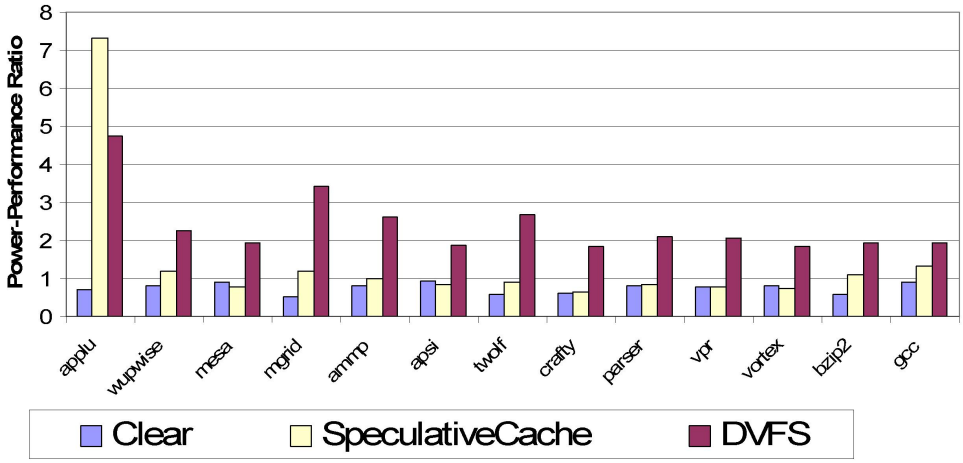


Figure 5.9: PPR of the three boosting techniques.

to the performance gained when voltage and frequency are increased. Speculative Cache and Clear often provide a more favorable power to performance ratio than DVFS. Therefore, as we explain in Section 5.3.7, we first use the available harnessed power for Speculative Cache and Clear and then any remaining power is used to engage DVFS.

5.3.6 Fundamental Trade-offs

In order to gain insight into the effectiveness of the different performance boosting techniques, the interactions between symbiotic deconfiguration and performance boosting, and the impact of locally versus globally managing the accumulated power, we developed a number of offline PowerTransfer managers. Given a set of initial deconfigurations (due to hard faults) and applications for the four cores, these managers have *a priori* knowledge of the performance benefits and power cost tradeoffs for each possible symbiotic deconfiguration and performance boosting possibility. We model this perfect knowledge by calculating the global BIPS for

all combinations of symbiotic deconfigurations and power boostings for the same 100ms time quantum, and pick the configuration that maximizes the geometric mean of all the cores' BIPS with respect to the baseline. We choose the geometric mean in order to avoid overly penalizing low IPC applications to benefit high IPC ones.

For this study, we evaluate 100 random 4-core CMP configurations. Each configuration consists of 4 random benchmarks-error combinations. All results are with respect to the same 4-core configuration with errors but without PowerTransfer.

5.3.6.1 Single versus Multiple Performance Boosting Techniques

Figure 5.10 compares the improvement in throughput for 100 four-core configurations with random initial errors for a manager that globally employs only DVFS, one that globally employs all three performance boosting techniques (DVFS, Speculative Cache Access, and Clear), and one that locally employs all three techniques, i.e., any harnessed power from a core is only applied to boosting that core's performance.

The results of Figure 5.10 confirm the intuition from Figure 5.8 that DVFS is not sufficient to reap the available performance benefits. Rather, a number of techniques in combination is necessary to boost different application classes. With all three techniques, the chip-wide throughput is improved on average by 22.2%, while using DVFS as the sole boosting technique achieves only a 6.3% average increase in chip-wide throughput. The individual Speculative Cache and Clear techniques also fall far short, increasing chip-wide throughput by an average of 9% and 13%, respectively.

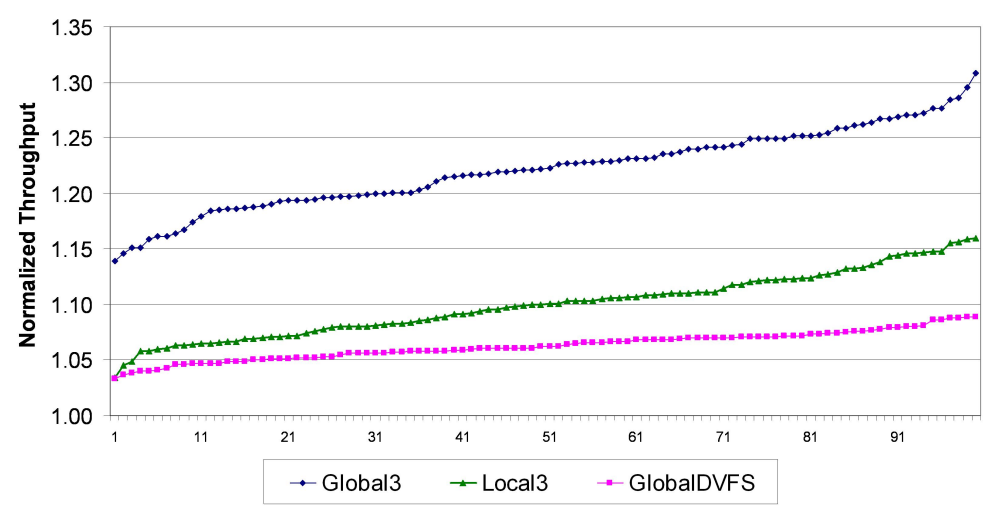


Figure 5.10: Throughput improvement with only DVFS used globally (GlobalDVFS), with all three boosting techniques used globally (Global3), and all three techniques used locally (Local3).

140 FE Errors

	GlobalDVFS	Global3	Local3
BEDeconf	63	98	66
LSQDeconf	18	20	9
NoDeconf	59	22	65

144 BE Errors

	GlobalDVFS	Global3	Local3
FEDeconf	2	13	15
LSQDeconf	22	67	15
NoDeconf	120	64	114

116 LSQ Errors

	GlobalDVFS	Global3	Local3
FEDeconf	7	4	5
BEDeconf	49	85	62
NoDeconf	60	27	49

Table 5.1: Symbiotic deconfiguration decisions given an initial error, the available boosting techniques, and whether decisions are made locally or globally.

	GlobalDVFS	Global3	Local3
DVFS	345	47	127
SpecCache	0	363	281
CLEAR	0	332	164

Table 5.2: The number of times that each boosting technique is engaged given the available boosting techniques and whether decisions are made locally or globally.

Table 5.1 shows the deconfiguration decisions as a function of the initial error, the available boosting techniques, and whether the decision is made locally or globally. Note that the symbiotic deconfigurations made by the managers are highly application and fault dependent. For example, out of the 100 initial 4-core configurations (total of 400 cores), 140 of them were randomly picked to have a Front End fault. In 98 of the 140 cases the best decision is to symbiotically deconfigure the Back End, in 20 cases the LSQ is deconfigured, and in 22 cases no symbiotic deconfiguration is performed. While there is often a bias towards the symbiotic deconfiguration of one region over another depending on the initial fault, it is not a clear-cut decision to engage symbiotic deconfiguration all the time.

When DVFS is used as a standalone performance boosting technique, symbiotic deconfiguration is performed less often because DVFS alone cannot compensate for the performance loss due to symbiotic deconfiguration, even if the power savings are high. Table 5.2 shows how often the three performance boosting techniques are used. When all three techniques are available (Global3), DVFS is engaged on only 47 out of a possible 400 occasions, due to its high power/performance ratio (Figure 5.9). On the other hand, the Speculative Cache and Clear techniques are enabled 363 and 332 times, respectively. Overall, DVFS contributes only 1% of the 22% chip-wide improvement of Global3, making it a "last resort" technique, used mainly to bring the total power usage as close to the maximum power budget

as possible. However, DVFS has the benefit that it can be run at multiple power and performance levels, whereas the other two techniques cannot.

5.3.6.2 Local versus Global Optimization

We assess the benefit of accumulating a global pool of power and applying it to the best combination of chip-wide boosting techniques by comparing Global3 with Local3, in which each core makes local symbiotic deconfiguration and boosting decisions. The 10% average throughput improvement for Local3 is significantly less than the 22.2% average improvement achieved by accumulating and distributing the power globally.

When decisions are made locally rather than globally, less symbiotic deconfiguration occurs (Table 5.1). This is due to the fact that the performance lost through deconfiguration cannot be made up as readily by Speculative Cache Access or Clear because many times the local power budget does not allow the activation of one or both of these techniques. With local management, DVFS is activated more frequently because of its ability to adjust the power usage level in small increments. Even though Clear is enabled for nearly half of the applications, these are not the applications for which Clear provides the most performance benefits (due to the power envelope restrictions).

5.3.6.3 Symbiotic Deconfiguration Advantages

Figure 5.11 compares two PowerTransfer designs. Global3 represents PowerTransfer where additional power is saved through symbiotic deconfiguration, and the performance boosting decisions are globally made at the chip rather than core

level. NoSymbioticDeconfiguration corresponds to a design where the global pool of power is accumulated only from the deconfiguration of the lane with the initial error (without symbiotically deconfiguring other lanes). Like Global3, it also engages all three performance boosting techniques at the global level. The results shown correspond to the worst performing configuration, best performing configuration, and the average throughput improvement over 100 configurations for both designs. In the worst case, reallocating power without regard for the pipeline imbalance resulting from deconfiguration results in an inefficient design, with modest performance improvement (5%) over simply isolating the initial error. On the other hand, addressing the pipeline imbalance results (in the worst case) in a substantial throughput improvement of 14%. In the case where the pristine performance of each of the four cores is crippled by the initial errors, both PowerTransfer with and without symbiotic deconfiguration perform well by redistributing the power to portions of the CMP that most need it, as reflected in the maximum throughput improvements of 31% and 27%, respectively. This is due to the fact that in some cases, symbiotic deconfiguration is not performed because it is not attractive, as seen in Figure 5.5. On average, PowerTransfer with symbiotic deconfiguration performs considerably better than without symbiotic deconfiguration (throughput is improved by 22% versus 16%).

5.3.6.4 Reduction of Complexity - Decoupled Decisions

Finally, we implemented an offline manager that decouples the symbiotic deconfiguration decision from the boosting decision. The manager first accumulates the largest amount of power possible using only symbiotic deconfigurations with a PPR of at least 2 (i.e., a minimum 2% power accumulation for a 1% performance loss). The manager then finds the combination of boosting techniques that maximizes

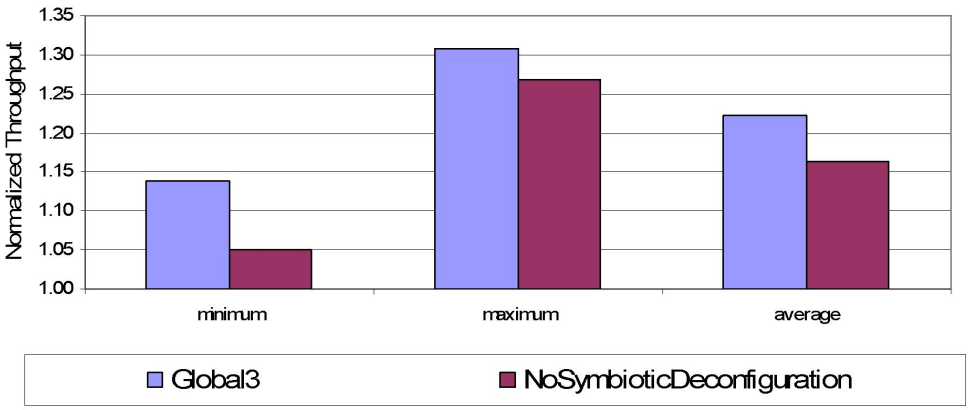


Figure 5.11: Normalized throughput improvement using PowerTransfer with symbiotic deconfiguration (Global3) and without symbiotic deconfiguration (NoSymbioticDeconfiguration).

performance within this accumulated power budget. We found that this decoupled offline manager achieved an average performance of 20.4%, which is very close to the 22.2% achieved by Global3. This simplification works well since the PPR is a good proxy for the effectiveness of the Speculative Cache and Clear boosting techniques. Since each approach incurs a baseline power cost, the PPR is a good indicator of whether a particular application will achieve good performance relative to the power cost for those techniques. A PPR of 2 also works well for DVFS since this threshold helps to distinguish memory-bound and CPU-bound applications (Figure 5.9).

The results from this section demonstrate that in order to reap the full benefits of PowerTransfer:

- Symbiotic deconfiguration decisions must account for the characteristics of the running applications; however, these decisions can be decoupled from the decisions of which boosting techniques to engage;

- Alternative CPU boosting techniques with a better PPR than DVFS should be used;
- Multiple performance boosting techniques should be implemented to account for a range of application types;
- A global pool of power should be accumulated and distributed to boosting techniques in a global fashion.

5.3.7 Power Transfer Runtime Manager

Previous sections presented the fundamental characteristics and limits of the PowerTransfer architecture by predicting results with *a priori* knowlege over the entire evaluation interval. In this section, we present overall results for the PTRM, a practical implementation of the manager. As explained in Sections 5.2.4 and 5.3.6.4 the PTRM first samples the three possible symbiotic deconfigurations, after which it makes a deconfiguration decision based on the Power-Performance Ratio. The symbiotic deconfiguration decisions are local decisions that are made for each core, and as such, the computation time remains constant for any CMP size. It then samples the system with engaged performance boosting techniques and determines the best power reallocation and the configuration that maximizes the global throughput improvement within the power budget. As we show in Section 5.3.7.2, the computation time required to select the best combinatorial chip-wide configuration grows exponentially, and heuristic algorithms are employed to constrain the execution time to less than 1% of the decision interval.

5.3.7.1 4-Core CMP

For four cores, we sample eight boosting configurations, corresponding to the seven combinations of Speculative Cache Access and Clear as well as no boosting. This results in 4^8 (4096) combinations, which can be exhaustively explored within less than 0.5% of the decision interval (denoted by 'Exhaustive' in the figures). Using this information, the best chip-wide configuration that meets the power budget is chosen. The remaining power (if any) is distributed using DVFS across the cores in a greedy fashion: the frequency of the highest IPC core is boosted to its maximum value, then the next highest IPC core, etc, until the power budget is exhausted.

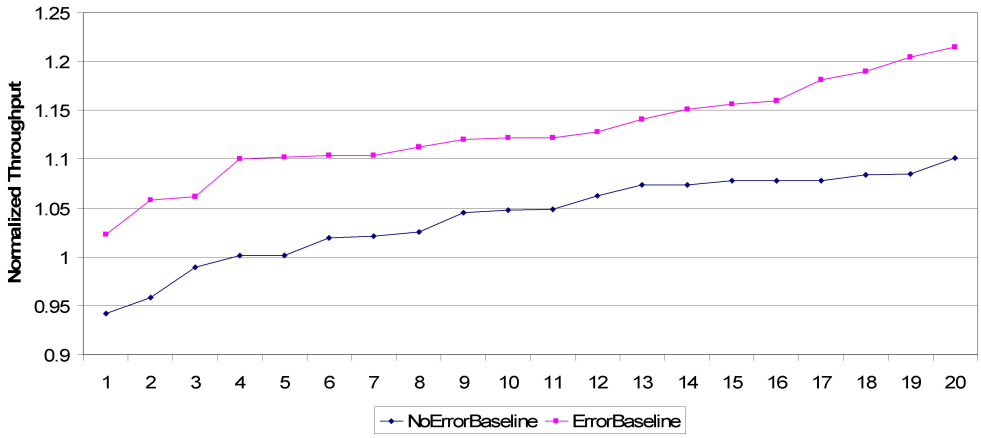


Figure 5.12: Normalized throughput improvement of Exhaustive PTRM for 4-core CMPs with respect to the defect-free CMP (NoErrorBaseline) and to the CMP with random initial errors (ErrorBaseline).

Figure 5.12 shows the relative throughput improvement of PowerTransfer with respect to two baselines. The first, denoted as ErrorBaseline in the figure, corresponds to 20 4-core CMP configurations, each with a random initial error. The second baseline, denoted as NoErrorBaseline in the figure, corresponds to pristine (fault free) CMPs. By addressing the pipeline imbalance created by the error and

redistributing the power in an efficient fashion, PowerTransfer outperforms architectures that simply deconfigure the faulty lane by up to 21.5%, and on average by 12.7%. Moreover, PowerTransfer is able to recoup the lost performance due to the initial faults in all but 3 of the 20 configurations. In those three cases, PowerTransfer is within 5% of the performance of a pristine microprocessor. Interestingly enough, in almost half the configurations, PowerTransfer outperforms the fault-free microprocessor by more than 5%. This is a strong indication that power inefficiencies exist even in fault-free microprocessors, making a case for the application of PowerTransfer even in the absence of faults.

5.3.7.2 Scalability Study

As the number of cores is increased, it becomes no longer feasible to exhaustively explore the entire combinatorial space. For 8 cores, the computation time is on average 2.5 seconds, which is 25 times longer than the time quantum at which decisions need to be made[2]. For 16 and 32 cores, the computation time becomes 1.36 years and $3.84 * 10^{14}$ years for each of the 20 initial configurations, which is obviously not computationally feasible.

In order to establish an upper bound estimate for 16 and 32 cores, we implemented a Hierarchical Exhaustive algorithm that groups a subset of the cores into regions, with 8 cores in each region. Exhaustive search is performed within each region and the results are combined to obtain a global performance improvement. In order to avoid pathological core groupings for each region, we performed four separate tests, each with a different core to region allocation. The computation time for the Hierarchical Exhaustive algorithm was 5.65 seconds for 16 cores and

[2]However, we compute the best solution as an upper bound for comparison purposes.

12.6 seconds for 32 cores. Although this comprises over 50 times the length of a decision interval (Figure 5.13), which makes it impractical as a Decision Algorithm technique, it allows us to obtain near-oracle results for these larger CMP configurations to compare against the Heuristic Optimization algorithms.

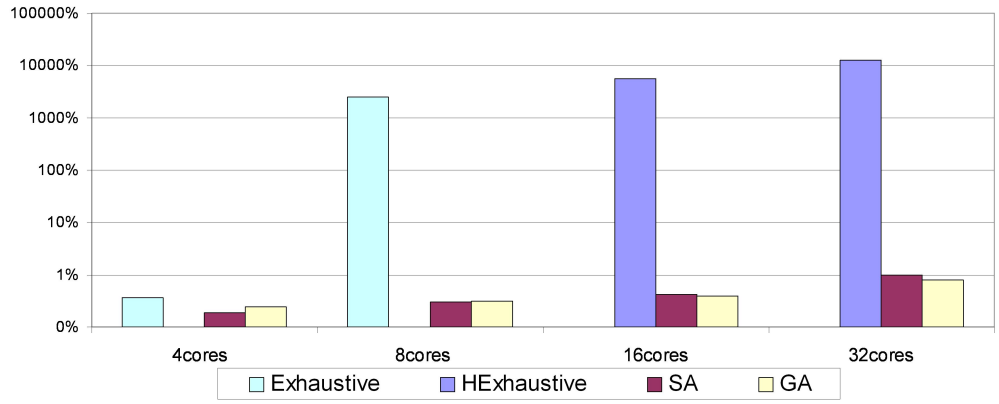


Figure 5.13: Computation time as a percentage of the decision interval for the Exhaustive and Heuristic Optimization algorithms (logarithmic scale).

Figure 5.13 also shows how the computational time as a percentage of the total decision interval time scales with the number of cores for the Genetic Algorithm (GA) and Simulated Annealing (SA). Although the number of objective function computations stays constant for all CMP sizes, the time taken to evaluate one objective function value increases, since it requires computing the BIPS improvement for each core. Still, the computation time of both heuristic algorithms scales extremely well with the number of cores. For 32 cores, the algorithm computation time comprises only 1% of the decision interval.

Figure 5.14 compares the average overall performance improvement of Exhaustive/HExhaustive, GA, and SA with respect to a CMP that deconfigures the faulty lane without employing PowerTransfer. Figure 5.15 compares the different approaches case by case for each of the 20 random initial configurations for a 32-core CMP. Both heuristic algorithms nearly match the performance of the exhaustive

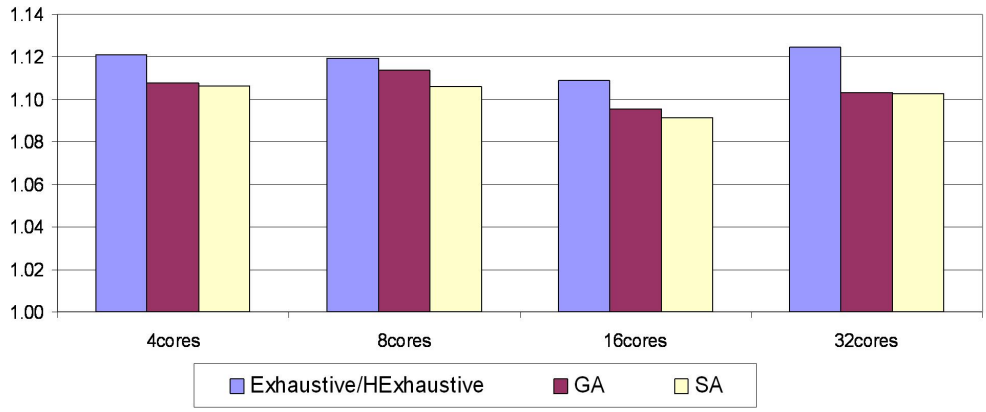


Figure 5.14: Normalized performance improvement over the deconfigured CMP without Power Transfer.

algorithms irrespective of the number of cores. Comparing the two heuristic algorithms results in some interesting observations. For four cores, both heuristic algorithms are able to search most of the solution space, performing on average almost identically. As the search space is increased, the Genetic Algorithm slightly outperforms Simulated Annealing. However, as the number of cores increases to 32, both algorithms are able to evaluate a smaller subset of the possible solutions, and their average performance becomes matched again. These results indicate that the search space may have many similar local optima that both algorithms choose once the search space becomes too large to completely explore in the given time.

Overall, our results show the following:

- Blindly applying symbiotic deconfiguration without accounting for application characteristics may lead to significant performance loss with little power accumulation;
- The selective application of symbiotic deconfiguration through systematic sampling can harness significant power at comparatively small performance cost;

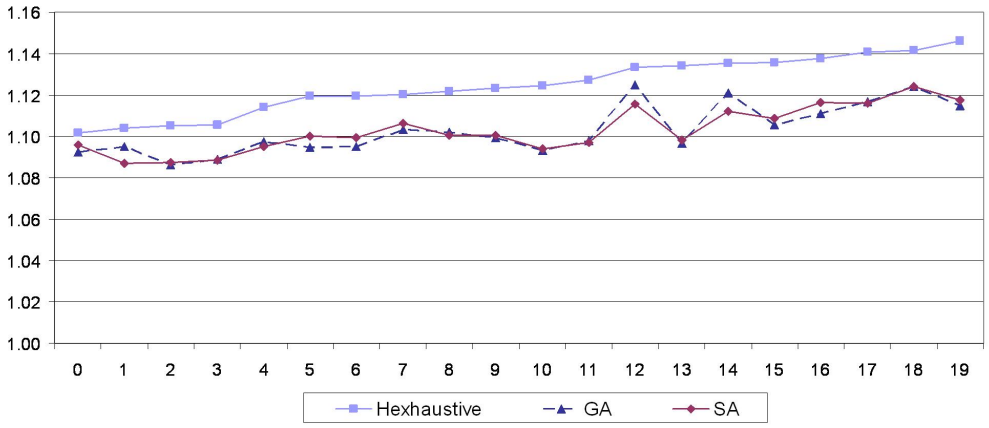


Figure 5.15: PTRM performance of 20 random initial configurations of a 32-core CMP compared to a CMP without PowerTransfer. The HExhaustive results are sorted from lowest to highest performance gain. The GA and SA results match the corresponding HExhaustive configuration.

- The use of several performance boosting techniques that can be applied to different application types is much more effective than the use of a single technique such as DVFS;
- PowerTransfer is able to fully recover the performance lost due to pipeline faults, and the heuristic algorithms provide good scalability to large-scale CMP systems.

5.3.7.3 Sampling Interval Considerations

There is a large difference between the predicted results from Section 5.3.6 and the realistic PTRM. Comparing Figures 5.10 and 5.12, the maximum average throughput improvement (obtained with a priori knowledge of the entire decision interval) is 22.2%, whereas the PTRM average throughput improvement is 12.7%. A small fraction of this difference is attributed to the decision to decouple symbiotic deconfiguration decisions from performance boosting decisions. However, an oracle

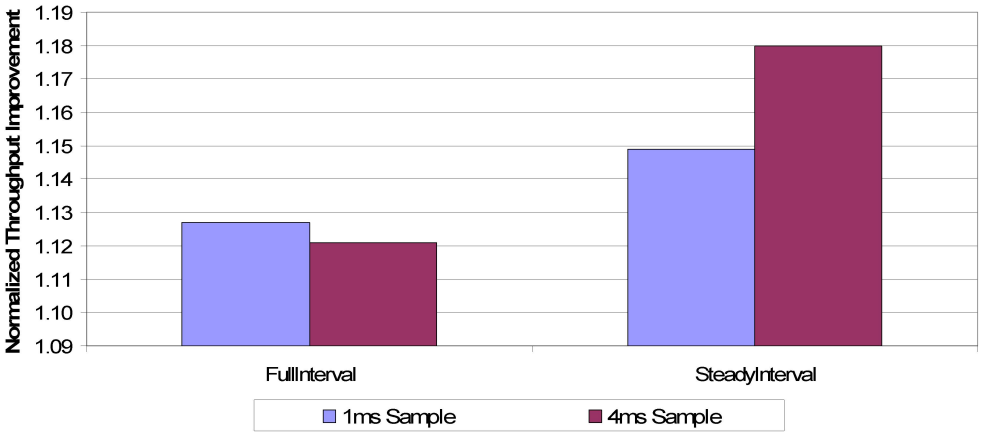


Figure 5.16: Throughput improvement over the error baseline for the full decision interval and the steady interval with different sample durations.

decoupled manager still obtains on average a 20.4% throughput improvement, so most of the difference between a realistic and an oracle manager comes from the inaccuracies in sampling. The proposed Power Transfer Runtime Manager runs a variety of configurations for short periods of time (samples), and then selects a configuration to be run during the remaining time (steady interval). If the samples are too short, they do not accurately forecast the behavior of the entire decision interval but less time is spent sampling inefficient configurations. On the other hand, long samples predict long-time behavior much more accurately, but the overall performance over the decision interval is reduced due to the long time spent sampling suboptimal configurations.

To demonstrate this point, Figure 5.16 shows the throughput improvement during both the entire decision interval (including the sampling phase penalty) and during the steady interval (which just shows performance of the best found configuration) for two sampling interval sizes. The left bars correspond to individual sample intervals of 1ms that were used throughout the proposal, which sums up to

a total 10ms (or 10% of the 100ms decision interval) spent in sampling phase. The remaining 90% of the time is spent running the predicted best configuration. The right bars show results for 4ms samples (corresponding to a total of 40 ms spent in sampling), resulting in only 60% of the decision interval spent running the predicted best configuration. The longer 4ms samples better predict long-term behavior as seen from the much higher throughput improvement over the error baseline (an average of 18%) during the steady interval. At the same time, the shorter samples result in a steady interval performance improvement of less than 15%. However, taking into account the entire decision interval, including the performance during the sampling period, the algorithm that uses 1ms samples marginally outperforms the 4ms samples because it spends most of the time running a good configuration.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

We introduce a novel lane-based modular architecture where cores are homogeneously designed and dynamically reconfigured to addresses both power and reliability concerns.

6.1 Power Limitations

The era of Dark Silicon will produce a disconnect between the number of devices that can be integrated on a die and the amount of chip power that can be economically supported. As such, the portion of a CMP that can be turned on at any given point will be limited in order to meet chip-wide power constraints. We demonstrate that there is a clear need for more sophisticated architectures and control algorithms as the power is increasingly constrained according to scaling predictions. We show limitations of currently used power management techniques and how a modular architecture can be used to complement existing techniques in an architects tool box when those techniques are not adequate for high performance computing.

To this end, we customize complex optimization techniques to the reality of online adaptivity. First, we introduce a formal methodology for dynamically characterizing application behavior using surrogate response surfaces. Second, we reduce the number of expensive sampling evaluations and improve their accuracy through methodical experimental design. Our architecture applies these techniques and takes advantage of the variety in application characteristics to tailor the underlying hardware to system goals by redistributing power among the cores of a chip

multiprocessor. The resulting average performance gains range from 30% to 60% over static architectures with the same core count and core strength that disable cores to meet the power budget, and average performance improvements of 10% to 24% over architectures that engage DVFS to meet the power budget.

6.2 Reliability Issues

Future CMPs built-in highly-scaled technologies also face the prospect of having to deconfigure hardware units in the face of manufacturing defects and aging-related faults. Such deconfiguration may lead to application specific pipeline imbalances that reduce the power-performance efficiency of the formerly well-balanced pipeline. However, this approach is able to maintain functionally correct execution, albeit at the expense of performance. As such, the issue of maintaining peak performance in the face of faults has not been addressed satisfactorily. Specifically, redundancy through the use of core spares incurs significant area overheads, discards considerable functionally correct realestate, and withstands a very limited number of faults.

We leverage our novel lane-based adaptive architecture towards PowerTransfer, a technique that dynamically identifies imbalances and rebalances the pipeline by proactively deconfiguring additional units. Doing so in an application-specific way yields additional power savings at little performance cost. The harnessed power is used to improve chip-wide performance by enabling a combination of performance boosting techniques chosen by a heuristic optimization algorithm. We demonstrate that PowerTransfer is able to fully recover the performance loss due to pipeline faults. We also show that PowerTransfer is scalable to many core systems without

diminishing the performance benefits, and can withstand at least as many faults as cores while maintaining peak operation.

6.3 Future Work

In this section, we describe a number of ways in which the work presented in this dissertation can be extended. The control algorithm methodology proposed in Chapter 4 can be applied to a variety of optimization goals in the context of resource utilization management. These include:

- Minimizing power usage under a certain performance guarantee;
- Fair or priority-based allocation of shared resources;
- Assigning the optimal number of processors for a certain task;
- Power allocation between general purpose cores and specialized hardware such as accelerators.

Our work has focused on multiprogrammed workloads based on existing benchmarks since they represent the challenge of optimizing multiple individual needs rather than the homogeneous tasks of parallel applications. However, the number of workloads available for our simulation infrastructure is limited, and as such our results scaled to many cores only partly show the pontential of our adaptive technique. This study can be extended to include workloads from a variety of industry or open-source standards, such as parallel, emerging, and mixed parallel-sequential workloads.

This dissertation has addressed inaccuracies in sampling and solutions to high frequency noise in Section 4.1.2.3. Low frequency noise is not mitigated by our

work, and may lead to sub-optimal hardware configurations if the applications enter a different phase during the OS time quantum. In order to eliminate low frequency noise, phase detection can be added to our architecture to trigger a re-evaluation before the time quantum ends. In a many core environment with a large number of applications running, this may lead to very frequent re-evaluation and large performance loss by repeatedly sampling suboptimal configurations. In such cases, a hierarchical approach may be employed, performing one chip-wide evaluation as presented in this work, and finer grained re-evaluations for a subset of cores given their initially allocated power budget.

Alternatively, different means for application characterization can be explored to perhaps eliminate the need for sampling. For example, monitoring processor usage patterns such as queue occupancy, port activity, miss rates, and issue rates may provide an alternative means for reconfiguration decisions. Alternatively, it is worth studying whether the adaptive manager can maintain a knowledge base of application charateristics to make informed decisions without the need for repeated sampling.

Finally, we present global optimization solutions based on heuristic algorithms. An extension to this work can explore alternative optimization algorithms. For example, fast non-dominated sorting (Pareto-front classification) can be applied in a two-step process, first locally to reduce the number of core configurations considered for global optimization, and then globally to find the optimal resource allocation.

BIBLIOGRAPHY

[1] Amit Agarwal, Bipul C. Paul, Hamid Mahmoodi, Animesh Datta, and Kaushik Roy. A process-tolerant cache architecture for improved yield in nanoscale technologies. *IEEE Transactions on Very Large Scale Integrated Systems*, January 2005.

[2] N. Aggarwal, P. Ranganathan, N.P. Jouppi, and J.E. Smith. Configurable isolation: Building high availability systems with commodity multicore processors. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2007.

[3] David H. Albonesi. Dynamic IPC/Clock Rate Optimization. In *Proceedings of the International Symposium on Computer Architecture*, 1998.

[4] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically Tuning Processor Resources with Adaptive Processing. *IEEE Computer*, December 2003.

[5] S.S.I. Association. The International Technology Roadmap for Semiconductors. *http://www.itrs.net/links/2009ITRS/Home2009.htm*, 2009.

[6] Todd M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the Annual International Symposium on Microarchitecture*, 1999.

[7] R. Iris Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *International Symposium on Computer Architecture*, 2001.

[8] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2000.

[9] Reinaldo Bergamaschi, Guoling Han, Alper Buyuktosunoglu, Hiren Patel, Indira Nair, Gero Dittmann, Geert Janssen, Nagu Dhanwada, Zhigang Hu, Pradip Bose, and John Darringer. Exploring power management in multicore systems. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 2008.

[10] Jason Blome, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Self-calibrating online wearout detection. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2007.

[11] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, November 2005.

[12] Fred A. Bower, Paul G. Shealy, Sule Ozev, and Daniel J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

[13] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2005.

[14] George E.P. Box and Donald W. Behnken. Some new three level designs for the study of quantitative variables. *Technometrics*, November 1960.

[15] G.E.P. Box and N. R. Draper. Empirical model-building and response surfaces. *John Wiley and Sons, New York*, 1987.

[16] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2000.

[17] James Burns and Jean-Luc Gaudiot. Area and System Clock Effects on SMT/CMP Throughput. *IEEE Transactions on Computing*, February 2005.

[18] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Peter, David Brooks Pradip Bose, and Peter Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the Great Lakes Symposium on VLSI*, 2001.

[19] Alper Buyuktosunoglu, Tejas Karkhanis, David H. Albonesi, and Pradip Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2003.

[20] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient checker processor design. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2000.

[21] De-Shiuan Chiou, Da-Cheng Juan, Yu-Ting Chen, and Shih-Chieh Chang. Fine-grained sleep transistor sizing algorithm for leakage power minimization. In *Proceedings of the Annual Design Automation Conference*, 2007.

[22] K. Deb. An Efficient Constraint Handling Method for Genetic Algorithms. In *Computer Methods in Applied Mechanics and Engineering*, 2000.

[23] G.C. Derringer and D. Suich. Simultaneous optimization of several response variables. *Journal of Quality Technology*, 1980.

[24] Steve Dropsho, Alper Buyuktosunoglu, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, Greg Semeraro, Grigorios Magklis, and Michael L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[25] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, January 1995.

[26] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the Annual International Symposium on Computer architecture*, 2011.

[27] Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Olay: Combat the signs of aging with introspective reliability management. In *Workshop in Quality-Aware Design*, 2008.

[28] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2001.

[29] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. StageWeb: Interleaving Pipeline Stages into a Wearout and Variation Tolerant CMP Fabric. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2010.

[30] S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. Mahlke. The StageNet

Fabric for Constructing Resilient Multicore Systems. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2008.

[31] S. Gupta, S. Feng, A. Ansari, and S. Mahlke. Erasing Core Boundaries for Robust and Configurable Performance. In *Proceedings of the Annual International Symposium on Microarchitecture*.

[32] S. Gupta, S. W. Keckler, and D. Burger. Technology Independent Area and Delay Estimates for Microprocessor Building Blocks. University of Texas at Austin Department of Computer Sciences, Technical Report TR2000-05. 2005.

[33] H.-M. Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 2001.

[34] J. Harrington. The Desirability Function. In *Industrial Quality Control*, 1965.

[35] J. Holland. Adaptation In Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. *Ann Arbor: University of Michigan Press*, 1975.

[36] Shiwen Hu, Madhavi Valluri, and Lizy Kurian John. Effective adaptive computing environment management via dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[37] Michael C. Huang, Jose Renau, and Josep Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2003.

[38] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2007.

[39] 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 Annual International Symposium on Microarchitecture*, 2006.

[40] Anoop Iyer and Diana Marculescu. Microarchitecture-level power management. *IEEE Transactions on Very Large Scale Integrated Systems*, June 2002.

[41] Hailin Jiang, Malgorzata Marek-Sadowska, and Sani R. Nassif. Benefits and

costs of power-gating technique. In *Proceedings of the International Conference on Computer Design*, 2005.

[42] Kunhyuk Kang, Keejong Kim, Ahmad E. Islam, Muhammad A. Alam, and Kaushik Roy. Characterization and estimation of circuit reliability degradation under NBTI using on-line IDDQ measurement. In *Proceedings of the Annual Design Automation Conference*, 2007.

[43] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, March 1999.

[44] S. Kim, S.V. Kosonocky, D.R. Knebel, and K. Stawiasz.

[45] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2008.

[46] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. In *Science, New Series*, 1983.

[47] Nevin Kirman, Meyrem Kirman, Mainak Chaudhuri, and Jose F. Martinez. Checkpointed early load retirement. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2005.

[48] Israel Koren and Zahava Koren. Defect Tolerance in VLSI Circuits: Techniques and Yield Analysis. In *Proceedings of the IEEE*, 1998.

[49] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2003.

[50] R. Kumar and G. Hinton. A Family of 45nm IA Processors. In *International Solid-State Circuits Conference*, 2009.

[51] C. LaFrieda, E. Ipek, J.F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2007.

[52] R. McGowen, C.A. Poirier, C. Bostak, J. Ignowski, M. Millican, W.H. Parks, and S. Naffziger. Power and temperature control on a 90-nm Itanium family processor. In *IEEE Journal of Solid-State Circuits*, 2006.

[53] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2007.

[54] Albert Meixner and Daniel Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[55] R.H. Myers and D. C. Montgomery. Response Surface Methodology: Process and Product Optimization Using Designed Experiments. *John Wiley and Sons, New York*, 1995.

[56] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the International Symposium on Computer Architecture*, 1997.

[57] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the Annual International Symposium on Microarchitecture*, 2001.

[58] R.G. Regis and C.A. Shoemaker. A Stochastic Radial Basis Function Method for the Global Optimization of Expensive Functions. In *INFORMS Journal on Computing*, 2007.

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

[60] Bogdan F. Romanescu and Daniel J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the International Conference on Parallel Architectures and Compiliation*, 2008.

[61] E. Schuchman and T.N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

[62] J. Sharkey, A. Buyuktosunoglu, and P. Bose. Evaluating Design Tradeoffs in On-Chip Power Management for CMPs. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2007.

[63] Kaijian Shi and David Howard. Sleep Transistor Design and Implementation - Simple Concepts Yet Challenges To Be Optimum. In *International Symposium on VLSI Design*, 2006.

[64] P. Shivakumar, S.W. Keckler, CR. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the International Conference on Computer Design*, 2003.

[65] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[66] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the International Symposium on Computer Architecture*, 2004.

[67] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

[68] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

[69] M. Steinhaus, R. Kolla, J. L. Larriba-Pey, T. Ungerer, and M. Valero. Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models. In *2nd Workshop on Complexity Effective Design*, 2001.

[70] Dennis Sylvester, David Blaauw, and Eric Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Journal of Design and Test*, November 2006.

[71] D. Tarjan, S. Thoziyoor, and N.P. Jouppi. Cacti 5.3. *HP Laboratories Palo Alto Technical Report*, 2005.

[72] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, 2008.

[73] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In

Proceedings of Architectural Support for Programming Languages and Operating Systems, 2010.

[74] D. Weiss, J. J. Wuu, and V. Chin. The on-chip 3MB subarray-based third-level cache on an Itanium microprocessor. In *IEEE Journal of Solid-State Circuits*, 2002.

[75] Whitepaper. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. http://download.intel.com/design/processor/applnots/320354.pdf. 2008.

[76] K.C. Yeager. The MIPS R10000 Superscalar Microprocessor, 1996.

[77] M. Yilmaz, S. Ozev, and D.J. Sorin. Low-Cost Run-Time Diagnosis of Hard Delay Faults in the Functional Units of a Microprocessor. In *Proceedings of the International Conference on Computer Design*, 2007.

[78] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. University of Virginia Technical Report CS-2003-05. 2003.