ABSTRACT

NAVADA, SANDEEP SURESH. A Unified View of Core Selection and Application Steering in Heterogeneous Chip Multiprocessors. (Under the direction of Professor Eric Rotenberg).

A single-ISA heterogeneous chip multiprocessor (HCMP) is an attractive substrate to improve single-thread performance and energy efficiency in the dark silicon era. We consider HCMPs comprised of non-monotonic core types where each core type is performance-optimized to different instruction-level behavior and hence cannot be ranked – different program phases achieve their highest performance on different cores. Although non-monotonic heterogeneous designs offer higher performance potential than either monotonic heterogeneous designs or homogeneous designs, steering applications to the best-performing core is challenging due to performance ambiguity of core types.

We present a unified view of selecting non-monotonic core types at design-time and steering program phases to cores at run-time. After comprehensive evaluation, we found that with N core types, the optimal HCMP for single-thread performance is comprised of an "average core" type coupled with N-1 "accelerator core" types that relieve distinct resource bottlenecks in the average core. This inspires a complementary steering algorithm in which a running program is continuously monitored for the distinctive bottlenecks. If the bottleneck signatures of the program and current core diverge, the program is migrated to the core with a matching signature.

Another key challenge in designing a ground-up non-monotonic HCMP is that it takes a lot of time to perform core selection. We propose using detailed bottleneck analysis to accelerate the design space exploration to find the best core for each of the application phases in a benchmark suite. Bottleneck analysis at a given design point provides a localized view of

the region around the design point without performing simulations at the neighboring points. On the other hand, a classical search/optimization technique has a global view of the design space and avoids getting stuck at a local maximum. We use this synergistic behavior between the bottleneck analysis (good locally) and the classical search/optimization techniques (good globally) to accelerate the design space exploration to find the best core for each application phase. Once the best core information is known, the core selection can be quickly performed by employing existing core-subsetting algorithms.

A Unified View of Core Selection and Application Steering in Heterogeneous Chip Multiprocessors

by
Sandeep Suresh Navada

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Professor Eric Rotenberg Committee Chair	Professor Greg Byrd
Professor James Tuck	Professor Vincent Freeh

DEDICATION

To Mom and Dad

BIOGRAPHY

Sandeep Navada received his PhD in the Department of Electrical and Computer Engineering at North Carolina State University. He got his Bachelor's degree in Electrical Engineering from Indian Institute of Technology (IIT), Roorkee, in 2006. He has done summer internships at Intel and IBM. His research interests are in heterogeneous chip multiprocessor design, processor microarchitecture, bottleneck analysis, machine learning and embedded systems.

ACKNOWLEDGMENTS

My six years at NC State University have been really enjoyable and fulfilling. It would not have been without the support, encouragement and help of so many people around me. I am extremely indebted to them.

First and foremost, I would like to thank my mom and dad for their love and encouragement all throughout my life. They have been extremely supportive in all my endeavors and have been great parents. I owe a lot to them. I have been really fortunate to have my brother, Santosh as a roommate during the last two years of grad life. His fun companionship was a nice escape from the grad school grunt.

My advisor, Eric Rotenberg, has been an amazing mentor. He gave me almost full autonomy to work on any topic and, at the same time, gave me valuable advice so that I don't go astray. I have always been amazed by his ability to quickly discern the essence of any research idea and his extreme attention to detail. His energetic personality has definitely pushed me to accomplish much more than I could.

I am thankful to Dr. Greg Byrd, Dr. James Tuck and Dr. Vincent Freeh for agreeing to serve on my committee. Their feedback definitely went a long way in improving my thesis.

CESR is one of the main reasons for making my grad life fun-filled and intellectually stimulating. I enjoyed the long discussions, both technical and philosophical, with Niket, Devesh, Rajesh, Elliot, Brandon, Ahmad, Siddharth, Salil, Hashem, Vimal, Mo, Rami, Ahmad, Anil, Jayneel, Hiran, Tanmay, Sanghoon, Fang, Sangyeol, George, Avik and Saurabh, which made the grad life so lively and enjoyable.

Gaurav has been a great friend and is always a phone call away whenever I need any help. He is the nicest guy I have ever met. Ganesh, Shivam, Harish, Sami and Sashi have been awesome roommates over the course of grad life. Long phone conversations with Subir, Hridish, Jaideep, Bala, Gaurav and Nidhish were always a welcome break during the long grad school toil. I also cherish the time spent in friendly banter with Amit, Abhik, Roopa, Richa and Dyuti.

Linda, Sandy and Kendall were always helpful to clarify all my doubts regarding the administrative process and were extremely diligent in taking care of the paperwork.

My research was supported by NSF grants No. CCF-0811707 and CCF-1018517, Intel, and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author, and do not necessarily reflect the views of the National Science Foundation.

Finally, I would like to thank my wife, Manisha who has been very patient, caring and compassionate during my grad life. Her happy-go-lucky personality, cheerful nature and lively smile made the strenuous grad life so effortless. I cannot imagine my life without her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Industry has fully embraced multi-core chip design for platforms ranging from mobile phones to desktops to supercomputers. However, as we enter the "dark silicon" era [8][24][30][32][78], the number of cores that can be active is limited by the chip's power budget and is less than the number of cores that can be fit on the chip. Specialization is one way to profit from dark silicon. While adding more of the same core design is of little use in this context, introducing differently-designed cores enables run-time adaptation: migrating a thread among a dedicated ensemble of different core types as the thread's instruction-level behavior changes. Matching varying instruction-level behavior to cores optimized for different behaviors, can lead to faster and more energy-efficient execution. A multi-core architecture with multiple core types, which are functionally-equivalent but microarchitecturally-diverse, is called a single-ISA heterogeneous chip multiprocessor (HCMP).

Early HCMP designs were limited to *monotonic core types*: they could be clearly ranked from high-performance/high-power to low-performance/low-power independent of the application, providing an unambiguous performance and power spectrum [46][47][76]. A more powerful class of HCMP employs *non-monotonic core types* [48]: each core type is performance-optimized to different instruction-level behavior and hence cannot be ranked – different applications achieve their highest performance on different cores [13][55][56]. Although non-monotonic heterogeneous designs offer higher performance potential than either monotonic heterogeneous designs or homogeneous designs, steering applications to the

best-performing core is challenging due to performance ambiguity of core types. Moreover, with non-monotonic cores, accurate steering is mandatory. Without it, performance can actually worsen since cores are tuned. Steering completes what was started by architecting non-monotonic cores in the first place.

In this dissertation, we demonstrate that *selecting* non-monotonic core types at design-time and *steering* programs to cores at run-time, are actually "two sides of the same coin". We performed an exhaustive RTL-based [13] design space exploration of HCMPs with one through four superscalar core types, that achieve the highest performance on 39 phases from SPEC. The addition of more core types yields more performance due to tailoring cores to diverse instruction-level behaviors (control-flow, data-flow, cache misses, etc.). With only one core type, the best core in the design space is a rather "average" core that strikes a reasonable balance between instruction-level parallelism (ILP) and frequency. If more core types are allowed, it turns out that the average core is *still* selected and additional core types are selected that relieve distinct resource bottlenecks in the average core, that constrain the performance of "outlier" program phases: a core type with larger window size (window bottleneck); one with higher issue width (width bottleneck); one with lower issue width (low ILP favors maximizing frequency); and so forth. In summary, with N core types, the optimal HCMP for single-thread performance is comprised of an "average core" type coupled with N-1 "accelerator core" types that relieve distinct resource bottlenecks in the average core.

The fact that the selection of non-monotonic cores is empirically driven by average resource provisioning (average core) coupled with distinct resource bottleneck relief (accelerator cores), inspires a complementary steering algorithm. As each accelerator core

provides distinct bottleneck relief, each accelerator core can be ascribed a unique bottleneck signature. The running program is continuously monitored for the distinctive bottlenecks. If the program's current bottleneck signature does not match with the current core's bottleneck signature, then the application is migrated to the accelerator core whose signature matches with that of the application. In case there is no accelerator core whose signature matches with the application's bottleneck signature, then the application is migrated to the average core. Please note that the bottleneck signature performs both the work of when to steer and where (which core) to steer. In other words, there is no need for an explicit phase detection mechanism to figure out when to steer: a new phase of interest is implicit in the application's and core's bottleneck signatures diverging.

In our evaluation, we show that the best 4-core-type HCMP improves single-thread performance up to 76% and 15% on average, over the best homogeneous chip multiprocessor, and our steering algorithm is able to capture most of this performance gain. We also show that our steering algorithm on the 4-core-type HCMP is 33% more energy-efficient, on average, than the homogeneous chip multiprocessor.

Another key challenge in designing a ground-up non-monotonic HCMP is that it takes a lot of time to perform core selection. This is because it requires exhaustive simulation of each application phase on each design point in the core design space. To give an idea of the simulation effort required, we had to perform (13,966)x(39) 10-million-instruction simulations for performing core selection. This is more than 500,000 10-million-instruction simulations, which is a humongous simulation effort.

To address this, researchers have used subsetting-based core selection approaches [50][56]. It basically consists of two steps. First, accelerated design space exploration is performed to find the best core for each application phase in a representative benchmark suite. Then, a core subsetting algorithm (K-means clustering [50] or configurational workload characterization [56]) is used to find the N core types which can fit in the chip from the best core information derived in the first step. Please note that, the first step of design space exploration - finding the best core for each phase - takes almost all of the time.

To accelerate the design space exploration to find the best core, researchers have used classical search/optimization techniques such as random walk, simulated annealing, genetic algorithms, evolution strategy, etc. [18][25][56] [38][49][61]. These techniques aim to reduce the number of design points that need to be searched and use the performance values at the previously simulated design points to guide the search. Further, these techniques have a global view of the design space and avoid getting stuck in a local maximum. Even though these search techniques are much better than an exhaustive design space exploration, it still takes a considerable amount of time to perform design space exploration [25][56][38]. The main reason for this is that these techniques do not leverage any insight as to how the different design parameters of a computer system interact to increase or degrade performance at a given design point and treat the computer system as a "black-box" [42].

To gain insight about the core, we propose using bottleneck analysis (criticality analysis) to accelerate the classical search/optimization techniques.

Section 1.1 explains how we use bottleneck information to accelerate the design space exploration. Section 1.2 discusses the contributions of this thesis. Section 1.3 describes the organization of the thesis.

1.1 Bottleneck-driven Design Space Exploration

We perform criticality analysis to find the critical design parameter(s) (the design parameter(s) which is most detrimental to the performance) along with the simulation of the design point. Subsequently, we use this knowledge within the classical search/optimization techniques to find the next design point(s) to be explored. Criticality analysis helps in finding the critical design parameter(s) at the current design point without performing simulations at the neighboring points. That is, by just performing the criticality analysis at the current design point, we obtain a localized view of the region around the design point. Hence, criticality analysis helps classical search/optimization techniques by avoiding unnecessary simulations of design points to understand the local contour of the region.

If standalone criticality analysis (without a classical search/optimization technique) is used for the design space exploration, then the performance bottlenecks will be successively eliminated at every iteration until a design point is reached which has no bottlenecks. This balanced design point is the best design point in the localized region (local maximum). However, this point may not be a good design point in the global design space. Hence, to give criticality analysis a global view, it is used in conjunction with the classical search/optimization technique which has a global view of the design space and avoids getting stuck in a local maximum. Hence, we see that there is a synergistic behavior between the criticality analysis (good locally) and the classical search/optimization techniques (good

globally). Criticality-driven design space exploration exploits this synergistic behavior to accelerate the design space exploration. To summarize, criticality-driven design space exploration uses the localized information from the criticality analysis to guide the globally-aware classical search/optimization techniques.

Figure 1.1(a) shows the performance landscape of a design space where the height represents the performance of the design point. The optimal design point is the highest point in the landscape. We see that standalone criticality analysis gets stuck in the local maximum. On the other hand, the globally-aware classical search technique reaches the global maximum in a large number of iterations. However, using criticality information to guide the classical search technique helps to reach the global maximum quickly.

Figure 1.1(b) illustrates the framework of criticality-driven design space exploration which uses the criticality information in addition to the performance value of past design point(s) to find the next design point(s) to be explored.

While criticality-driven design space exploration is sufficiently general as to be adaptable for the design space exploration of any computer system, in this dissertation, we demonstrate an entire framework for fast, automated design space exploration of an out-of-order superscalar processor. To enable this, we need a high-fidelity superscalar design space and a detailed criticality model, which models all the components of the superscalar processor.

Our high-fidelity superscalar design space encompasses structure sizes (reorder buffer, issue queue, load queue, store queue, instruction cache, data cache, etc.), widths of pipeline stages, and clock period. To take clock period into account, we need to measure the

propagation delays of different pipeline stages in the superscalar processor. This requires physical implementation of a superscalar processor in a given technology. Therefore, we use the delay data of canonical pipeline stages of the superscalar processor from the FabScalar toolset [14] to build the design space. The high-fidelity design space unveils the delicate interplay between the instruction-level parallelism (ILP) extracting structures and the clock period [7][31].

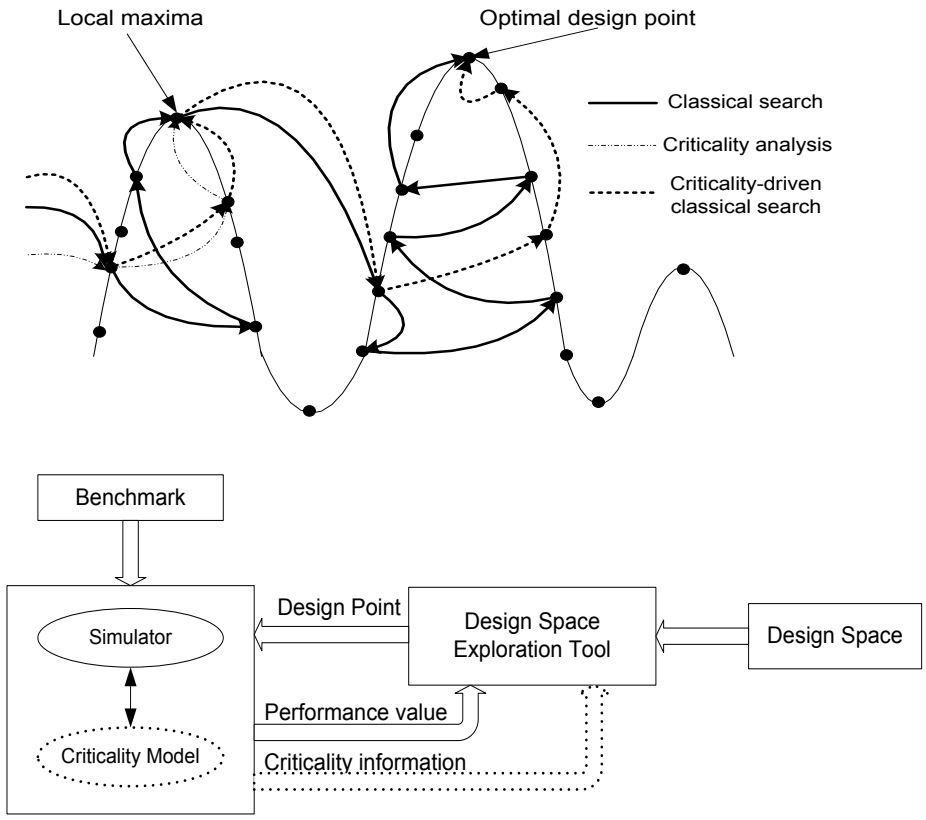

Figure 1.1: (a) (top) The synergistic behavior of criticality analysis and classical search/optimization techniques. (b) (bottom) Block diagram of criticality-driven design space exploration.

In addition to a high-fidelity superscalar design space, we have built a detailed criticality

analysis model which is able to model all the components of an out-of-order superscalar processor. It should be noted that in the issue queue, instructions enter in-order but leave in an out-of-order fashion as the data dependences get resolved and this makes modeling the issue queue more challenging than other resources such as the reorder buffer, load queue and store queue. Because of this complex out-of-order nature of the issue queue, it was not modeled in previous criticality analysis models [26][74] even though it is one of the most critical components of a superscalar processor [26][42]. We have successfully incorporated it into our model. Further, we have extended Field's 3-node dispatch, execute and commit criticality model [26] to a 7-node criticality model in which there is a node corresponding to each of the canonical pipeline stages of a superscalar pipeline, to make our analysis more fine-grained.

For our evaluation, we use simulated annealing [49] and a less sophisticated technique, random walk, as the baseline search/optimization techniques. Note that simulated annealing is one of the state-of-the-art classical search/optimization techniques and has a proof for theoretical convergence [34]. For the design-space exploration of superscalar processors on SPEC 2000 benchmarks [73], on average (harmonic mean), criticality-driven walk achieves 3.8x speedup over random walk and criticality-driven simulated annealing achieves 2.3x speedup over simulated annealing. Another key finding is that even though simulated annealing runs faster than random walk, criticality-driven walk performs better or comparable to simulated annealing on all SPEC 2000 benchmarks. This shows that using criticality analysis over a simple classical search/optimization technique is more effective than only using a sophisticated classical search/optimization technique.

1.2 Contributions

A Unified View of Core Selection and Application Steering

- *Average core / accelerator core phenomenon*: We are the first to empirically discover the average core/accelerator cores phenomenon of HCMPs architected for single-thread performance. This discovery was critical to coming upon the idea of using very simple distinctive bottleneck analysis for steering – a novel and practical idea that is only obvious in hindsight and which is theoretically supported only by the average/accelerator insight.
- *Bottleneck-based steering approach:* We propose a simple and intuitive, bottleneck-based steering algorithm for HCMPs. In the bottleneck-based steering algorithm, there is no need for an explicit phase detection mechanism because a new phase is implicit in the application and core signatures diverging. Further, it captures most of the performance gain in an HCMP and outperforms the current state-of-art steering algorithm (sampling algorithm).
- *Robustness of bottleneck-based steering approach*: We show that the average/accelerator phenomenon holds for different single-thread power budgets – the power allowed for a single thread switching among a dedicated ensemble of N core types. In particular, while the constituent cores change for different single-thread power budgets, we empirically observe that the best homogeneous core type for this power budget always shows up as the average core type in the HCMP and the other

core types relieve distinct bottlenecks in the average core type. Thus, our steering approach is robust from less-constrained to more-constrained designs.

Bottleneck-driven Design Space Exploration

- We propose the use of criticality analysis to drive classical search/optimization techniques to accelerate the design space exploration of a processor core using the synergistic behavior between them.
- We present a detailed implementation of using criticality information to drive simulated annealing and random walk for an out-of-order superscalar processor. Further, we have performed detailed evaluation of criticality-driven simulated annealing and criticality-driven walk.
- We have built a detailed criticality analysis model of a whole out-of-order superscalar processor, including the issue queue. This detailed criticality model can also be used for other purposes like fine-grained bottleneck analysis.

1.3 Organization of the Dissertation

The dissertation is organized as follows. Chapter 2 discusses related work. Chapter 3 presents the unified view of core selection and application steering in HCMPs. Chapter 4 explains how the bottleneck information is used to accelerate the design space exploration. Chapter 5 describes the core selection in HCMPs using subsetting algorithms. Finally, Chapter 6 gives the summary and future work.

CHAPTER 2: RELATED WORK

2.1 HCMP Design

Initial work on single-ISA heterogeneous chip multiprocessors assumed that the cores consisted of different generations of the alpha processor family or frequency-scaled versions of x86 processors [46][47]. These proposals demonstrated improved performance and power efficiency for the heterogeneous architectures.

Subsequently, Kumar et al. [48] demonstrated significant performance benefits for multi-programmed workloads when the HCMP is designed from the ground-up and is not necessarily composed of pre-existing cores. The cores obtained are non-monotonic in performance, i.e., there is no rank ordering of cores in terms of performance. Our work focuses on improving the performance of single-threaded workloads (instead of multi-programmed workloads) in a non-monotonic HCMP for both unconstrained and constrained power budgets.

Azizi et al. [5] studied the energy-performance tradeoffs in processor architecture and performed a marginal-cost analysis. We find the optimal HCMP designs for maximizing single-thread performance for both unconstrained and constrained power budgets. Further, our analysis is based on a detailed high-fidelity RTL model [14].

Dynamic voltage and frequency scaling (DVFS) is orthogonal to heterogeneity. Further, recent research has shown that DVFS is showing diminishing returns on newer platforms [69]. Non-monotonic HCMP provides a compelling alternative.

2.2 Application Steering in HCMP

All the current techniques for application steering can be classified into the following categories: sampling-based approaches, heuristics-based approaches, static approaches and model-based approaches.

2.2.1 Sampling-based Approaches

Sampling-based approaches run the application on every core type after a switching interval to determine the best core type for the application for the next interval.

Kumar et al. [47] proposed the sampling algorithm for steering in a monotonic HCMP for exploiting intra-thread diversity. Becchi et al. [6] proposed a steering algorithm, which relied on the speedup factor (performance improvement on a fast core relative to a slow core) for a monotonic HCMP. The speedup factor was computed by running a thread on each core separately. Winter et al. [80] explored thread scheduling and global power management techniques in HCMP. They compared different algorithms like brute force, greedy and local search for thread scheduling. All examined schemes required sampling to determine the best thread-to-core assignment. Sawalha et al. [67] proposed recording instructions-per-cycle (IPC) of the different program phases in a table and using it for scheduling when the phase reoccurs. However, it still required sampling the performance of the threads on each core type.

Each of the above sampling techniques suffers from the overhead of successively migrating the application on all the core types after every switching interval. Further, this overhead would increase with the number of core types [19]. In our technique, the

application directly migrates from the current core type to the best core type without any need for sampling.

2.2.2 Heuristics-based Approaches

Heuristics-based approaches use heuristics to determine the optimal core mapping in HCMP.

Saez et al. [63] proposed a dynamic algorithm for steering the program phases in a frequency-scaled version of the same processor. They measured the L2 miss rate to find the optimal core mapping. Koufaty et al. [45] used proprietary tools to emulate an asymmetric system where the cores differed in the number of micro-ops that could be retired per cycle. They assumed cores of two types: a big core capable of retiring four micro-ops per cycle and a small core capable of retiring a single micro-op per cycle. Still, these cores are monotonic in nature and do not exploit the full performance advantage of HCMP design. They correlated the application behavior with off-chip and on-chip stalls. They scheduled the applications suffering from memory stalls and other resource stalls on the smaller core and the rest on the bigger core. Patsilaras et al. [59] dynamically scheduled threads based on the amount of MLP which is estimated by the number of L2 misses.

It has been shown that the above approaches may cause suboptimal scheduling as memory intensity alone is not enough for determining the optimal workload-to-core mapping [19]. Further, these techniques have been used in monotonic HCMP and it is not clear how it can be extended to non-monotonic HCMP.

2.2.3 Static Approaches

Static approaches use offline techniques to determine the optimal core mapping for the code fragment.

Chen and John [12] proposed a scheduling algorithm that matches cores to the programs. The algorithm studied the program characteristics and the hardware configurations of each core. Then the scheduler matched the programs to cores depending on the program characteristics, resource demands, and the physical characteristics of cores. This algorithm looked only at the inter-program diversity and does not adapt to the changing workload resulting from phase changes in the programs. Shelepov et al. [68] proposed a static algorithm for steering program phases in a frequency-scaled version of the same processor. They embedded reuse distance profile signatures into the binary, which enable the core to quickly estimate the L2 cache miss rate. Using the L2 cache miss rate, they were able to find the optimal core mapping. Sondag et al. [72] developed an automated static analysis tool, which matches code fragments to the optimal core type.

These static approaches assume that the underlying machine configuration is known. Further, the workload behavior may be drastically impacted by the input data [19]. On the other hand, our technique is a dynamic technique.

2.2.4 Model-based Approaches

Model-based approaches use a dynamic model to determine the best core for the application.

Craeynest et al. [19] collected MLP and ILP information to predict the performance on the other core in a monotonic HCMP. Dubach et al. [22] used machine learning to

dynamically predict the best hardware configuration for a program phase in a reconfigurable processor.

It is unclear how these techniques can be adapted to a non-monotonic HCMP. Our technique uses a bottleneck model to predict the optimal workload-to-core mapping in a non-monotonic HCMP.

2.3 Design Space Exploration

Accelerating design space exploration is a well-studied topic. There are two orthogonal approaches for accelerating design space exploration: speeding up the simulation of a single design point and reducing the number of points to be searched.

For accelerating the simulation of a single design point, there are again two methods: analytical methods and sampling methods. In analytical methods, an analytical expression is obtained for the performance. Karkhanis' analytical model [40] is a direct approach for obtaining the analytical expression. It expresses IPC in terms of steady state IPC and penalties due to branch mispredictions and cache misses. A number of methods use an indirect approach to obtain an analytical expression by sampling the design space and fitting an analytical model to the data [37][35][23][51]. Lee et al. [51] used a regression model and Ipek et al. [35] used artificial neural networks to get the analytical expression. Analytical methods trade accuracy for speed. The reported average error rates of these methods are between 3% and 7%. However, the worst-case error rate is a lot higher and may yield an erroneous conclusion about the optimal design point. Sampling techniques reduce the number of instructions to be simulated while still being representative of the benchmark [18][58][70][81]. This technique is orthogonal to our approach and in fact we use SimPoints

[70] to accelerate the time taken to simulate a given design point.

An orthogonal approach for accelerating design space exploration is to avoid exhaustive search and concentrates on reducing the number of design points that need to be simulated [38][25][56][18]. Classical search/optimization techniques like simulated annealing, genetic algorithms, etc., belong to this category. Criticality-design space exploration builds on these classical search/optimization techniques to make them even faster.

Yi et al. [82] proposed using Plackett and Burman design to find the most critical parameters in the entire design space. Chow et al. [15] and Cai et al. [11] used principal components analysis and multivariate analysis to identify the most critical parameters in the design space. This can help in pruning the design space by focusing on the most important parameters. On the other hand, our technique finds the most critical parameters at a design point (local level) which is not possible using the above methods. This fine-grained information is needed to find the bottleneck at the current design point, which helps to accelerate the design space exploration.

2.4 Criticality Analysis

Criticality analysis is very useful in improving performance in a highly concurrent system like a computer system. It has been proposed to be used inside the processor in the form of a criticality predictor, which is used to identify critical instructions to improve performance [26][27][74]. It is used for better resource arbitration by giving priority to critical instructions [26][74]. It can also be used for misspeculation reduction by restricting speculation to critical instructions [26].

It can also be used offline for bottleneck analysis in the computer system [27][64][65]. It

has been successfully used to identify bottlenecks in processor architectures like TRIPS, clustered architectures, etc. [1][54][66].

In a simpler design space, criticality analysis has been directly used for design space exploration [39]. However, in a more complicated superscalar design space, using only criticality analysis leads to a local maximum which is a balanced but sub-optimal design. Our work uses criticality analysis in conjunction with classical search/optimization techniques to avoid getting stuck at a local maximum. Our work opens up another line of research by using criticality analysis in conjunction with classical search techniques for fast design space exploration.

CHAPTER 3: A UNIFIED VIEW OF CORE SELECTION AND APPLICATION STEERING

An HCMP consists of differently designed core-types which may differ in their widths, structure sizes, cache sizes, predictor sizes, clock periods, pipeline depths, etc. We find that in the optimal HCMP design, we have a core-type with an average width, average structure sizes, etc. (average core-type) and many accelerator core-types, each of which alleviates distinct bottlenecks in the average core-type. This gave us the revelation for our steering algorithm -- that it should be based on profiling for these distinct bottlenecks and migration should be performed whenever the application's bottlenecks do not line up with the design of the current core-type.

Section 3.1 details our algorithm for performing core selection to find the optimal HCMP design. Section 3.2 explains the process of application steering in an HCMP. Section 3.3 and Section 3.4 describe experimental methodology and results, respectively.

3.1 Core Selection in HCMP

Selecting the core-types, which make up a good HCMP design is not straightforward because each application phase prefers a different core-type. As we can only have a limited number of core-types in a given HCMP, this results in a compromise. To choose the core-types, which make up the optimal HCMP design, we perform a rigorous design space exploration using genetic algorithms [29] for the SPEC benchmark suite [73].

The core-types thus obtained provide interesting insights. In the optimal 4-core-type HCMP, there is an average core-type, which strikes a right balance between instruction-level

parallelism (ILP) and frequency. Interestingly, this core-type is also the core-type in the best homogeneous CMP design (CMP with one core-type). This core-type strikes a good balance between large structure size and small structure size, large width and small width, and large clock period and small clock period [3]. The other three core-types represent the accelerator core-types. Each of the accelerator core-types relieves a distinct bottleneck in the average core-type, namely, the width bottleneck, the window bottleneck and the ILP bottleneck, respectively.

Section 3.1.1 discusses our methodology for architecting an N-core-type HCMP. Section 3.1.2 describes the optimal HCMP design with two, three and four core-types.

3.1.1 Methodology

3.1.1.1 Core Design Space

The core design space is spanned by the independent design parameters enumerated in Table 3.1. We see that the core design space consists of design points created by varying the clock period and the complexity (structure sizes and pipeline widths) of different pipeline stages of a superscalar processor. We assume the cores have the following canonical pipeline stages: Fetch, Decode, Rename, Dispatch, Issue, Register Read, Execute, Writeback and Retire. Sub-pipelining is performed within the logical boundaries of the canonical pipeline stages. The Dispatch and Retire stages are always 1-deep and 2-deep, respectively. Other stages have a range of depths. The Fetch stage ranges from 2-deep to 5-deep. The Decode, Issue and Rename stages can go from 1-deep to 3-deep. However, the Issue loop depth, which is the number of cycles taken to wake-up and select dependent instructions, can be

either 1 cycle or 2 cycles. The depth of the Register Read stage varies from 1-deep to 4-deep. The depth of the Writeback stage matches the Register Read stage. Further, we assume three types of functional units: a simple Arithmetic Logic Unit (ALU), which executes integer and branch instructions; a complex ALU, which executes floating-point instructions and integer multiply/divide instructions, and a memory unit, which executes load and store instructions.

Table 3.1: Microarchitectural design parameters.

Parameter	Value Range	Number
Front-end width	2, 3, 4, 5, 6, 7, 8	**7**
Issue width	2, 3, 4, 5, 6, 7, 8	**7**
Physical Register File size	64, 128, 192, 256, 384, 512	**6**
Issue Queue size	16, 24, 32, 48, 64, 96, 128	**7**
Load Queue/Store Queue sizes	8/8, 16/16, 24/24, 32/32, 40/40, 48/48, 56/56, 64/64	**8**
L1 Instruction Cache size	8KB, 16KB, 32KB, 64KB, 128KB	**5**
L1 Data Cache size	8KB, 16KB, 32KB, 64KB, 128KB	**5**
L2 Cache size	2MB	**1**
Clock period	0.5→1.2 ns (granularity=0.1ns)	**8**

The depth of a canonical pipeline stage for a given design point is the degree of sub-pipelining required so that a given complexity of the pipeline stage may fit in a given clock period. A cartesian product of all parameter values gives *3.3 million design points*. Not all of these design points are valid. Firstly, as we are using the FabScalar [13] canonical template, a pipeline stage cannot be made arbitrarily deep. Secondly, for a canonical pipeline stage, only those design points are considered that have the largest-sized structure for a given width, frequency and depth of that canonical pipeline stage. This helps in pruning the design space and retaining only the good design points. By performing this design space pruning, we are able to restrict it to *13,966 design points*.

We further assume private L1 instruction and data caches, and a shared L2 cache. The cache block size is fixed at 64 bytes. The associativity of the L1 caches is 4 and that of the L2 cache is 8. Further, the latency of the caches is computed from Cacti [71].

Instructions-per-cycle (IPC) is obtained using FabScalar's [13] cycle-accurate C++ simulator, which has been validated against FabScalar's register-transfer-level (RTL) designs.

To get a reliable estimate of the clock period, we make use of FabScalar's Performance-Power-Area (PPA) tool [13]. The FabScalar PPA tool provides propagation delays for each and every pipeline stage design in the entire Canonical Pipeline Stage Library (CPSL), and these delays are from logic synthesis and FabMem (for highly ported RAMs/CAMs). The CPSL allows composing thousands of cores and their cycle times are known from the detailed, comprehensive CPSL characterization just described.

For power modeling, we again use FabScalar's cycle-accurate C++ simulator, which like the Wattch method [9], combines per-unit energy numbers with their activity counts to determine the instantaneous power in each cycle. Unlike Wattch, the per-unit dynamic/static energy numbers come from the FabScalar PPA tool, which provides per-unit energy numbers from logic synthesis (for logic) and FabMem (for highly ported RAMs/CAMs), for hundreds of fine-grain components in the core. Moreover, per-unit energies for every superscalar width, depth, and structure sizes are fully represented in the FabScalar PPA tool.

3.1.1.2 Core Selection Algorithm

An HCMP consists of N core-types selected from the core design space. Hence, the HCMP design space consists of every combination of N core-types selected from the core design space. Clearly, finding the optimal HCMP design is a big search endeavor. To give you an idea of the size of the HCMP design space, let us consider the 4-core-type HCMP. The design space of the 4-core-type HCMP will have every 4-core combination selected from 13,966 design points in the core design space. This amounts to $1.59*10^{15}$ HCMP design points. This is a huge design space and it is impossible to exhaustively search this design space. Hence, we resort to genetic algorithm [29] to find the best N-core-type HCMP design.

Table 3.2 shows the SimPoint [70] phases obtained from the SPEC 2000 benchmark suite [73]. These 39 SimPoint phases are exhaustively run on each core design point. For the genetic algorithm, an individual is a given N-core-type HCMP design selected from the HCMP design space and the objective function is the harmonic mean value of the performance values (billions-of-instructions-per-second (BIPS)) of 39 phases (assuming each phase is steered to the best core-type out of the N core-types). We use the standard mutation, crossover and selection operators in the genetic algorithm. We take the population size of 100 and we keep running until the best HCMP design does not change for 10,000 generations.

Table 3.2: SPEC CPU SimPoints.

Benchmark	*Interval Number*
bzip	341, 1873, 3089, 9277
crafty	234, 5647, 12166, 21513
gap	3229, 15783, 16913, 21010
gcc	264, 473, 628, 873
gzip	779, 1361, 5030
mcf	2018, 3091
parser	5201, 10758, 19972, 24856
perl	18, 415, 781, 1309
twolf	2482, 8155, 15968, 27755
vortex	5877, 7432, 9025
vpr	1350, 3463, 6120

We use this methodology to find the core configurations of 2-core-type, 3-core-type and 4-core-type HCMPs. The next section discusses the core configuration obtained from this methodology.

3.1.2 HCMP Design

First, we discuss the optimal unconstrained HCMP design in Section 3.1.2.1. Then, the optimal HCMP designs with different power budgets are described in Section 3.1.2.2.

3.1.2.1 Unconstrained HCMP Design

The core-types in the optimal 2-core-type, 3-core-type and 4-core-type HCMPs are given in Table 3.4. The microarchitecture configurations of these core-types are shown in Table 3.3.

Table 3.3: Microarchitecture configurations of different core-types in the unconstrained HCMP design.

Core-type	*Clock Period (ns)*	*ILP-extracting buffers (IQ, LSQ, ROB)*	*Widths (fetch, issue)*	*Caches (KB) (I, D)*
A	0.6	32, 128, 128	3, 4	64, 64
N	0.5	32, 64, 64	2, 2	16, 16
L	0.7	48, 128, 384	4, 4	128, 128
W	0.7	32, 128, 128	6, 6	128, 32
LW	0.7	48, 128, 192	4, 5	128, 32

Table 3.4: Core-types in the optimal 2-core-type, 3-core-type and 4-core-type HCMPs.

Number of core-types in HCMP	**Core combinations**
2	A, LW
3	A, LW, N
4	A, L, N, W

We see that, irrespective of the number of core-types, the A core-type - which is also the best homogeneous core design (called the "average" core-type) - is always present in the optimal heterogeneous design. We further see that the other core types in the optimal HCMP (called accelerator core-types) aim to remove distinct bottlenecks. In the optimal 2-core-type HCMP, in addition to the average core-type, we have the core-type LW (which has both a larger window and a bigger issue width) that targets application phases with a window bottleneck or a width bottleneck. In addition to the average core-type and the core-type LW,

the optimal 3-core-type design consists of the core-type N (narrow core) that focuses on application phases with a severe ILP bottleneck. In the optimal 4-core-type HCMP, in addition to the average core-type, we have the core-type N (narrow core) which concentrates on application phases with a severe ILP bottleneck, the core-type L (large window core) which targets application phases with a window (reorder buffer, issue queue) bottleneck and the core-type W (wide core) which focuses on application phases with a width (fetch width, issue width) bottleneck.

In other words, in the optimal 4-core-type HCMP, the core-type L targets application phases with distant ILP, the core-type W concentrates on application phases with nearby ILP, the core-type N focuses on application phases with low ILP, and the average core-type targets the rest of the application phases.

In summary, the optimal HCMP consists of (1) an average core-type that is comparable to a commercial out-of-order processor (4-issue, 128 in-flight instructions, etc.), (2) accelerator core-types that match distinct and intuitive ILP behaviors. The next section uncovers if this average core/accelerator core phenomenon is also applicable for HCMP designs with a given power budget.

3.1.2.2 Power-constrained HCMP design

The peak power dissipation fundamentally drives processor design [44]. Thermal budgeting of the processor, power supply cost, cooling cost and packaging cost all depend on the peak power dissipation of the processor.

In our case, the peak power of a HCMP design is the peak power of the core-type (out of the N core-types in the HCMP design) that consumes the most peak power. This is because,

at any given time, only one core is powered on and all the other cores are fully power-gated. Please note that having a peak power constraint constrains the core design space, as more complex core-types that exceed the peak power constraint will be weeded out. Hence, the HCMP core selection algorithm will have a smaller palette of core-types to choose from.

We find that the peak power of HCMP designs varies from 1.0 W to 5.5 W. Hence, to understand the effect of having a power budget on the optimal HCMP design, we vary the power budget from 1.5W to 5.5W at 1W intervals. The optimal HCMP designs for peak power constraints of 3.5W, 4.5W and 5.5W are found to be the same as that with the unconstrained power budget. This shows that the optimal unconstrained HCMP design is still a good, power-efficient design. The following sub-sections discuss the optimal HCMP designs with power constraints of 2.5W and 1.5W.

2.5W Power Budget

The core-types obtained in the optimal 4-core-type HCMP design with a 2.5W peak power constraint are given in Table 3.5.

Table 3.5: Microarchitecture configurations of different core types in the optimal 4-core-type HCMP design with 2.5W power budget.

Core-type	*Clock Period (ns)*	*ILP-extracting buffers (IQ, LSQ, ROB)*	*Widths (fetch, issue)*	*Caches (KB) (I, D)*
A1	0.6	32, 128, 128	3, 4	64, 64
N1	0.5	32, 64, 64	2, 2	16, 16
W1	0.7	48, 128, 128	4, 5	128, 32
L1	0.8	64, 128, 256	4, 5	32, 64

In Table 3.5, the A1 core-type is the best homogeneous core design (average core-type) for a 2.5W power budget. The core-type W1 has a bigger width and hence concentrates on

application phases with a width bottleneck. The core-type L1 has a bigger window and hence focuses on application phases with a window bottleneck. The core-type N1 has a higher clock frequency and hence targets application phases with an ILP bottleneck.

1.5W Power Budget

At a 1.5W power budget, we find that there is no performance improvement if we have more than 3 core-types. This is because at the 1.5W power budget, the HCMP design is so constrained that we can only have 202 feasible core-types (compared to 13,966 core-types in the original design space). Hence, we find the core types in the optimal 3-core-type HCMP design (instead of 4-core-type HCMP design).

The core-types obtained in the optimal 3-core-type HCMP design with a 1.5W peak-power constraint are given in Table 3.6.

Table 3.6: Microarchitecture configurations of different core-types in the optimal 3-core-type HCMP design with 1.5W power budget.

Core-type	*Clock-period (ns)*	*ILP-extracting buffers (IQ, LSQ, ROB)*	*Widths (fetch, issue)*	*Caches (KB) (I, D)*
A2	0.6	48, 32, 64	3, 2	64, 8
L2	1.0	16, 128, 192	3, 4	128, 128
W2	0.8	16, 64, 64	3, 6	128, 16

In Table 3.6, the A2 core-type is the best homogeneous core design (average core-type) for the 1.5W power budget. The core-type L2 has a bigger window size and hence targets the application phases with a window bottleneck. The core-type W2 has a bigger width and hence targets the application phases with a width bottleneck.

Further, we see that irrespective of the number of core-types, the average core-type always shows up in the optimal design for both 2.5W and 1.5W power budgets. In summary, we see that even with a constrained power budget, the optimal HCMP consists of (1) an average core-type and (2) accelerator core-types that relieve distinct bottlenecks in the average core-type.

3.2 Application Steering in HCMP

In the optimal unconstrained or constrained HCMP, we have the following core-types: 1) an average core-type, which is the best homogeneous core-type, and 2) accelerator core types, which relieve distinct bottlenecks in the average core-type. The accelerator core-types that aim to alleviate distinct bottlenecks in the average core-type, gave us a good starting point to develop our steering algorithm. As each accelerator core-type alleviates distinct bottlenecks, we can associate a unique bottleneck signature with each accelerator core-type. This drives the fundamental principle behind our steering mechanism, which is that an application phase performs best on the core-type whose bottleneck signature matches with the application phase's signature. Our application steering mechanism is explained in more detail in the next paragraph.

Any application steering mechanism in an HCMP should address two things: when to steer and where (which core-type) to steer. Our novel steering mechanism makes use of the bottleneck information to perform both. The running application is continuously monitored using performance counters. Every 10K instructions, a bottleneck signature is created using the performance counter values and the performance counters are reset. If the bottleneck

signature differs from the current core-type's bottleneck signature, then it has to be migrated to another core-type. To find which core-type to migrate to, the bottleneck signature is indexed into the signature table to find the optimal core-type for the new phase. Once the optimal core-type is figured out, the application phase is steered to that core-type. It is interesting to see that there is no need for explicit phase detection. The profiling mechanism implicitly performs the phase detection.

Section 3.2.1 describes our profiling infrastructure used to compute the bottleneck signature. Section 3.2.2 explains how we perform steering using the bottleneck signature.

3.2.1 Profiling

Table 3.7 shows the counters used for profiling in the core. Each of these counters, except the cycle counter and the branch misprediction counter, denotes the cycles stalled due to the corresponding resource. We see that only seven counters are used for profiling. Hence, the hardware cost is minimal. Once the profiling is complete, the counters are normalized with respect to the total cycle count of the profiled segment of execution. If a normalized performance counter is above a certain threshold, then the corresponding resource is considered as a bottleneck. This threshold is determined empirically for each resource and for each core-type. This bottleneck information obtained from the handful of counters is used to create the bottleneck signature.

Table 3.7: Counters used for profiling.

Performance counter	Comments
Branch misprediction recovery counter	Number of cycles the pipeline is stalled due to branch misprediction recovery[1]
L2-cache stall counter	Number of cycles a load is stalled because of a L2 cache miss
I-cache stall counter	Number of cycles instruction fetch is stalled because of an instruction cache miss
D-cache stall counter	Number of cycles a load is stalled because of a data cache miss
Issue-width stall counter	Number of cycles for which an instruction issue is stalled because of lack of issue bandwidth
Window stall counter	Number of cycles for which an instruction dispatch is stalled because of a blocked issue queue, reorder buffer or load-store queue
Cycle counter	Number of cycles taken to execute the profiled segment

Our steering scheme is a dynamic (hardware) scheme that is isolated from the Operating System. The steering is managed autonomously to optimize performance at fine-grain time slices.

The next section discusses how we use the bottleneck information to perform the application steering.

3.2.2 Application Steering using Bottleneck Signature

The key insight behind our steering algorithm is that the migration should be performed whenever the application's bottlenecks do not line up with that of the current core-type.

[1] The branch misprediction penalty has three components: (i) Pipeline drain penalty: This is the performance penalty due to not bringing in correct-path instructions between the time the mispredicted branch is fetched and the time it executes. (ii) Pipeline flush penalty: This is the penalty for flushing the pipeline, recovering the rename map table and correcting the PC after the mispredicted branch is resolved. (iii) Pipeline refill penalty: This is the performance penalty for refilling the pipeline after the misprediction recovery. Measuring the pipeline drain and refill penalties is challenging due to overlapped execution, and at the least is intrusive to the pipeline. Hence, we only account for the pipeline flush penalty for evaluating the branch misprediction bottleneck.

To find the best core quickly after profiling of the application bottlenecks, we use a bottleneck signature table. A bottleneck signature is basically a compact representation of various bottlenecks present in the current core-type. Once the profiling is completed, a bottleneck signature is created from the bottleneck information available from the performance counters. This bottleneck signature is then indexed into the bottleneck signature table, which predicts the best core-type for steering.

To avoid ping-ponging of the application between the core-types, we use a confidence counter [36]. We only migrate to a certain core-type if the steering algorithm predicts migrating to the same core-type a given number of times consecutively. Hence, we have a confidence counter with each core-type. A core-type's counter is incremented when the steering algorithm predicts migrating to that core-type. It is reset when the algorithm predicts a different core-type. If a confidence counter saturates, then the application is migrated to the corresponding core-type. The next section details the building of the bottleneck signature table.

3.2.2.1 Building of Bottleneck Signature Table

Finding the Thresholds

Recall that, if a normalized performance counter is above a certain threshold, then the corresponding resource is considered as a bottleneck. These threshold values are empirically determined by performing a design space exploration (DSE). We use a set of threshold values in the bottleneck signature table as the input to the design space exploration. For training, we select 4 benchmarks having different application characteristics: bzip, mcf, swim and mgrid. The objective function for the DSE is to maximize the performance value

obtained by bottleneck steering using the given bottleneck signature table. Please note that we use a genetic algorithm to perform the DSE. One of the advantages of using thresholds is that the signature table can be easily retuned for other performance metrics like $BIPS^2$/watt, $BIPS^3$/watt, etc.

Constructing the Signature

The signature for each core-type is constructed as follows. We first find the strengths and the weaknesses of each accelerator core-type in comparison with the average core-type. For example, the strength of core-type L (large core) in the optimal unconstrained 4-core-type HCMP is its large window and its weakness is its large clock-period. Then we find the bottlenecks corresponding to its strengths and weaknesses. For example, the bottleneck corresponding to the strength of core-type L is the window bottleneck. Similarly, the bottlenecks corresponding to the weakness of core-type L are the L2 cache bottleneck and branch misprediction bottleneck, because these ILP-degrading factors favor frequency over large window or width.

To generalize, let the bottlenecks corresponding to strengths be S1, S2, S3,... and those corresponding to the weaknesses be W1, W2, W3,… Then, the signature of the core-type is {S1 || S2 || S3...} && !{W1 || W2 || W3...}. In other words, application phases having S1 or S2 or S3 bottlenecks and at the same time not having W1 or W2 or W3 bottlenecks, will prefer this core-type. In the next section, we intuitively build the bottleneck signature table for the unconstrained HCMP (Section 3.2.2.2) and constrained HCMPs (Section 3.2.2.3).

3.2.2.2 Unconstrained HCMP

Let us consider a 4-core-type HCMP. In the optimal unconstrained 4-core-type HCMP, we have the L core-type, the W core-type, the N core-type and the A core-type. The L core-type has a bigger window (reorder buffer, issue queue and load/store queues) than the other core-types. Hence, application phases having a window bottleneck will prefer the L core-type. In addition, if the application phase has to take advantage of a large window, it should not have branch misprediction or L2 cache bottlenecks. Similarly, the W core-type has a larger width than all the other core-types and, hence, application phases having a width bottleneck, no branch misprediction bottleneck, and no L2 cache bottleneck, will prefer the W core-type. The N core-type has a smaller width and a smaller window. However, it has a higher clock frequency. Hence, application phases having low ILP will prefer the N core-type. That is, application phases having branch misprediction or L2 cache bottlenecks will prefer the N core-type. The above discussion is summarized in Table 3.8(a). If the signature of the application phase does not match any of the above signatures, then it is steered to the A core-type.

Table 3.8(a): Bottleneck signature table for the optimal unconstrained 4-core-type HCMP.

Bottleneck Signature	Core-type
Window && !(CTRL || L2)	L
Width && !(CTRL || L2)	W
(CTRL || L2) && !(Width)	N
Rest	A

We can fine-tune the bottleneck signature table by taking into account the I-cache and D-cache behaviors. We see that the L core-type has a bigger D-cache. Hence, application phases having a D-cache bottleneck will prefer the L core-type. Similarly, the W core-type has a bigger I-cache but a smaller D-cache. Hence, application phases having an I-cache bottleneck and no D-cache bottleneck will prefer the W core-type. The updated bottleneck signature table is presented in Table 3.8(b).

Further, according to the algorithm described in the previous section, the signature of the N core-type should be (CTRL || L2) && !(Width || Window). However, if we have a L2 bottleneck, then L2-missed load instructions will definitely clog the window, causing there to be a window bottleneck, too. In other words, there is a window bottleneck whenever there is a L2 bottleneck. This will cause the naively prescribed signature of the N core-type ((… || L2) && !(… || Window)), to not be set when there is a L2 bottleneck. Hence, the signature of the N core-type is changed to (CTRL || L2) && !(Width). Similarly, when there is a L2 cache bottleneck, there will be an I-cache and a D-cache bottleneck, too. And the signature is changed accordingly in that case. Please note that this is the only exception to the algorithm stated before.

Table 3.8(b): Updated Bottleneck Signature Table for the optimal unconstrained 4-core-type HCMP.

Bottleneck Signature	Core-type
(Window || D$) && !(CTRL || L2)	L
(Width || I$) && !(CTRL \|\| L2 \|\| D$)	W
(CTRL || L2) && !(Width)	N
Rest	A

Validation of the Bottleneck Signature Table

To validate the bottleneck signature table, we use the benchmarks in the SPEC benchmark suite. We ran the first 4 billion instructions of each benchmark in the optimal unconstrained 4-core-type HCMP assuming oracle steering. By oracle steering, we mean that the benchmark always runs on the best core-type for a given interval. Further, note that the steering occurs at the granularity of 10K instructions.

For every 10K instruction interval, we extract the performance counter values and find the bottlenecks. In our framework, we model six bottlenecks, namely Branch misprediction bottleneck (CTRL), L2 cache bottleneck (L2), I-cache bottleneck (I$), D-cache bottleneck (D$), Window bottleneck and Width bottleneck. Therefore, there can be 2^6 possible combinations of bottlenecks. For each bottleneck combination, we find the core-type that is selected for the majority of 10K intervals with that bottleneck combination. Further, for accelerator core-types, we only consider those 10K instruction intervals, which give more than 1% performance improvement over the A core-type.

Table 3.9 shows the preferred core-types for different bottleneck combinations. Further, Table 3.10 enumerates the different bottleneck combinations exhibited by each core-type in the bottleneck signature table (Table 3.8).

We see that the set of bottleneck combinations for each core-type in Table 3.9 is a subset of the set of bottleneck combinations for the same core-type in Table 3.10. In other words, the bottleneck signature table shown in Table 3.8 captures the entire bottleneck combination table in Table 3.9. Hence, this experiment validates our bottleneck signature table.

Table 3.9: Core-types preferred by different bottleneck combinations.

Core-type	Bottleneck combination {CTRL, L2, I$, D$, Window, Width}
L	{0, 0, 0, 0, 1, 1} {0, 0, 0, 1, 0, 1} {0, 0, 0, 0, 1, 0}
W	{0, 0, 1, 0, 0, 1} {0, 0, 0, 0, 0, 1}
N	{1, 0, 0, 0, 1, 0} {0, 1, 1, 1, 1, 0}
A	{0, 0, 0, 0, 0, 0} {1, 0, 0, 0, 1, 1} {1, 0, 0, 0, 0, 1}

Table 3.10: Bottleneck combinations exhibited by the bottleneck signature table.

Core-type	Bottleneck combination {CTRL, L2, I$, D$, Window, Width}
L	{0, 0, X, 0, 1, X} {0, 0, X, 1, 0, X} {0, 0, X, 1, 1, X}
W	{0, 0, 0, 0, 0, 1} {0, 0, 1, 0, 0, 0} {0, 0, 1, 0, 0, 1}
N	{0, 1, X, X, X, 0} {1, 0, X, X, X, 0} {1, 1, X, X, X, 0}
A	{0, 0, 0, 0, 0, 0} {0, 1, X, X, X, 1} {1, 0, X, X, X, 1} {1, 1, X, X, X, 1}

3.2.2.3 Power-constrained HCMP

Under a 2.5W power budget, we have the L1, W1, N1, and A1 core-types. Since the L1 core-type has a bigger window and a smaller I-cache, it targets application phases with a window bottleneck, no branch misprediction bottleneck, no L2 bottleneck and no I-cache bottleneck. Further, the W1 core-type has a bigger width, a bigger I-cache and a smaller D-

cache, therefore, it concentrates on application phases with a width bottleneck or an I-cache bottleneck. At the same time, the application phases should not have a branch misprediction bottleneck or a L2 bottleneck or a D-cache bottleneck. The N1 core-type has a higher frequency and, hence, it focuses on application phases with an ILP bottleneck. That is, it targets application phases with a branch misprediction bottleneck or a L2 bottleneck and no width bottleneck. This is shown in Table 3.11.

Table 3.11: Bottleneck signature table for the optimal 4-core-type HCMP under a 2.5W power budget.

Bottleneck Signature	Core-type
(Window) && !(CTRL || L2 || I$)	L1
(Width || I$) && !(CTRL \|\| L2 \|\| D$)	W1
(CTRL || L2) && !(Width)	N1
Rest	A1

Under a 1.5W power budget, we have the L2, W2, and A2 core-types. Similar to the optimal 2.5W HCMP, we obtain the bottleneck signature table for the optimal 3-core-type HCMP at 1.5W, which is shown in Table 3.12.

Table 3.12: Bottleneck signature table for the optimal 3-core-type HCMP under a 1.5W power budget.

Bottleneck Signature	Core-type
(Window || D$) && !(CTRL || L2)	L2
(Width || I$) && !(CTRL || L2)	W2
Rest	A2

3.3 Experimental Methodology

3.3.1 Benchmarks

For our evaluation, we used the SPEC 2000 benchmark suite with the given reference inputs. We simulate the first four billion instructions of each benchmark. Further, we used MiBench for robustness analysis. Please note that it is important to simulate for a longer time to capture the different phases of the program and to see the effects of thread migration.

3.3.2 Metric

We use billions-of-instructions-per-second (BIPS) to quantify the performance of a benchmark executing on a processor core in a HCMP design. $BIPS^3$/watt is a voltage-independent metric and, hence, $BIPS^3$/watt is considered as a better metric than energy-delay-product [52]. Hence, for quantifying the combined performance and power efficiency of our steering algorithm, we use $BIPS^3$/watt. Please note that watt here refers to the average power and not the peak power.

3.3.3 Evaluation Methodology

To evaluate our technique, we proceed as follows. We simulate each benchmark for 4 billion instructions. Every 10,000 instructions, we gather the performance counter values and construct the bottleneck signature. Based on the bottleneck signature, the steering algorithm decides whether to steer to a different core or not.

The thread migration overhead is assumed to be 100 cycles. This is the overhead due to the copying of register state between the processors. For the fast copying of cache state, we assume working set prediction to perform prefetching, which incurs negligible overhead as reported by Brown, Porter, and Tullsen [10]. Further, when the inbound core is being woken-up, the program is still run on the outbound core and the register state is transferred after the wakeup. Hence, the core-wakeup overhead is hidden. Therefore, we believe that 100 cycles should be enough for thread migration, although even longer latencies will only make our results look better compared to the sampling algorithm which migrates more often; in fact, we wanted to be conservative with our speedups so we used an aggressive migration latency.

Nevertheless, for robustness analysis, we also show results for thread migration overheads of 1K cycles and 10K cycles in Section 3.4.3. Please note that we use a 1-bit confidence counter for 100-cycle migration overhead and a 4-bit confidence counter for 1K-cycle and 10K-cycle migration overheads.

For the baseline, we run the 4 billion instructions of a given benchmark on the average (A) core-type. Please note that all the graphs in the result section are normalized with respect to the average core-type (homogeneous design). We further compare our steering technique with oracle steering to get the upper bound. In oracle steering, for each 10K instruction segment, we run it on the best core-type for that segment.

3.4 Results

3.4.1 Performance

3.4.1.1 Evaluation of Steering Algorithm

In the optimal unconstrained 4-core-type HCMP, the bottleneck-driven steering algorithm performs 12% better than the average core-type for SPEC benchmark suite. We further see that the oracle steering algorithm performs 15% better than the average core-type. This shows that the bottleneck-driven steering algorithm captures most of the performance improvement given by oracle steering.

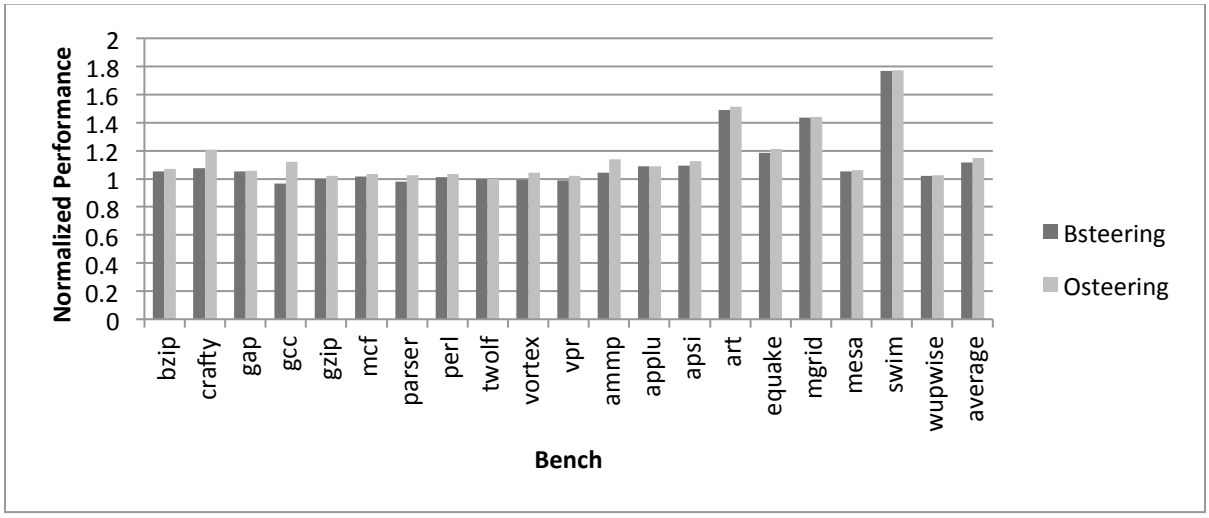

Figure 3.1: Performance of bottleneck-driven steering algorithm (Bsteering) in the unconstrained 4-core-type HCMP for the SPEC benchmark suite. Oracle steering (Osteering) is also shown for comparison.

Figure 3.2 shows the misprediction rate (ratio of incorrect and correct steering predictions) of the bottleneck-driven steering algorithm in the optimal unconstrained 4-core-

type HCMP for the SPEC 2000 benchmark suite. Notice that crafty, gcc, vortex and ammp, which show lower performance compared to oracle steering, have a relatively higher misprediction rate.

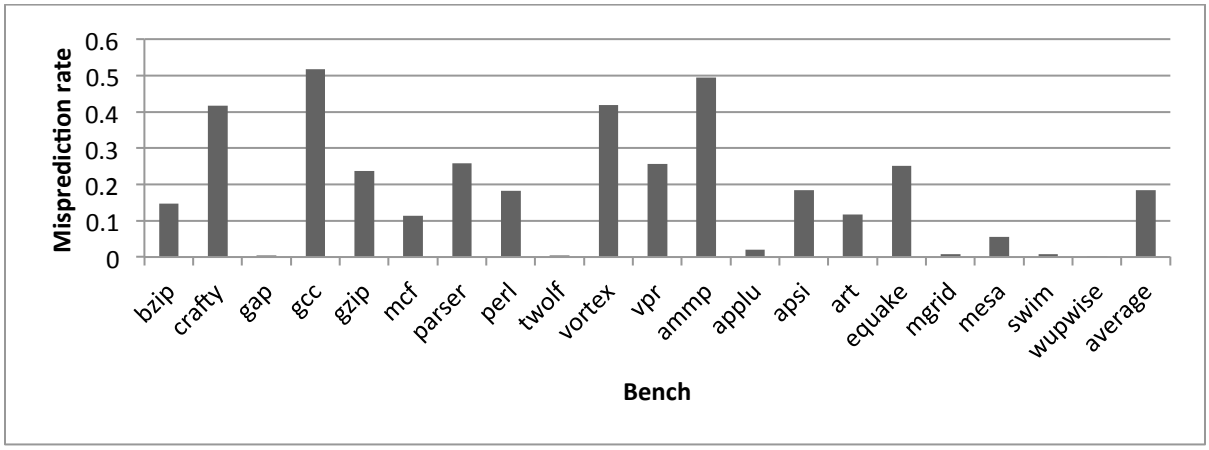

Figure 3.2: Misprediction rate of bottleneck-driven steering algorithm in the unconstrained 4-core-type HCMP for the SPEC benchmark suite.

Figure 3.3 shows the occupancy of different core-types in the unconstrained 4-core-type HCMP for the SPEC benchmark suite assuming oracle steering. It is interesting to see that there are significant differences in the occupancies of different core-types in the 4-core-type HCMP, across benchmarks.

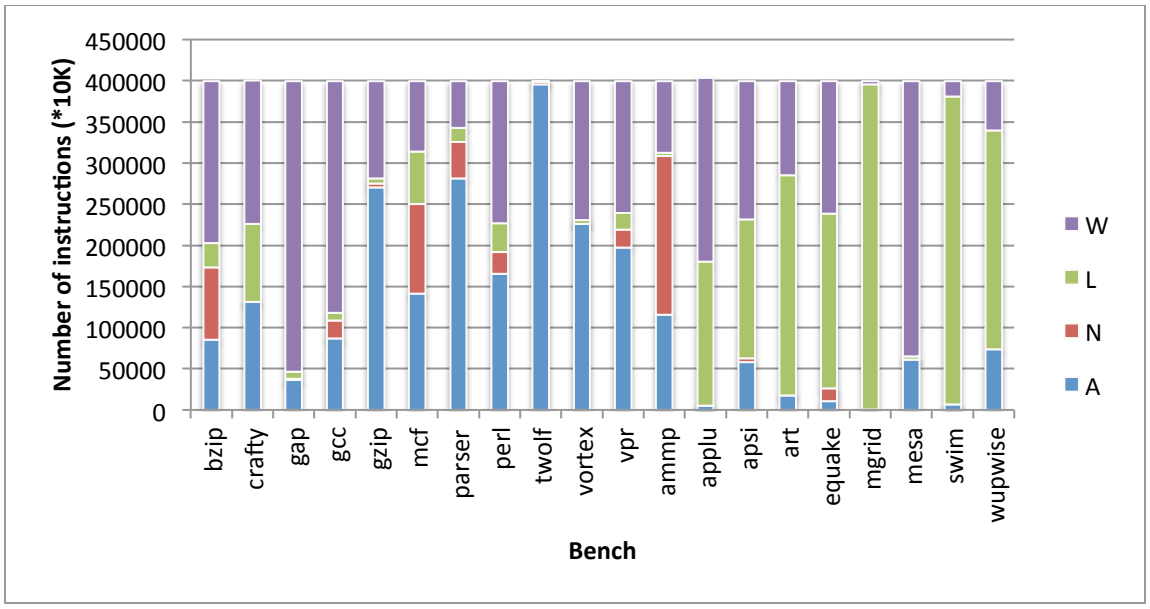

Figure 3.3: Number of instructions spent in each core-type in the unconstrained 4-core-type HCMP for the SPEC benchmark suite (Best viewed in color).

3.4.1.2 Comparison with the Sampling Algorithm

Past research on application steering used sampling algorithms [6]. To model the sampling algorithm, we introduce two parameters: sampling interval and switching interval. In the sampling algorithm, after every switching interval, the application is run on each core-type for the given sampling interval. After sampling, the application is run on the best core-type found during sampling, for the rest of the switching interval. We compare different sampling schemes with different sampling/switching intervals and we found that a sampling interval of 10K instructions and switching interval of 1M instructions works best (Figure 3.4). In Figure 3.5, we see that this sampling scheme performs only 8.9% better than the average core-type. The bottleneck steering algorithm performs 12% better than the average core-type and, hence, outperforms the sampling scheme.

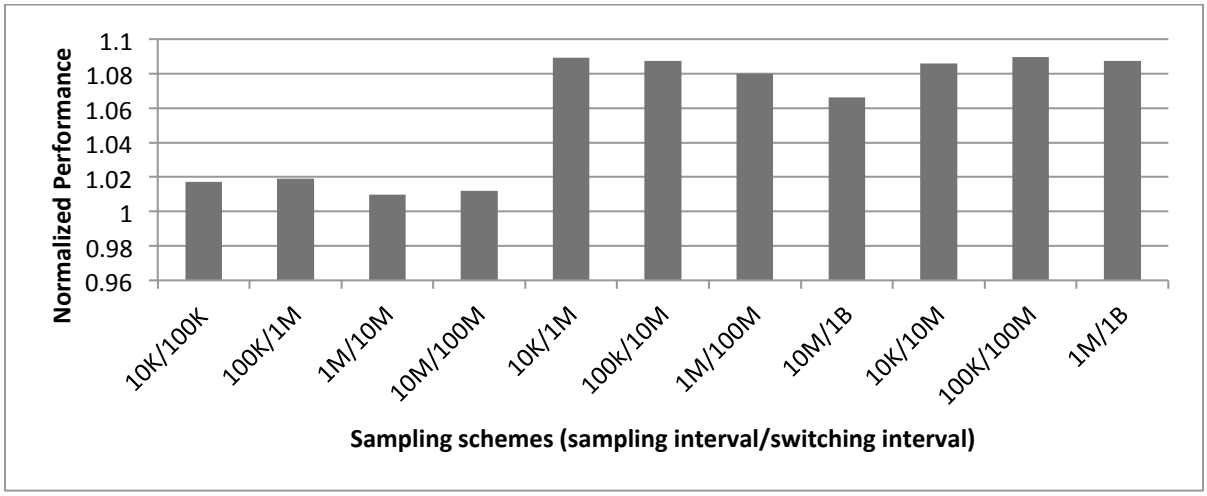

Figure 3.4: Comparison of different sampling schemes.

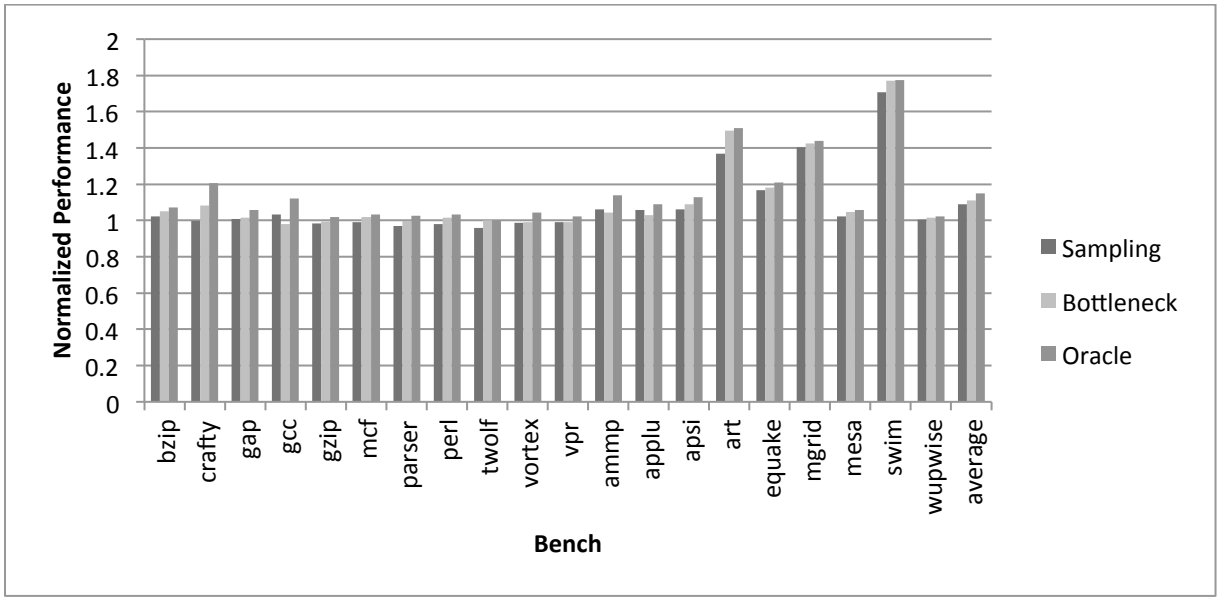

Figure 3.5: Comparison of normalized performance of sampling algorithm with that of bottleneck steering algorithm.

3.4.2 Efficiency

To evaluate the efficiency of our steering algorithm, we compare $BIPS^3$/watt for our steering algorithm with that of the average core-type. We see that our steering algorithm outperforms the average core-type by 33% in terms of efficiency. In contrast, the sampling algorithm only performs 25% better than the average core-type. Full results are shown in Figure 3.6.

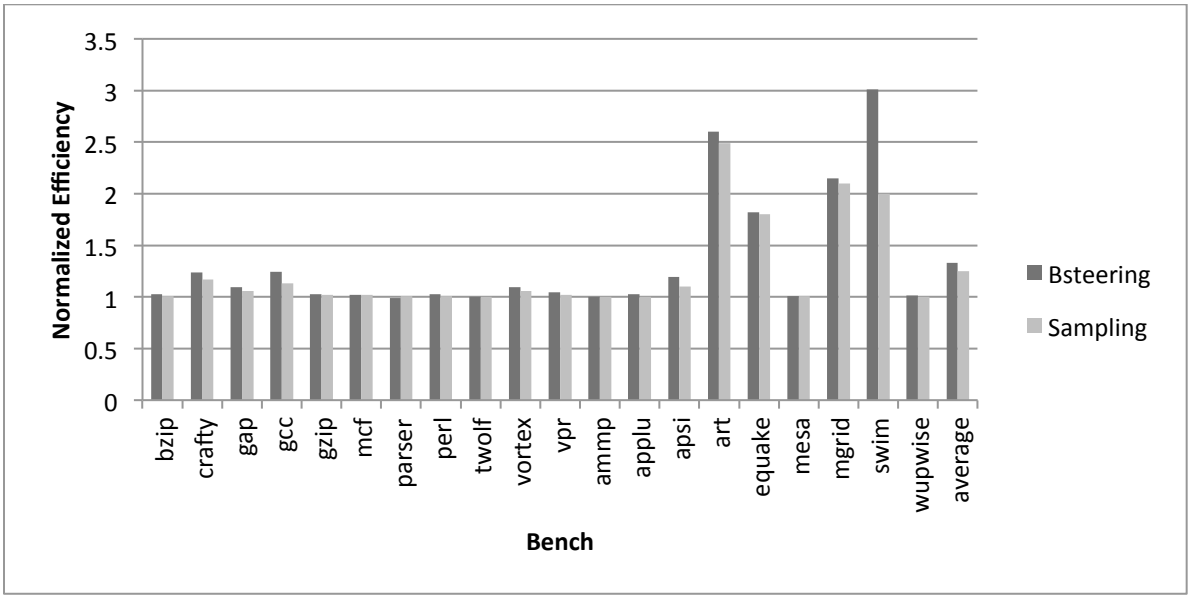

Figure 3.6: Normalized efficiency ($BIPS^3$/watt) of the bottleneck and sampling steering algorithms.

3.4.3 Robustness Analysis

3.4.3.1 Power-constrained HCMP

At the 2.5W power budget, the bottleneck-driven steering algorithm in the 4-core-type HCMP performs 8% better than the average core-type (at 2.5W power budget), as opposed to

the 11.5% performance gain achieved by oracle steering. At the 1.5W power budget, the bottleneck-driven steering algorithm in the 3-core-type HCMP performs 11% better than the average core-type (at 1.5W power budget) as opposed to the 15% performance gain achieved by oracle steering. The 2.5W and 1.5W HCMP results are shown in Figure 3.7 and Figure 3.8, respectively.

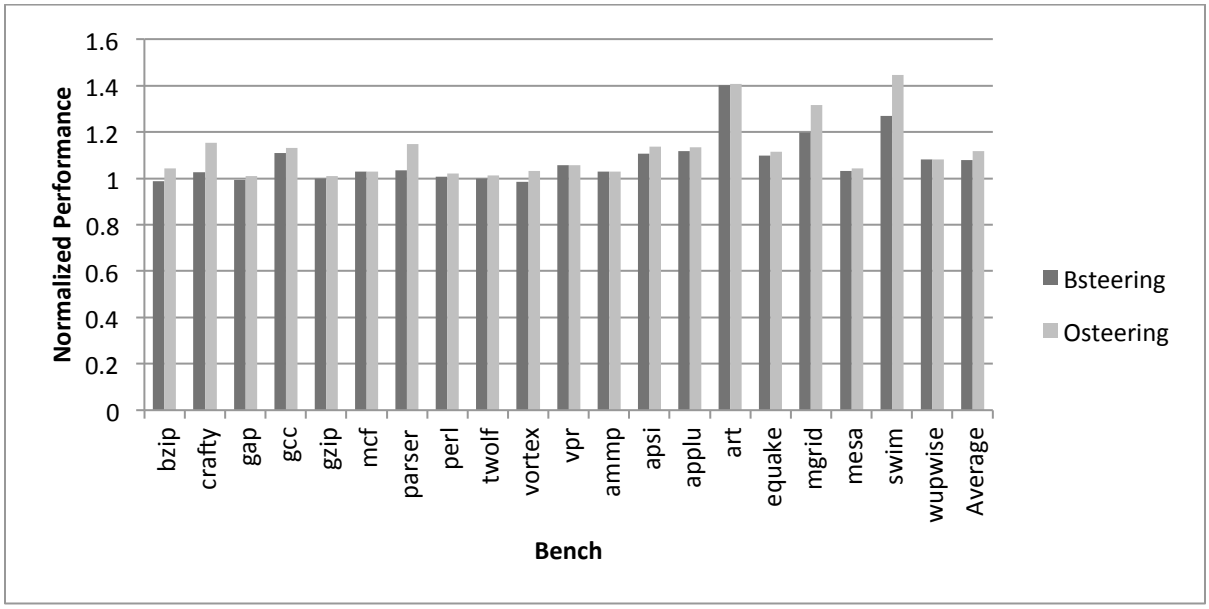

Figure 3.7: Peformance of bottleneck-driven steering algorithm (Bsteering) in the 4-core-type HCMP for 2.5W power budget.

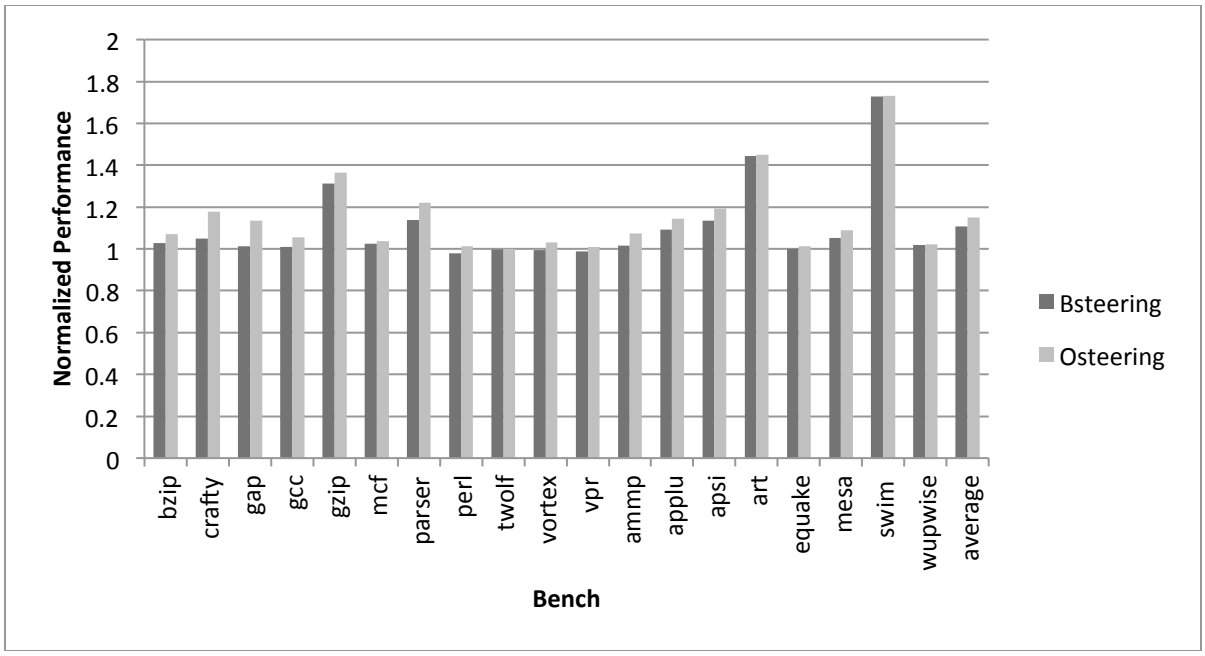

Figure 3.8: Peformance of bottleneck-driven steering algorithm (Bsteering) in the 3-core-type HCMP for 1.5W power budget.

3.4.3.2 Changing the Benchmark Suite

For the MiBench benchmark suite in the unconstrained 4-core-type HCMP, the bottleneck-driven steering algorithm performs 3.5% better than the average core-type. We further see that the oracle steering algorithm performs 6% better than the average core-type. Full results are shown in Figure 3.9.

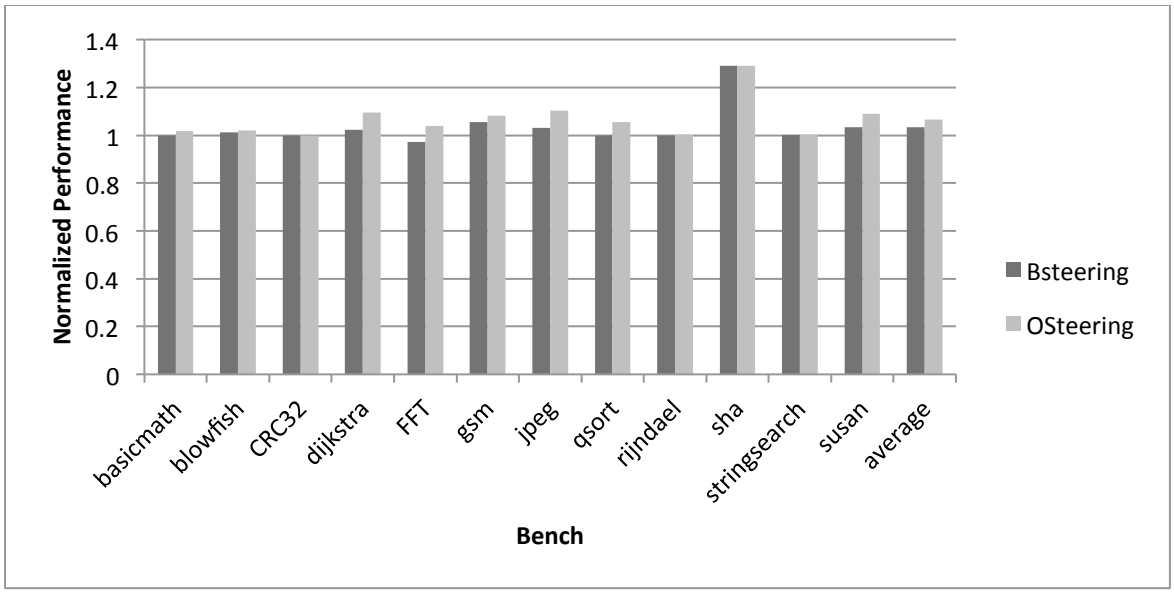

Figure 3.9: Performance of bottleneck-driven steering algorithm (Bsteering) in the unconstrained 4-core-type HCMP for the MiBench benchmark suite.

3.4.3.3 Changing the Number of Core Types

We see that, as we decrease the number of core types to 3, the average performance gain for oracle steering decreases from 15% to 13% (Figure 3.10). Further, when we decrease the number of core types to 2, the performance of the oracle steering reduces to 12% (Figure 3.11). In both cases, our bottleneck-driven steering algorithm is able to capture most of the gain.

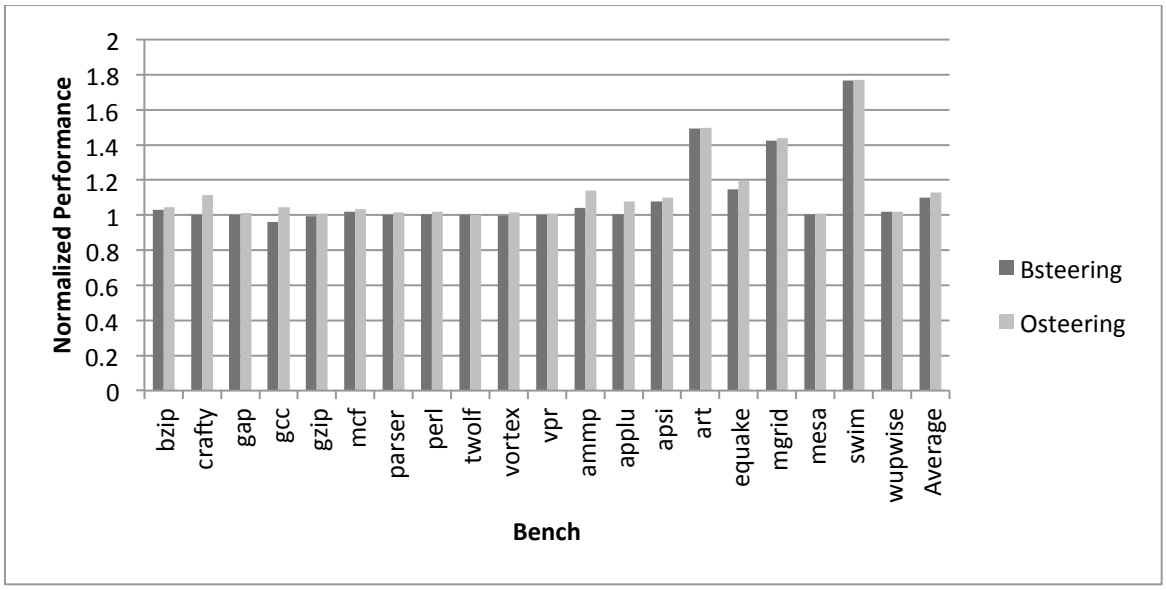

Figure 3.10: Performance of bottleneck-driven steering algorithm (Bsteering) in the unconstrained 3-core-type HCMP.

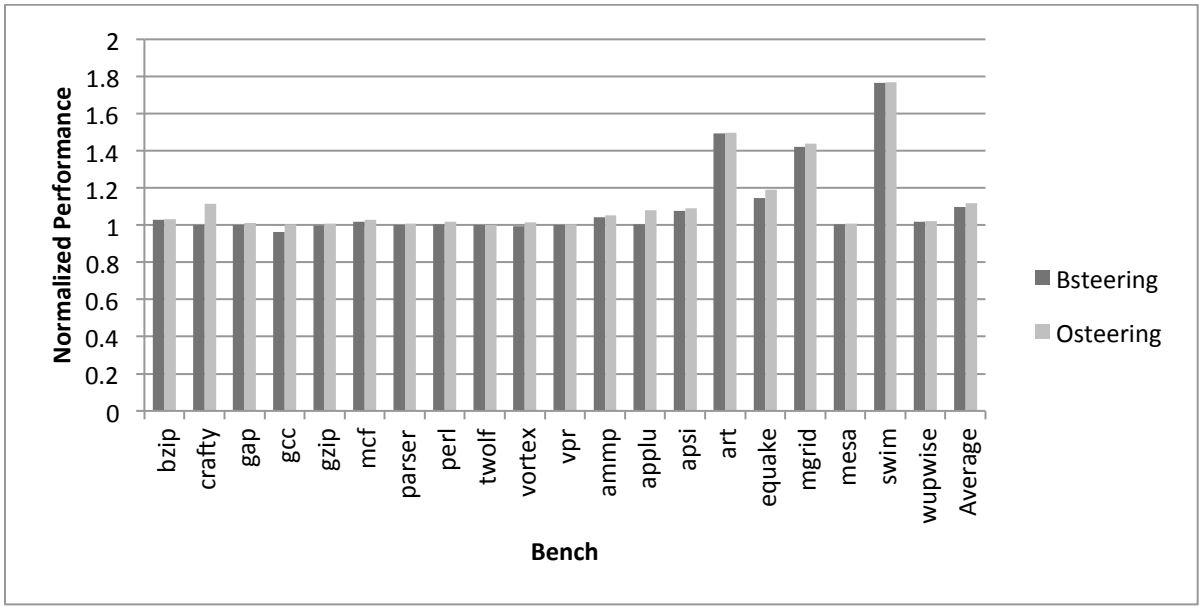

Figure 3.11: Performance of bottleneck-driven steering algorithm (Bsteering) in the unconstrained 2-core-type HCMP.

3.4.3.4 Changing the Thread Migration Overhead

Figures 3.12 and 3.13 show sensitivity to the thread migration overhead. As we increase the thread migration overhead to 1,000 cycles, the average performance gain of the bottleneck steering algorithm reduces to 11%. If we further increase the thread migration overhead to 10,000 cycles, the average performance gain reduces to 10%. This shows that even with a one or two order of magnitude increase in thread migration overhead, the performance does not reduce drastically.

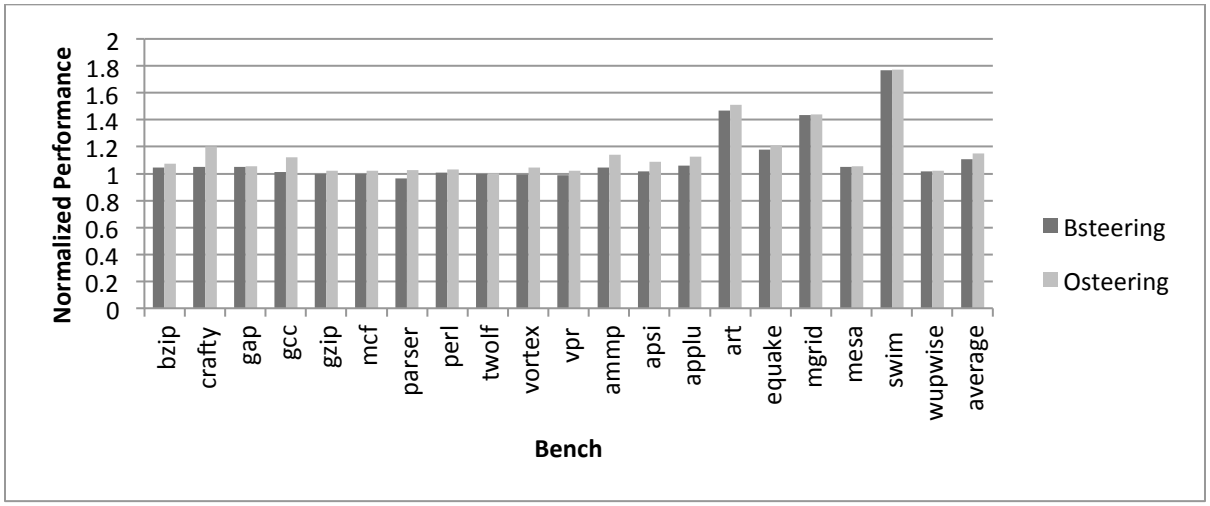

Figure 3.12: Performance of bottleneck-driven steering algorithm (Bsteering) in the unconstrained 4-core-type HCMP for 1,000-cycle thread migration overhead.

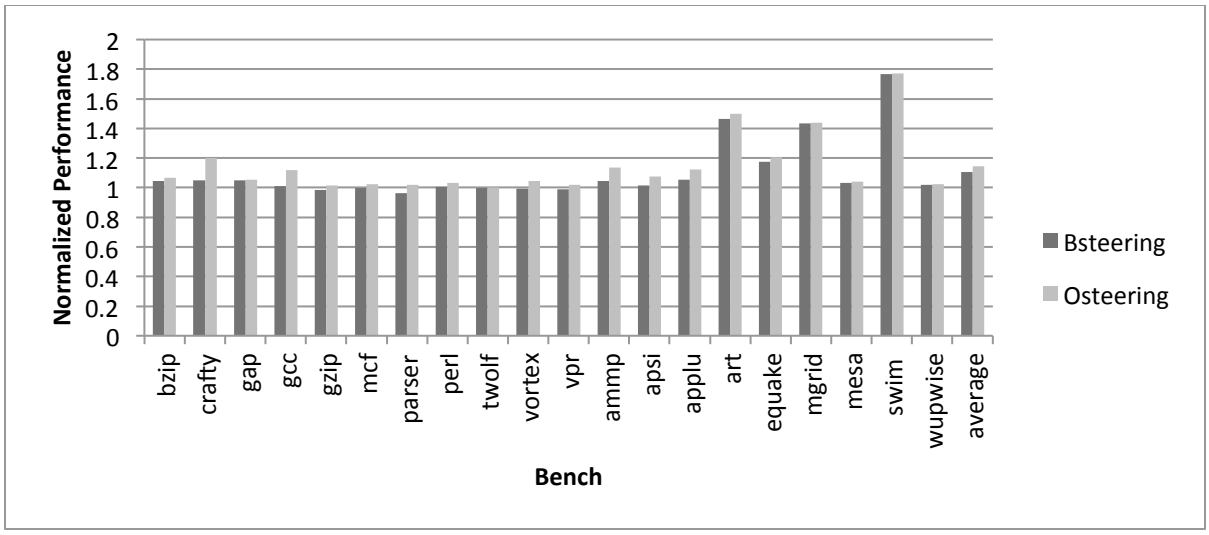

Figure 3.13: Performance of bottleneck-driven steering algorithm (Bsteering) in the unconstrained 4-core-type HCMP for 10,000-cycle thread migration overhead.

CHAPTER 4: BOTTLENECK-DRIVEN DESIGN SPACE EXPLORATION

As pointed out before, another key challenge in designing a non-monotonic HCMP from the ground-up is that it takes a lot of time to perform the core selection. This is because it requires exhaustive simulation of each application phase on each design point in the core design space. To address this, researchers have used subsetting-based core selection approaches [50][56]. It basically consists of two steps. First, accelerated design space exploration is performed to find the best core for each application phase in a representative benchmark suite. Then, a core-subsetting algorithm (e.g., configurational workload characterization [56]) is used to find the N core-types, which can fit in the chip, from the best-core information derived in the first step. Please note that the first step of design space exploration - finding the best core - takes all the time. In this chapter, we show how bottleneck analysis can be used to accelerate the design space exploration. The next chapter details the core-subsetting algorithm for core selection in a non-monotonic HCMP, once the best core information is known.

This chapter is organized as follows. In Section 4.1, we provide background on superscalar processors including the factors that affect their performance. Section 4.2 describes how criticality information is used to drive the design space exploration. Section 4.3 describes our high-fidelity superscalar design space. Criticality analysis using our detailed criticality model of a superscalar processor is explained in Section 4.4. Sections 4.5 and 4.6 describe our experimental methodology and results.

4.1 Background on Superscalar Processors

4.1.1 Dimensions of a Superscalar Processor

A superscalar processor exposes and exploits instruction-level parallelism (ILP) in programs. The pipeline of a superscalar processor can be characterized along three dimensions:

- The sizes of ILP-extracting units (issue queue, load and store queues, physical register file / reorder buffer), caches, and predictors.
- The widths of pipeline stages.
- Clock frequency.

Another dimension is pipeline depth. Each high-level pipeline stage might be subdivided into multiple sub-stages. We refer to the number of sub-stages as the *depth of the pipeline stage*. The depth of a pipeline stage is not treated as an independent parameter; rather, it is determined by frequency and the total propagation delay (logic and wire delays) of the pipeline stage.

4.1.2 Understanding Performance of a Superscalar Processor

When designing a superscalar processor, there is a tradeoff between accelerating the execution of *independent instructions* (*i.e.*, exposing and exploiting more ILP) and reducing the latency of *dependent instructions*, and this tradeoff plays out differently for different programs and program phases. Exploiting more ILP requires increasing the sizes of ILP-extracting units and the widths of pipeline stages, increasing their propagation delays. Deeper pipelining can help maintain the clock frequency despite the longer delays. However, the longer delays increase the latency of dependent instructions. Whether overall performance

increases or decreases depends on which factor, parallelism or latency, ends up dominating. Different programs and program phases have different arrangements of independent and dependent instructions. This causes them to be characterized by different optimal pipeline designs.

4.1.3 Pipeline Loops

Above, we pointed out that longer propagation delays increase the latency of dependent instructions irrespective of deeper pipelining. The reason is that dependent instructions exercise pipeline loops [7]. Three well-known pipeline loops are:

1. Control-dependence loop: This loop is exposed when a branch is mispredicted. The execute stage redirects the fetch unit in the case of a mispredicted branch. The delay of this loop is the total propagation delay between the fetch and execute stages, plus the latency of misprediction recovery.

2. Register-dependence loop: This loop is exposed for data-dependent instructions (but mainly single-cycle producer instructions). The delay of this loop is the propagation delay of the wakeup-select logic in the issue unit, and determines the minimum time to execute chains of data-dependent instructions.

3. Memory-dependence loop: This dependence is exposed when there is a store instruction followed by a load instruction to the same memory location within the pipeline. The delay of this loop is equal to the propagation delay of store-to-load forwarding.

4.2 Criticality-driven Design Space Exploration

Simulated annealing is one of the state-of-the-art classical search/optimization techniques and has a proof for theoretical convergence [49]. Random walk can be considered

as a simpler version of simulated annealing. As both random walk and simulated annealing do not get stuck in local maxima, they are widely used in global optimization. In Section 4.2.1, we explain random walk and simulated annealing. In Section 4.2.2, we discuss how to use criticality information to drive random walk and simulated annealing. Section 4.2.3 describes criticality-driven design space exploration using other search/optimization techniques.

4.2.1 Classical Search Techniques

4.2.1.1 Random Walk

Random walk is one of the simplest but widely used search/optimization techniques used to perform global optimization. In this section, we describe how random walk can be used to perform design space exploration of superscalar processors [49].

A random design point is used as the starting point. Simulation is performed at this design point to obtain the performance value. A parameter is selected at random from the current design point and it is changed by the least significant unit to get a new design point. We call this a *random perturbation*. Simulation is again performed at the new design point to get its performance value. Random perturbation is then performed at the new design point and this process is continued. During the entire process, the design point which generated the best performance value is kept track of. If the performance value at the current design point dips below 50% of the best performance value, then we restart from the best design point. Because of the restarting procedure, a lot of iterations are not wasted in searching through bad design points.

As random walk performs random perturbations and accepts new design points which

are worse than the current design point, it does not get stuck in local maxima.

4.2.1.2 Simulated Annealing

Simulated annealing shares the spirit of random walk except that there is a probability associated with the acceptance of a new design point. If the new design point is not accepted, then random perturbation is again performed at the current design point to get another design point.

The probability function is given as below:

$$P = \begin{cases} 1 & \text{if } perf_{new} \geq perf_{current} \\ e^{\left(\frac{perf_{new} - perf_{current}}{T}\right)} & \text{if } perf_{new} < perf_{current} \end{cases}$$

Where $perf_{new}$ is the performance at the new design point, $perf_{current}$ is the performance at the current design point, and T is the temperature function which is high at the start of the design space search and decreases every 20 iterations according to following equation:

$$T = T_0 \cdot (0.9)^{n/20}$$

Where T_o corresponds to the initial temperature (constant) and n is the number of design points traversed.

Note that the probability of accepting the new design point is always 1 if the new design point is better than the current design point. However, if the new design point is worse than the current design point, then the probability of accepting the design point is close to 1 at the beginning of the design space search when the temperature is high. However, as the design space search progresses, the probability function tends to 0. In other words, design space search starts as a random walk. However, when it reaches close to the optimal region, it only

accepts better design points (gradient ascent). This prevents the wastage of unnecessary iterations when the optimal region is close. However, the initial temperature has to be carefully set. Otherwise, gradient ascent might start too early or too late. This might result in simulated annealing performing worse (taking more iterations) than random walk.

4.2.2 Criticality-driven Search Techniques

Although random walk and simulated annealing are much better than exhaustive design space search, using knowledge from performance bottleneck analysis can make it even faster. In criticality-driven design space search, we use criticality-driven perturbation instead of random perturbation to give random walk and simulated annealing a localized view of the region around the design point. This avoids unnecessary simulations in the localized region and, hence, accelerates the design space exploration. Section 4.2.2.1 explains criticality-driven perturbation. Sections 4.2.2.2 and 4.2.2.3 discuss how criticality-driven perturbation is used in criticality-driven walk and criticality-driven simulated annealing, respectively.

4.2.2.1 Criticality-driven Perturbation

To perform criticality-driven perturbation at the current design point, criticality analysis is performed along with the simulation of the design point. Criticality analysis assigns cycles from the total execution cycles to each of the processor resources (structure sizes and widths of pipeline stages) and program dependences (control, register and memory dependences). The processor resource/program dependence which has the most cycles attributed to it is the most critical bottleneck. Criticality analysis gives the most critical bottleneck at the given design point. Using this bottleneck information, the design parameter to be perturbed to improve the performance is found in the following way.

If the most critical bottleneck is a processor resource, the implication is that there is more instruction-level parallelism (ILP) to be extracted but doing so is prevented by the limited resource. Therefore, the next point in the design space will be based on increasing the size of that resource (structure size or width of a pipeline stage). Also, the affected pipeline stage will be increased in depth (sub-pipelined deeper) to accommodate the larger structure or width while maintaining the current clock frequency. However, if the most critical bottleneck is a program dependence, it implies that the processor resources are not critical. This means that the resources of the processor are oversized at the current design point. In this case, the clock frequency is increased while keeping the pipeline depth of each pipeline stage the same, which would decrease all the processor resources. This helps to increase the performance as the clock frequency is increased.

We see that using the most critical bottleneck always leads to increasing a parameter. However, we also need to decrease the design parameters to make sure that the entire design space is reachable. In fact, to make the entire design space reachable, we need to increase and decrease the design parameters with the same probability. To decrease the design parameters, criticality analysis is also used to find the least critical bottleneck (processor resource or program dependence which has the least number of cycles attributed to it). Using the least critical bottleneck, the design parameter is decreased to improve performance in the following way.

If the least critical bottleneck is a processor resource, it implies that the processor resource is oversized. Hence, the next design point is found by decreasing the oversized resource. The affected pipeline stage will be decreased in depth to accommodate the resource

while maintaining the current clock frequency. This helps to improve performance as the pipeline loop latency like branch misprediction loop latency, etc. [7], will decrease due to the decrease in pipeline depth corresponding to the processor resource. On the other hand, if the least critical bottleneck is a program dependence, it means that the processor resources are critical. This implies that all the processor resources are limited and more parallelism can be extracted by increasing the processor resources. Hence, the clock frequency is decreased while keeping the pipeline depth of each pipeline stage the same, which increases all the resources.

Table 4.1 gives a summary of the above discussion on the design parameter which needs to be perturbed based on the bottleneck information from the criticality analysis.

Table 4.1: Criticality-driven perturbation.

Bottleneck	Whether most/least critical	Design parameter to be perturbed	Comments
Processor resource	Most critical	Processor resource increased	Depth of stage increased
Program dependence	Most critical	Clock frequency increased	Decreases all resources
Processor resource	Least critical	Processor resource decreased	Depth of stage decreased
Program dependence	Least critical	Clock frequency decreased	Increases all resources

To understand if the criticality-driven perturbation yields performance improvement, we randomly picked 1,000 design points and for each performed a single criticality-driven perturbation. On average, we see that around 82% of criticality-driven perturbations resulted in a performance improvement. The remaining 18% of perturbations result in performance degradation.

We find that one of the major reasons for performance degradation is the balanced design points (local maxima). A balanced design point is a design point at which all the design parameters contribute roughly equal numbers of cycles to the total execution cycles. Hence, at a balanced design point, there is no bottleneck and any perturbation at this point may result in performance degradation.

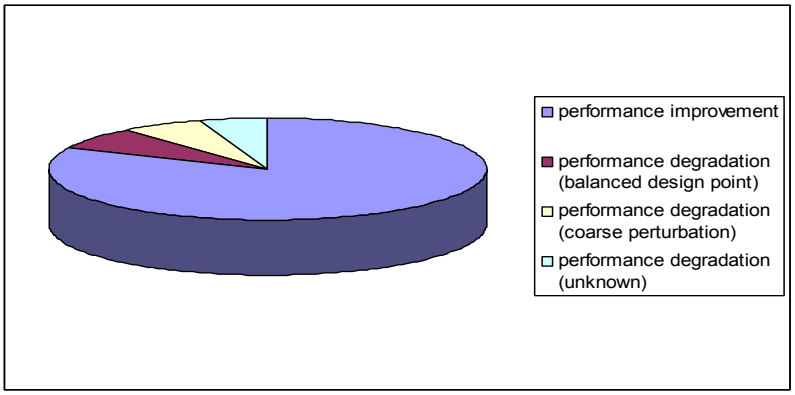

Figure 4.1: Breakdown of criticality-driven perturbation.

Other reasons for performance degradation could be the presence of parallel and near-critical paths. As we only model one critical path, the performance improvement is not guaranteed at these points [66]. Note that the fidelity of the criticality analysis would not affect the correctness of the design space exploration. It might just slightly slow down the design space exploration. However, our criticality model has enough fidelity to significantly accelerate the design space exploration.

To summarize, as 82% of criticality-driven perturbation results in a performance improvement, it shows that the criticality information is very useful in providing localized information to the design space exploration.

4.2.2.2 Criticality-driven Walk

In criticality-driven walk, to get a localized view, we perform criticality-driven perturbations instead of random perturbations. However, if a design point is revisited, then a random perturbation is performed instead, to avoid going in a loop.

At every iteration, criticality-driven walk tries to improve performance by selecting the parameter which would most likely yield performance improvement. However, when it reaches a balanced design point (local maximum) where all parameters are almost equally critical, the parameter that does get selected might not improve performance. Once it drifts apart from this balanced design point, it again tries to improve performance and ultimately reaches some other balanced design point (local maximum). After a certain number of iterations, it reaches a balanced design point which is the global maximum. Note that accepting design points which are worse than the current design point when it reaches local maxima helps criticality-driven walk avoid getting stuck at local maxima.

4.2.2.3 Criticality-driven Simulated Annealing

Similar to criticality-driven walk, to get a localized view, we use criticality-driven perturbation instead of random perturbation to generate a new design point. In case the new design point is not accepted, we resort to random perturbation. Also, if a design point is revisited, then a random perturbation is performed to avoid going in a loop.

During the beginning of the design space search, criticality-driven simulated annealing acts like a criticality-driven walk (high “temperature”). Hence, it reaches the optimal region in fewer iterations. Once it is close to the optimal region, it behaves like a gradient ascent method. Since criticality-driven perturbation has a localized view, the design point generated

is more likely to be accepted by the probability function compared to random perturbation. Hence it takes fewer iterations even when it is doing a gradient ascent. Hence, overall, criticality-driven simulated annealing performs better than simulated annealing.

4.2.3 Extension to Other Search/Optimization Algorithms

As other search/optimization algorithms use random perturbations, we can easily use criticality-driven perturbation in other search/optimization algorithms [61][38]. For example, in genetic algorithms, there are two operators to generate new design points: mutation and crossover [61][38]. Mutation is a random perturbation. Hence, we can use criticality analysis to drive genetic algorithms by changing the mutation operator to a criticality-driven perturbation operator. Also, criticality analysis can be used to find good crossover sites in the design points while performing a crossover operation.

4.3 Superscalar Design Space Creation

This section explains the process of creating a high-fidelity superscalar design space. Later, it discusses a mechanism to prune the superscalar design space.

The performance of a superscalar processor is a function of both instructions per cycle (IPC) and clock frequency. To take clock frequency into account, we need to measure propagation delay. Hence, we use detailed synthesizable Verilog models and physical designs of superscalar processors from the FabScalar toolset [14]. FabScalar defines a canonical superscalar pipeline shown in Figure 4.2. The Canonical pipeline stage library (CPSL) of FabScalar provides many different designs of each canonical pipeline stage that differ in their complexity (structure sizes and pipeline width) and depth.

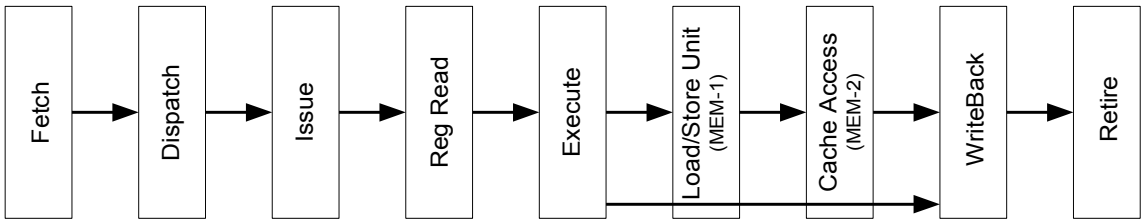

Figure 4.2: Canonical superscalar pipeline stages.

All possible combinations of pipeline stages that fit in a given clock period gives the design points for the given clock period. Doing this for different clock periods forms the entire design space of superscalar processors.

The following design space pruning strategy is used to decrease the size of the design space to a manageable size. Design space pruning is performed by weeding out among the design points having pipeline stages of same depth and same clock period. Among these design points having the same depth and clock period, the design point having pipeline stages with the highest complexity is retained and other design points having lower complexity are weeded out. This is because design points with lower complexity will always perform worse than the design point having highest complexity. For example, suppose issue widths of 2 and 4 for a 16-entry issue queue both fit into an issue depth of 2 and clock period of 0.4ns. We discard the design point having issue width of 2 as it would always perform worse than the design point with issue width of 4. Sometimes it might be hard to distinguish which of the alternative design points is most complex. In such cases, we don't perform pruning. For example, it is difficult to rank the complexity of a 16-entry issue queue with issue width 2 and an 8-entry issue queue with issue width 4. In the case of a design space exploration

within a fixed power/area budget, the above design space pruning is done only on those points, which fit within the budget.

4.4 Criticality Analysis

The design parameter to be perturbed in the criticality-driven design space exploration is identified using the bottleneck information of the criticality analysis. This section explains the process of finding out the bottlenecks for a given superscalar processor configuration using criticality analysis. Section 4.4.1 gives an overview of criticality analysis for a superscalar processor. Even though the issue queue lies at the heart of the superscalar processor, the issue queue was not modeled in past work on criticality analysis. Section 4.4.2 describes modeling of the issue queue in the criticality model. Section 4.4.3 illustrates the working of criticality analysis using an example. Section 4.4.4 gives the overhead of using criticality analysis.

4.4.1 Overview of Criticality Analysis

There are three main steps in finding the bottlenecks for a superscalar processor running a given application using criticality analysis.

1) Building of critical-path graph model

The critical-path graph model [26][64] of a processor is a graph-based model which helps to uncover the bottlenecks in the processor. It is a directed graph that models the resource constraints of the processor (structure sizes and pipeline widths) and control, register, and memory dependence constraints of the program currently being executed. It expresses the dependence relationships between different events occurring in the processor,

which are dictated by different constraints of the processor and the program being executed. It is built using the trace of committed instructions of the program being executed.

Each node in the graph represents an event when the instruction *enters* a given pipeline stage. For every non-memory instruction in the dynamic instruction stream, there is a fetch (F) node, a dispatch (D) node, an issue (I) node, a register-read (R) node, an execute (E) node, a write-back (W) node and a commit (C) node. There are two additional nodes, namely mem1 (M1) and mem2 (M2) for memory instructions. An edge in the graph corresponds to the dependence relationship between the two events (nodes) because of some resource or program constraint. If there is an edge from node A to node B and the weight on the edge is W, then the event B cannot occur earlier than W cycles after the event A has occurred. For example, if the fetch pipeline depth is 1, then there is a directed edge with weight 1 from the fetch (F) node to the dispatch (D) node of an instruction. However, there can be multiple incoming edges to a node. In that case, the edge that dominates will decide the time of the event.

The graph is dynamically built along with the simulation of the given design point (processor configuration). As an instruction is committed in the simulator, nodes corresponding to all the pipeline stages through which the instruction passed are added. Then, the edges incoming to these nodes are added according to resource constraints of the processor and dependence constraints of the program. According to the incoming constraints to the nodes, the nodes are scheduled. That is, the time (cycle) of the event corresponding to the node is determined. It should match with the simulator timestamp of the instruction entering the corresponding pipeline stages. Table 4.2 gives an overview of constraints

modeled by past research on criticality models [26][77].

Table 4.2: Overview of the constraints modeled by the criticality model of a superscalar processor.

Constraint Modeled	Edge	Comments
Pipeline dependence	$F_i \rightarrow D_i$, $D_i \rightarrow I_i$, $I_i \rightarrow R_i$, $R_i \rightarrow E_i$, $E_i \rightarrow W_i$, $W_i \rightarrow C_i$	Applicable to non-mem. instructions
	$F_i \rightarrow D_i$, $D_i \rightarrow I_i$, $I_i \rightarrow R_i$, $R_i \rightarrow E_i$, $E_i \rightarrow M1_i$, $M1_i \rightarrow M2_i$, $M2_i \rightarrow W_i$, $W_i \rightarrow C_i$	Applicable to mem. instructions
In-order pipeline stage constraints	$F_{i-1} \rightarrow F_i$, $D_{i1} \rightarrow D_i$,, $C_{i-1} \rightarrow C_i$	In-order constraint
	$D_{i-w} \rightarrow F_i$, $I_{i-w} \rightarrow D_i$	Width constraint, 0<w<=width
Reorder buffer	$C_{i-R} \rightarrow I_i$	R=ROB size
Load/Store Queue	$C_{K1} \rightarrow I_i$, $C_{K2} \rightarrow I_i$	K1 → bottleneck load instruction K2 → bottleneck store instruction
I-cache miss	$F_i \rightarrow D_i$, $F_i \rightarrow F_{i+1}$	Weight of the edge is i-cache miss latency
D-cache miss	$M2_i \rightarrow W_i$	Weight of the edge is d-cache miss latency
L2-cache miss	$F_i \rightarrow D_i$, $F_i \rightarrow F_{i+1}$, $M2_i \rightarrow W_i$	Weight of the edge is L2-cache miss latency
Issue width [77]	$R_b \rightarrow R_i$	b → instruction which consumed the last issue slot
Control dependence	$E_{i-1} \rightarrow F_i$	i-1 → mispredicted branch
Data dependence	$R_p \rightarrow R_c$	p → producer instruction of data c → consumer instruction of data
Memory dependence	$M2_p \rightarrow M2_c$	p → producer store instruction c → consumer load instruction

2) Finding the critical path

The critical path is the longest path from the starting node (fetch of the first instruction in the dynamic instruction stream) to the end node (commit of the last instruction in the dynamic instruction stream). The length (sum of weights of the edges) of the critical path is the execution time of the program. Hence, any parameter affecting the critical path would affect the performance of the processor.

3) Profiling the critical path to find the bottlenecks

Each edge in the program critical path is due to some processor resource (size of a structure or width of a pipeline stage) or program dependence (control, data, or memory dependence). Accordingly, we break down the entire critical path and assign contributions to different processor resources and program dependences. The processor resource or program dependence which has the most cycles assigned to it is the most critical bottleneck and the one which has the least cycles associated is the least critical bottleneck. Note that the most critical bottleneck could be a program dependence. This would happen if the resources of the processor are oversized for the program.

4.4.2 Modeling of the Issue Queue in the Criticality Model

The size of a FIFO structure is straightforward to model in the graph. As an example, consider a reorder buffer of size S. The reorder buffer cannot hold more than S instructions, and instructions enter and leave the reorder buffer in program order. This implies that instruction “i” cannot enter the reorder buffer before instruction “i-S” has left it. This constraint is enforced by inserting an edge between the relevant nodes of instruction “i-S”

and instruction "i".[2]

The key point here is that modeling the size S of a FIFO structure is simple because the *critical instruction* with respect to an instruction "i" is easily identified: it is always instruction "i-S".

In contrast, modeling the size of the issue queue is not straightforward because instructions leave it out-of-order. It was not modeled in previous criticality-based models [26] [42] [77]. Yet, the issue queue is one of the most performance-critical components in an out-of-order superscalar processor: the tradeoff between exposing more ILP and minimizing pipeline loop delay (wakeup-select loop) is particularly acute in the issue queue.

Consider an issue queue of size S. As with the reorder buffer, an instruction "i" cannot enter the issue queue before some prior instruction has left it. Unfortunately, because instructions leave the issue queue out-of-order, this critical instruction is not simply "i-S". The critical instruction could be any prior instruction to this point. Our approach is to maintain a list of S prior instructions that are the latest to leave the issue queue, among all prior instructions. That is, the list contains the S latest-departing instructions. The critical instruction is among these S instructions. Namely, the critical instruction is the one that leaves the issue queue first. Its departure makes room for instruction "i". Accordingly, an edge is inserted between the relevant nodes of this critical instruction and instruction "i".[3]

[2] In particular, the edge is between the commit (C) node of instruction "i-S" and the issue (I) node of instruction "i": $C_{i-S} \rightarrow I_i$. Note that the commit node corresponds to an instruction leaving the reorder buffer and the issue node corresponds to an instruction entering the reorder buffer and other structures (issue queue and possibly the load or store queue).

[3] In particular, the edge is between the register-read (R) node of the critical instruction and the issue (I) node of instruction "i": $R_{critical} \rightarrow I_i$. Note that the register-read node corresponds to an instruction leaving the issue queue and the issue node corresponds to an instruction entering the issue queue.

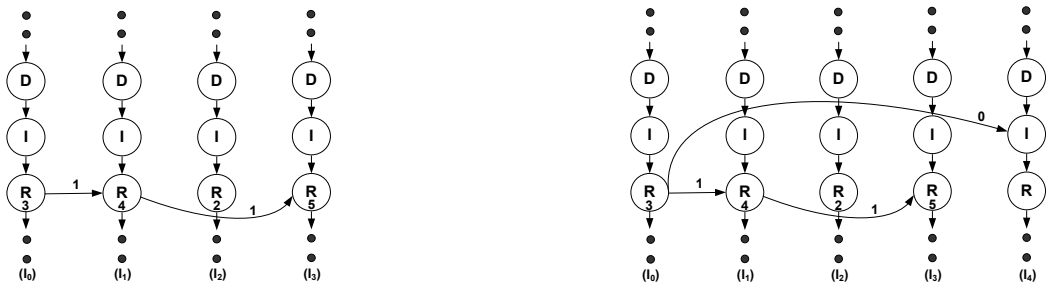

Figure 4.3: Modeling the issue queue in the criticality model.

The list of S instructions must be updated after each instruction "i" is added to the graph. This is simple to do: the critical instruction as identified above is removed from the list and instruction "i" is added to it. The critical instruction is the earliest-departing one and instruction "i" will necessarily have a later departure (since the critical instruction made way for it). Thus, the new list of S latest-departing instructions excludes the critical instruction and includes instruction "i".

Figure 4.3 (left) shows the criticality graph corresponding to instruction trace I_0, I_1, I_2, I_3. Note that the number on the edge is the weight of the edge and the number on the node is the time stamp corresponding to that node. For simplicity, we have not drawn the incoming issue queue edges for I_0, I_1, I_2 and I_3. The issue queue size is assumed to be 3. When the instruction I_4 is committed, the nodes corresponding to it are added and edges corresponding to pipeline dependences are added. The latest three instructions to be issued are I_0, I_1 and I_3 (set S = {I_0, I_1, I_3}). I_0 has the smallest register read (R) time stamp in set S. Hence, to model the issue queue bottleneck, we have an edge from R node of I_0 to I node of I_4. Figure 4.3 (right) shows the criticality graph corresponding to instruction trace I_0, I_1, I_2, I_3 and I_4.

4.4.3 Illustrative Example of Criticality Analysis

The resource constraints modeled are as follows: ROB size = 4, issue queue size = 2, issue width = 2, fetch width = 2, commit width = 2. The critical path model for the dynamic instruction trace for the above resource constraints is shown in Figure 4.4. The longest path from the fetch node of I0 (F0) to the commit node of I8 (C8) is shown in bold. This path is the critical path and has nodes F0, F1, D1, I1, R1, R2, R3, E3, W3, C3, I7, R7, E7, F8, D8, I8, R8, E8, W8, C8. The critical path is profiled and cycles are assigned to different processor resources and program dependences as follows: Register dependence = 2, Control dependence = 1, ROB = 1, unclassified = 15. Note that the unclassified cycles are the cycles attributed to pipeline dependences which cannot be attributed to any of the processor resources or program dependences. We clearly see that the register dependence is the most critical bottleneck. Note that many of the processor resources and program dependences have 0 cycles assigned to them because the dynamic trace in consideration is very short.

Dynamic Instruction trace

I_0 :R5=0
I_1 :R3=1
L1: I_2 :R1=R3+2
I_3 :R6=R1+2
I_4 :R3=R3+1
I_5 :R5=R6+R5
I_6 :cmp R7,0
I_7 :br L1
I_8 :R5=R5+100

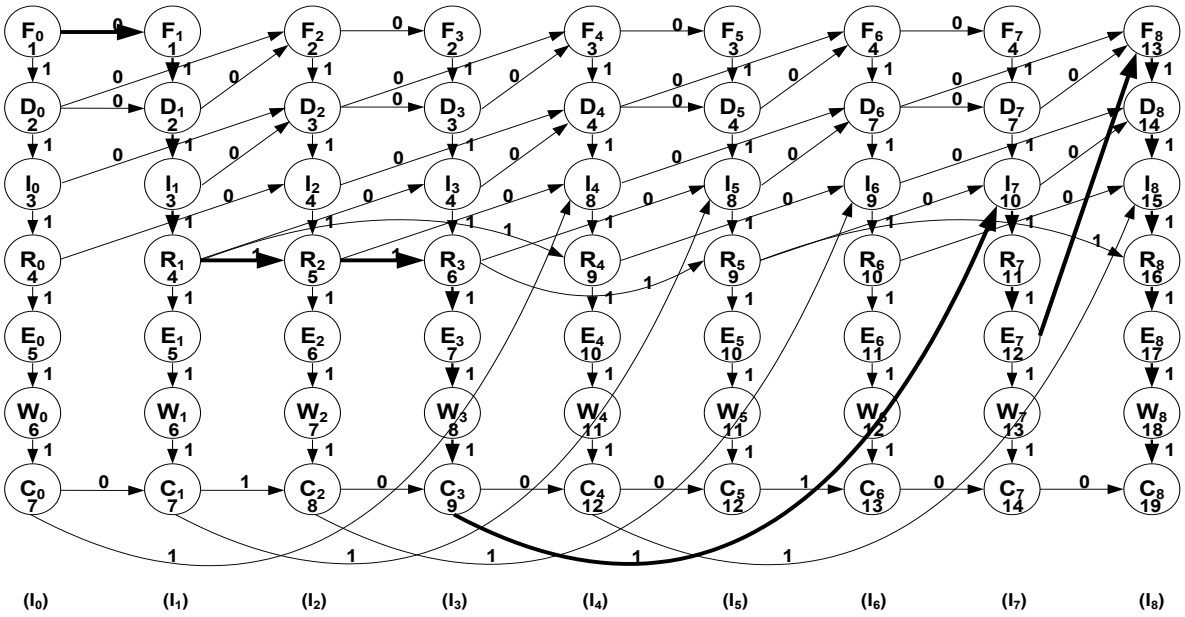

Figure 4.4: Criticality model of the dynamic instruction trace.

4.4.4 Overhead of Criticality Analysis

Simulation Time Overhead

Criticality analysis requires the processing of timestamps of instructions and bottleneck events (branch mispredictions and cache misses) to build the dependence graph. Hence, criticality analysis takes lesser time compared to the time taken for cycle-level simulation. In our simulations, criticality analysis takes less than 3% of the time taken to perform cycle-accurate simulation.

Storage Overhead

To perform the criticality analysis, the entire dependence graph obtained had to be built and stored. This incurs a large storage overhead. For example: a trace of 100 million instructions might take up to 10 GB of storage space. As done in [54], we remove this

overhead by not storing the entire graph and performing criticality analysis as the graph is being built. This is done by realizing that the edges to the nodes corresponding to future instructions come from only a handful of nodes. We see that the storage requirement of the critical path analysis model is of the order of number of constraints modeled. Hence, the storage requirement of the graph is drastically reduced.

Design Effort Overhead

We need the timestamps of the instructions entering different pipeline stages and the information of different bottleneck events like branch mispredictions, icache miss, dcache miss, etc. We can then build the dependence graph from this information. Hence, it requires minimal effort to retrofit the simulator to get the critical path information.

4.5 Experimental Methodology

This section describes the design space of superscalar processors and the simulation environment used for evaluation.

4.5.1 Design Space

Our design space consists of design points created by varying the clock period and complexity (structure sizes and pipeline widths) of different pipeline stages of a superscalar processor. The design space is spanned by the independent design parameters as enumerated in Table 4.3. Note that the depths of each pipeline stage are dependent parameters as explained in Section 4.1.1. The depth of a canonical pipeline stage for a given design point is the degree of sub-pipelining required so that the given complexity of the pipeline stage may fit in the given clock period.

Table 4.3: Microarchitectural design parameters.

Parameter	Value Range	Number
Front-end width	2, 4, 6, 8	**4**
Issue width	2, 4, 6, 8	**4**
Physical Register File size	32, 64, 128, 256, 512	**5**
Issue Queue size	16, 32, 64, 128	**4**
Load Queue/Store Queue sizes	8/8, 16/16, 24/24, 32/32, 40/40, 48/48, 56/56, 64/64	**8**
L1 Instruction Cache size	8KB, 16KB, 32KB, 64KB, 128KB	**5**
L1 Data Cache size	8KB, 16KB, 32KB, 64KB, 128KB	**5**
L2 Cache size	0.5MB, 1MB, 2MB, 4MB	**4**
Clock period	0.3→0.95 ns (granularity=0.05ns)	**14**

Our design space consists of *3.6 million design points*. By design space pruning as explained in section 4.3, we reduce the design space to about *104K design points*. Using design space pruning, we have eliminated nearly 97% of the design points from the original design space and this helps in fast design space exploration. Note that in our evaluation, we used the pruned design space for both criticality-driven design space exploration and baseline design space exploration.

4.5.2 Simulator and Benchmarks

For our evaluation, we used the SPEC 2000 benchmark suite with the given reference inputs [73]. We simulated the instruction trace using SimPoint [70], so that our simulation run is representative of the entire benchmark. Our simulator is a timestamp-based, trace-driven simulator based on SimpleScalar [4]. We use a trace-driven simulator to save time in the evaluation to make detailed evaluation of design space exploration possible. Note that

using a trace-driven simulator does not affect the general conclusions of the thesis proposal as we only treat it as a "black-box" to get the performance value.

4.5.3 Metric

In our evaluation, the design space exploration is performed for optimizing the performance of the superscalar processor. We use billions-of-instructions-per-second (BIPS) to quantify the performance of the superscalar processor. Our metric for the design space exploration is the exploration time, which is the amount of time elapsed before a design point is reached whose performance value is within 0.5% of the optimal performance value [25]. The optimal design point is found by doing an exhaustive design space search. The exploration time is normalized to the time taken by a single simulation of the bzip benchmark. Note that the exploration time includes the time taken to perform criticality analysis in the case of criticality-driven design space exploration.

4.6 Results

In this section, we present the performance results and sensitivity studies with the criticality-driven design space exploration. In Section 4.6.1, we discuss the performance results of criticality-driven walk and criticality-driven simulated annealing. In Section 4.6.2, we show the sensitivity studies with the change in the starting point, change in the size of the design space, and adding power constraints.

4.6.1 Performance of Criticality-driven Design Space Exploration

4.6.1.1 Criticality-driven Walk

In Figure 4.5(top), we see that for all benchmarks, criticality-driven walk (CRW) reaches

the optimal point much faster than random walk (RW). On average (harmonic mean), CRW obtains 3.8x speedup over RW. For some benchmarks, like gzip and parser, the speedup obtained is close to 10x. This shows that the criticality information is very useful in accelerating random walk.

4.6.1.2 Criticality-driven Simulated Annealing

For simulated annealing to be effective, the initial temperature should be carefully set. For finding a good initial temperature, we use the initial temperature of 1 to 100K. In Figure 4.5(bottom), we see that there is no single initial temperature which is good for all the benchmarks. However, an initial temperature of 10 is good for many benchmarks. *Hence, we use an initial temperature of 10 for all our evaluations*. Note that a significant amount of time must be invested in temperature tuning to make simulated annealing better than random walk.

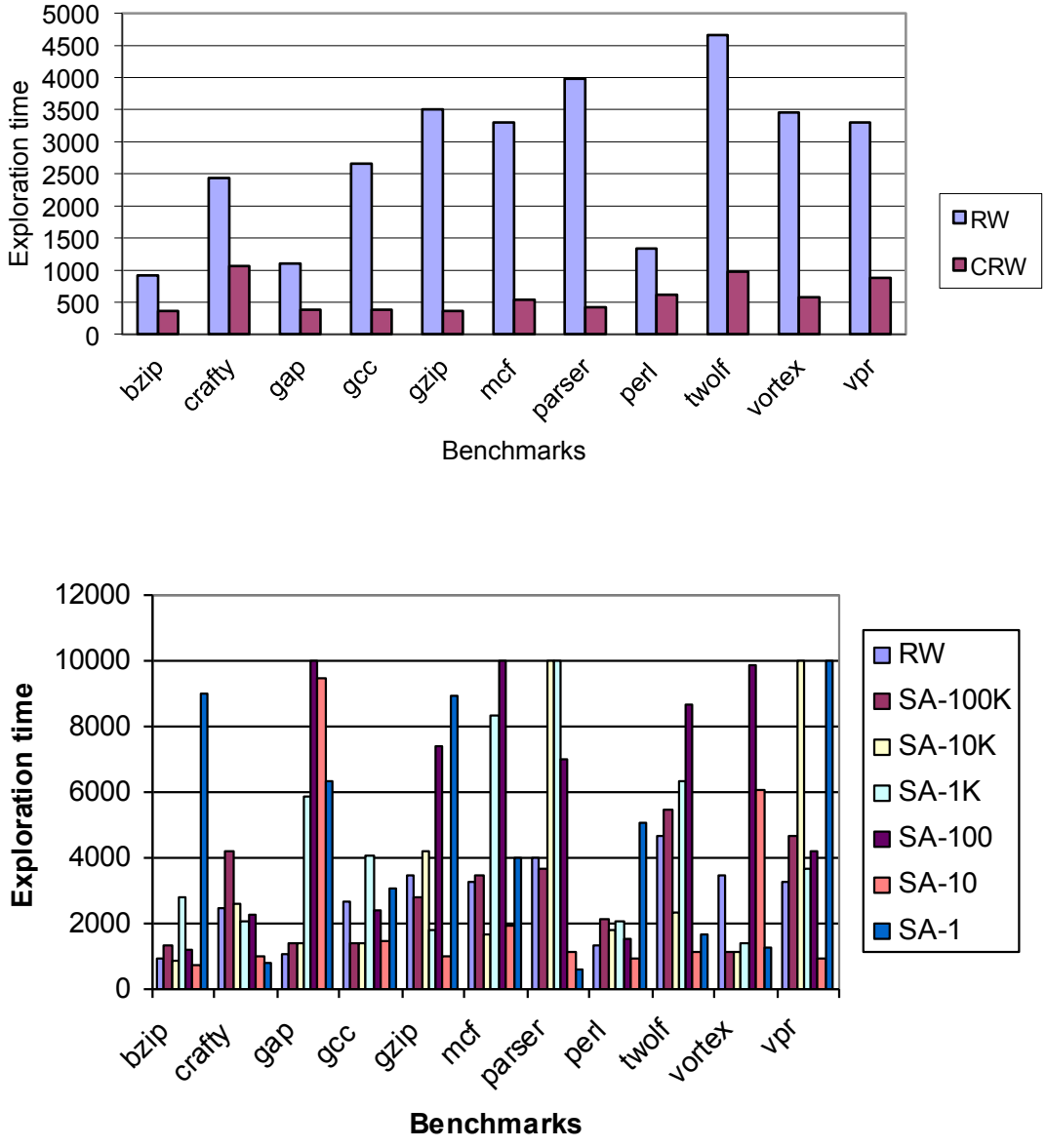

Figure 4.5: Comparison of criticality-driven walk (CRW) vs. random walk (RW) (top) and the effect of changing the initial temperature on simulated annealing (SA) (bottom).

In Figure 4.6(top), we see that using criticality information helps to speed up simulated annealing. On average (harmonic mean), we see that criticality-driven simulated annealing (CSA) obtains 2.3x speedup over simulated annealing (SA). Even after removing vortex and

gap, CSA obtains 1.9x speedup over SA. Using criticality analysis to guide simulated annealing helps to reach the optimal point faster than conventional simulated annealing.

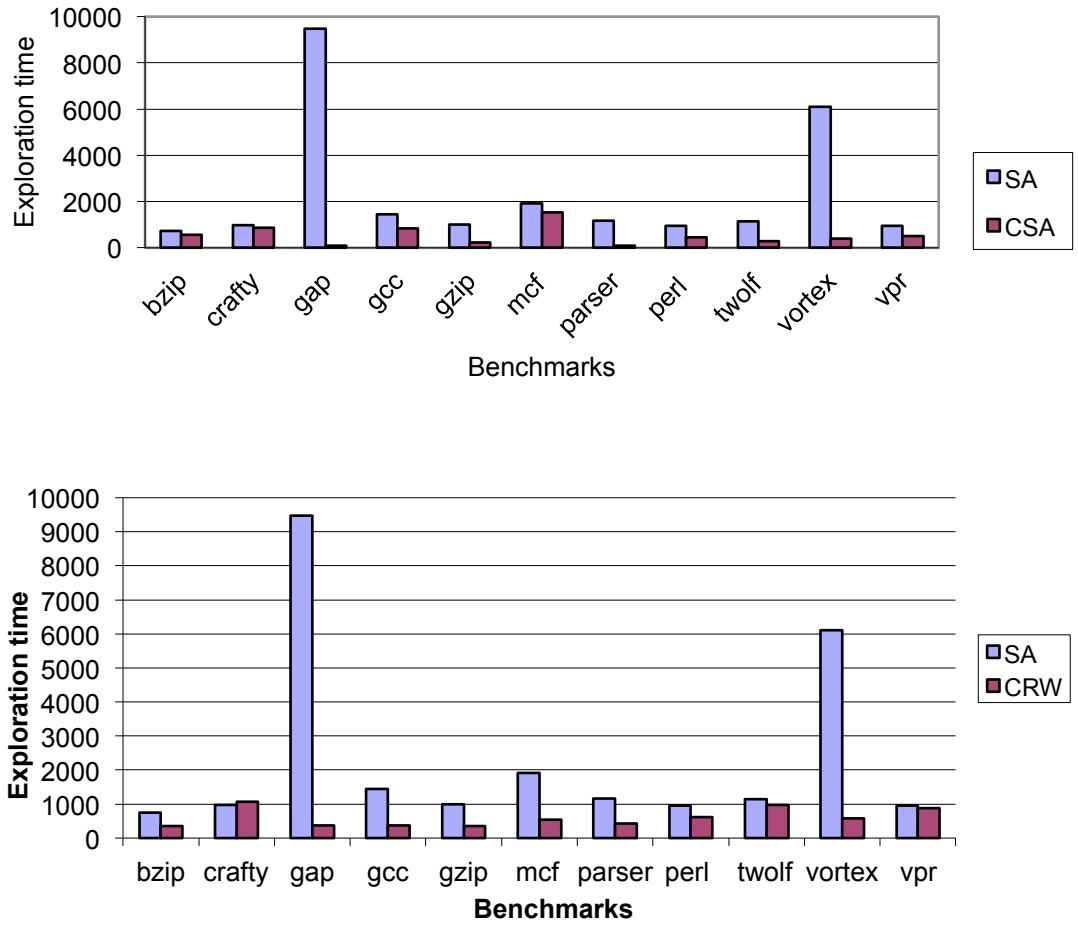

Figure 4.6: Comparison of criticality-driven simulated annealing (CSA) vs. simulated annealing (SA) (top) and comparison of simulated annealing (SA) and criticality-driven walk (CRW) (bottom).

As pointed out before, simulated annealing requires significant temperature tuning to outperform random walk. In Figure 4.6(bottom), we compare criticality-driven walk (CRW) and conventional simulated annealing (SA) and see that CRW is better than or comparable to

SA for the SPEC 2000 benchmark suite. Hence, this shows that building the criticality model is more useful than using more sophisticated algorithms like simulated annealing which are sensitive to tuning.

4.6.1.3 Dynamic Behavior of Criticality-driven Design Space Exploration

To understand how the performance of the best design point found changes with exploration time for criticality-driven design space exploration, we plot the IPT of the best design point found with the exploration time, for criticality-driven walk and criticality-driven simulated annealing, in Figure 4.7. For brevity, we only show the graph for the bzip benchmark. However, other benchmarks also show similar trends. We see that the localized information from criticality analysis helps criticality-driven design space exploration reach better design points more quickly.

4.6.2 Sensitivity Studies

To understand how robust our technique is, we explore sensitivity studies to the starting point, the size of the design space, and a power-constrained core design space.

4.6.2.1 Changing the Starting Point

We choose two other random starting points. Figure 4.8(top) shows the normalized exploration time needed for criticality-driven walk (CRW) and random walk (RW) for the two starting points. We see that CRW outperforms RW for all benchmarks except gcc in the second starting point. In Figure 4.8(bottom), we see that criticality-driven simulating annealing (CSA) performs better than simulated annealing (SA) for all benchmarks except mcf and vpr in the first starting point. Because of non-deterministic behavior (random perturbations) of random walk/simulated annealing, in very few cases, it might go through a

slightly shorter route to the optimal design point. However, in general, criticality-driven design space exploration is much better than conventional design space exploration.

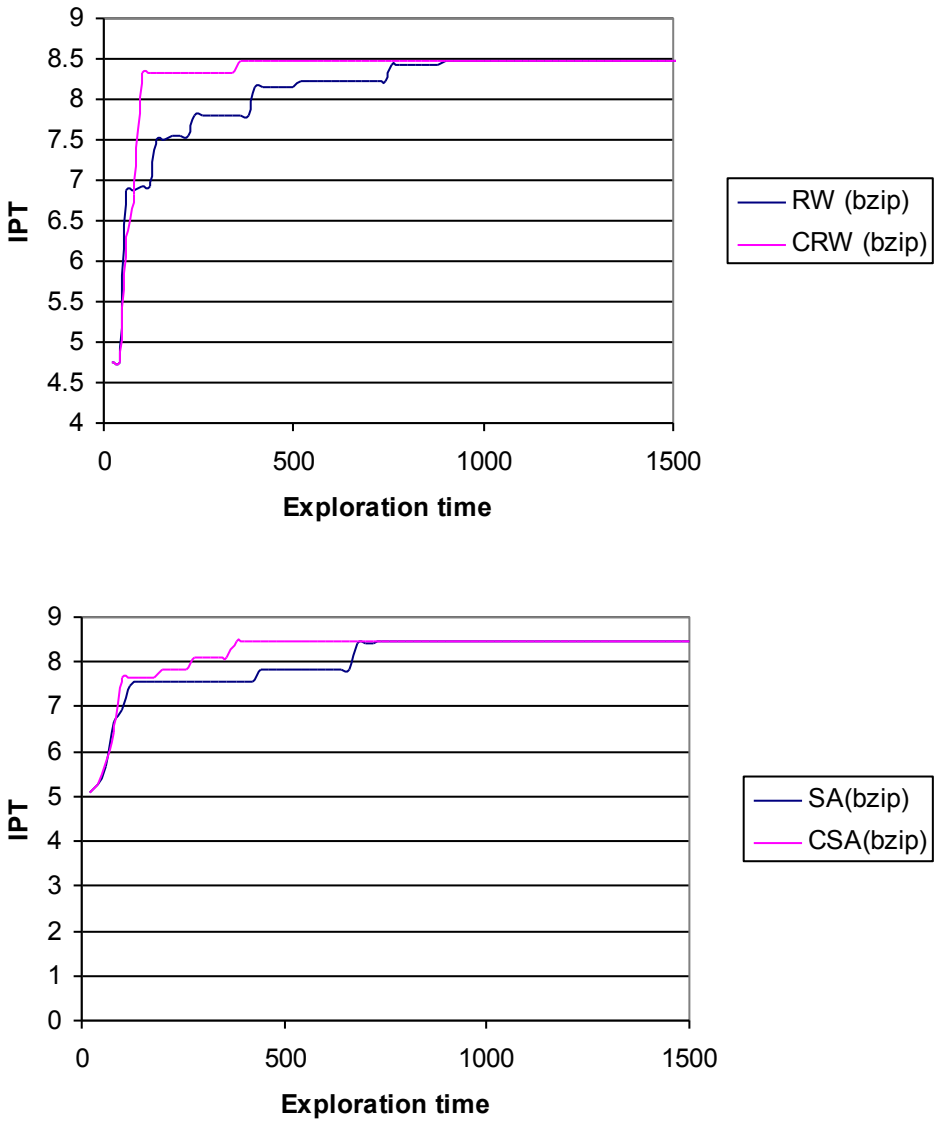

Figure 4.7: Performance of the best design point found as a function of exploration time, for criticality-driven walk (CRW) vs. random walk (RW) (top) and for criticality-driven simulated annealing (CSA) vs. simulated annealing (SA) (bottom), for the bzip benchmark.

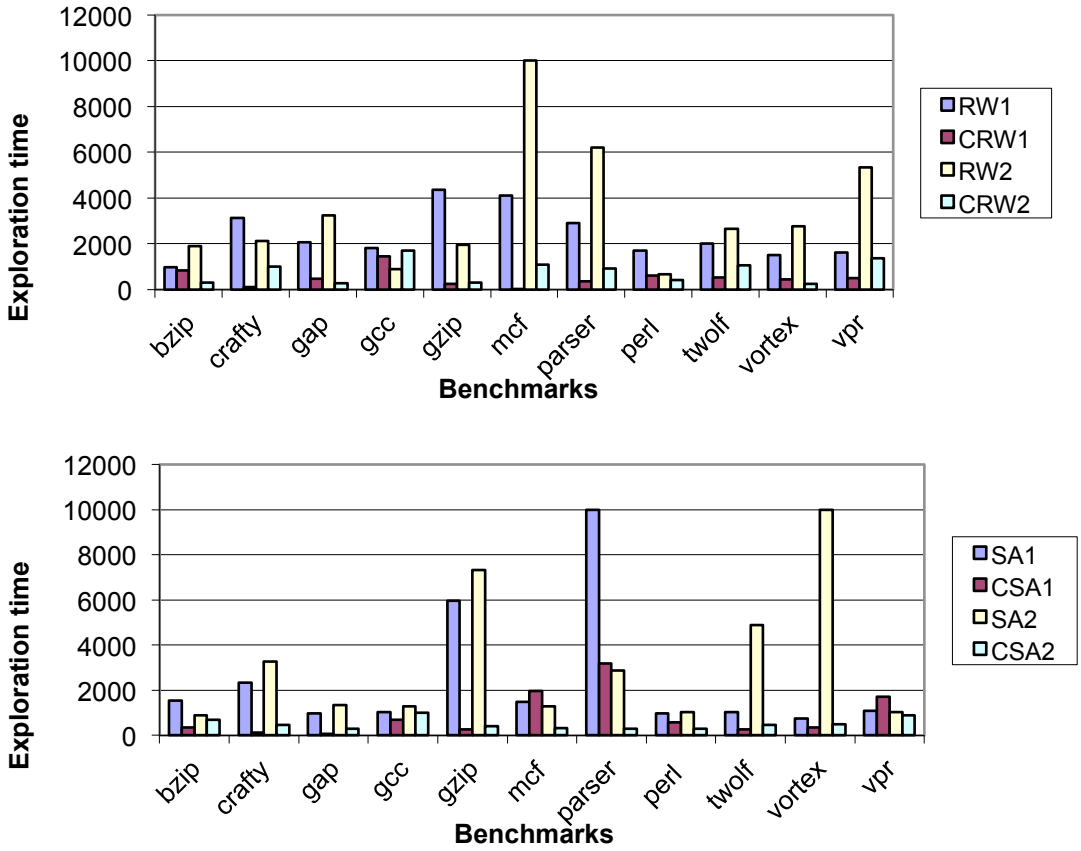

Figure 4.8: Effect of changing the starting point on criticality-driven walk (CRW) (top) and criticality-driven simulated annealing (CSA) (bottom).

4.6.2.2 Changing the Size of the Design Space

We decrease the size of the design space to about 60,000 design points by increasing the clock period granularity from 0.05ns to 0.1ns. In Figure 4.9, we see that even for the smaller design space, criticality-driven walk and criticality-driven simulated annealing perform better than random walk and simulated annealing, respectively. Further, we see that the temperature selected for simulated annealing for the large design space is not suited for the small design

space, as simulated annealing performs worse than random walk on average.

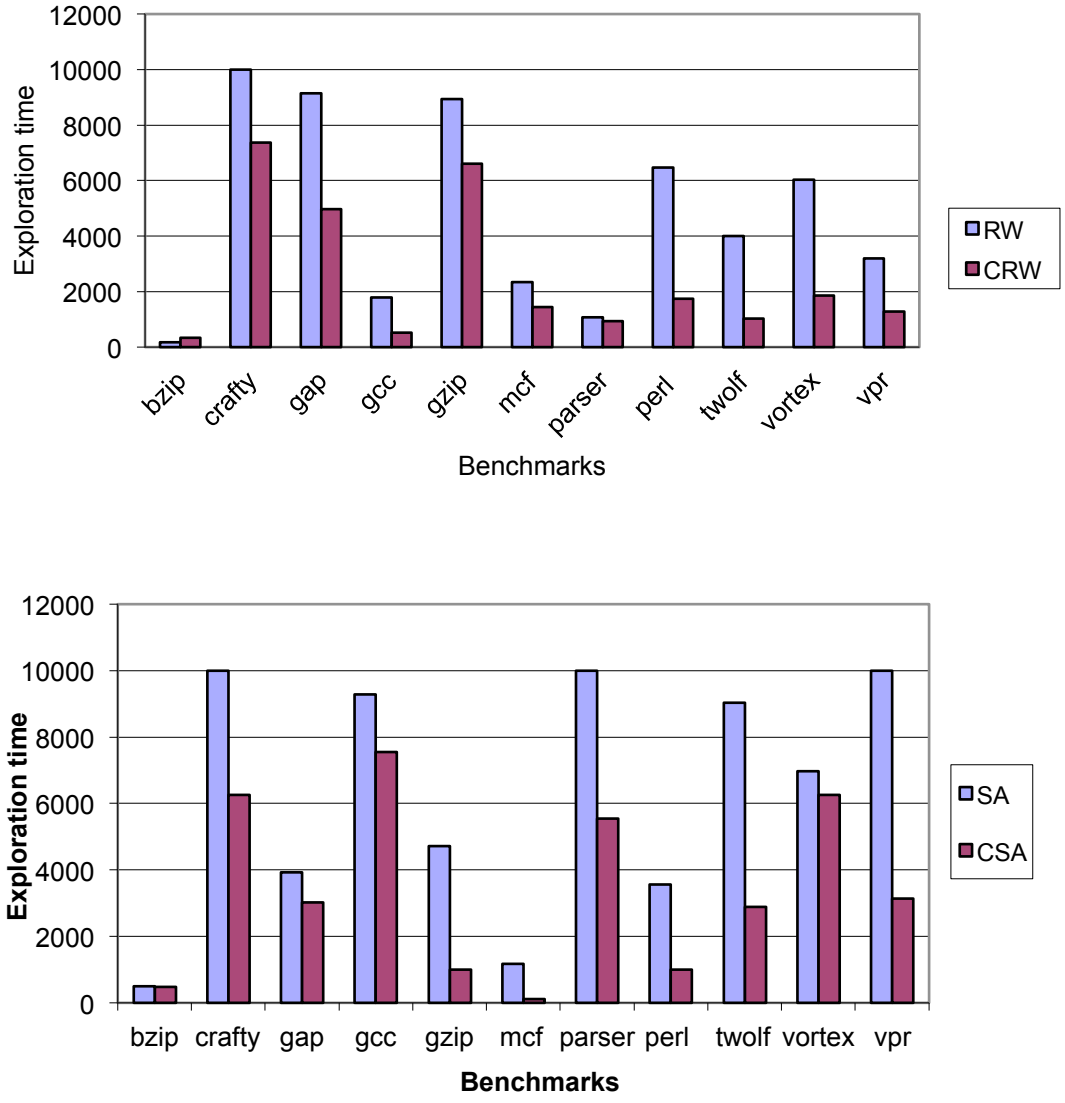

Figure 4.9: Effect of making the size of the design space smaller on criticality-driven walk (CRW) (top) and criticality-driven simulated annealing (CSA) (bottom).

4.6.2.3 Adding Power Constraint

For modeling power for a given processor configuration, we use the energy numbers from the FabScalar PPA tool. For details, please refer to Section 3.1.1.1. We calculate the peak power of the processor by assuming all the components, memory structures and combinational logic elements, are switching with an activity factor of 1.

For the sensitivity analysis, we constrain the design space by keeping the peak power budget of the processor at 50 W. In Figure 4.10, we see that even after adding the power constraint, criticality-driven design space exploration performs better than conventional design space exploration. However, it is slightly less effective than before. This is because some criticality-driven perturbations lead to design points which exceed the power budget and, in these cases, criticality-driven design space exploration has to rely on random perturbations.

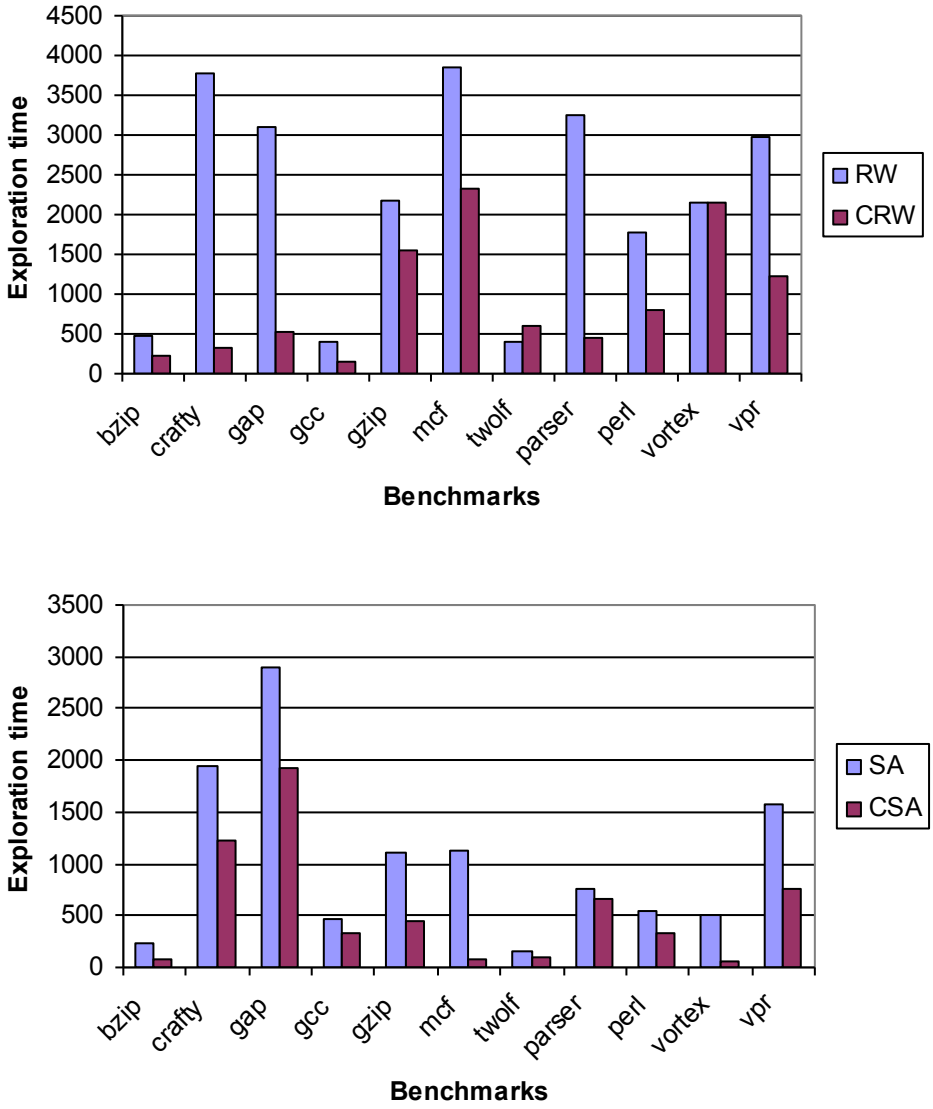

Figure 4.10: Effect of adding peak power budget of 50 W on criticality-driven walk (CRW) (top) and criticality-driven simulated annealing (CSA) (bottom).

CHAPTER 5: CORE SELECTION USING SUBSETTING ALGORITHM

To reduce the large simulation overhead in the core selection of HCMPs, we are exploring subsetting-based core selection approaches. The last chapter dealt with accelerating the design space exploration for finding the best core for a given program phase. This chapter explores HCMP core selection using a subsetting algorithm, once the per-phase best core information is known.

Section 5.1 explains our methodology for performing the subsetting-based core selection. Section 5.2 compares the result of performing subsetting-based core selection with the genetic-algorithm-based core selection, which we used in Chapter 3.

5.1 Methodology

We use the same simulation infrastructure and core design space as in Chapter 3. We obtain 39 ten-million-instruction phases from the SPEC 2000 benchmark suite using the SimPoint tool [70] as shown in Table 5.1. We use the FabScalar C++ simulator [13] for obtaining the IPCs of different application phases. To get a reliable estimate of clock period, we make use of the FabScalar toolset [13]. FabScalar's Performance-Power-Area (PPA) tool provides propagation delay for each and every pipeline stage design in the entire Canonical Pipeline Stage Library (CPSL), and these delays are from synthesis and FabMem (for highly ported RAMs/CAMs). The goodness metric for the HCMP design is the harmonic mean of the performance values (BIPS) of the 39 SimPoint Phases.

The core design space consists of design points created by varying the clock period and the complexity (structure sizes and pipeline widths) of different pipeline stages of a superscalar processor. The design space is spanned by the independent design parameters enumerated in Table 5.2. The depth of a canonical pipeline stage for a given design point is the degree of sub-pipelining required so that a given complexity of the pipeline stage may fit in a given clock period. A cartesian product of all parameter values gives *3.3 million design points*. Not all of these design points are valid. Firstly, as we are using the FabScalar [13] canonical template, a pipeline stage cannot be made arbitrarily deep. Secondly, for a canonical pipeline stage, only those design points are considered that have the largest-sized structure for a given width, depth, and frequency of that canonical pipeline stage. This helps in pruning the design space and retaining good design points. By performing this design space pruning, we are able to restrict it to *13,966 design points*.

For the subsetting algorithm, we use configurational workload characterization (CWC) [56]. In CWC, after finding the best core for each application phase, we simply make take best cores as our core design space for HCMP core selection. That is, the core design space consists of up to 39 design points, each of which is the best core for a SimPoint phase. The core selection process reduces to finding the best N-core-type HCMP from these 39 design points. To evaluate the quality of CWC-based core selection, we compare it with the genetic algorithm-based core selection, which was applied in Chapter 3.

Table 5.1: SPEC 2000 SimPoints.

Benchmark	*Interval Number*
bzip	341, 1873, 3089, 9277
crafty	234, 5647, 12166, 21513
gap	3229, 15783, 16913, 21010
gcc	264, 473, 628, 873
gzip	779, 1361, 5030
mcf	2018, 3091
parser	5201, 10758, 19972, 24856
perl	18, 415, 781, 1309
twolf	2482, 8155, 15968, 27755
vortex	5877, 7432, 9025
vpr	1350, 3463, 6120

Table 5.2: Microarchitectural design parameters.

Parameter	Value Range	Number
Front-end width	2, 3, 4, 5, 6, 7, 8	**7**
Issue width	2, 3, 4, 5, 6, 7, 8	**7**
Physical Register File size	64, 128, 192, 256, 384, 512	**6**
Issue Queue size	16, 24, 32, 48, 64, 96, 128	**7**
Load Queue/Store Queue sizes	8/8, 16/16, 24/24, 32/32, 40/40, 48/48, 56/56, 64/64	**8**
L1 Instruction Cache size	8KB, 16KB, 32KB, 64KB, 128KB	**5**
L1 Data Cache size	8KB, 16KB, 32KB, 64KB, 128KB	**5**
L2 Cache size	2MB	**1**
Clock period	**0.5→1.2 ns (granularity=0.1ns)**	**8**

5.2 Results

Table 5.3 shows the best cores for the 39 SimPoint phases. Please note that there are only 24 cores, as some application phases prefer the same core. Once the best core information is known, we use CWC to perform the core selection of HCMPs having 1 core-type to 5 core-types. Table 5.4 shows the result. We see that GA and CWC produce the same cores for 1-core-type HCMP, 4-core-type HCMP and 5-core-type HCMP. For the 2-core-type HCMP and 3-core-type HCMP, only one of the core-types is different. To see how much difference this makes, we compare the performance of CWC and GA for different numbers of core-types (Table 5.5). We found that there is less than 0.4% difference in performance. This shows that CWC performs exceedingly well. Hence, the subsetting algorithm for core selection can help immensely in reducing the simulation effort in the core selection process and at the same time give almost the same performance as the detailed genetic algorithm.

Table 5.3: The best core table for the 39 SimPoint Phases.

Core-type	*Clock Period (ns)*	*ILP-extracting buffers (IQ, LSQ, ROB)*	*Widths (fetch, issue)*	*Caches (KB) (I, D)*	*Phases*
C1	0.5	32, 64, 64	2, 2	16, 16	mcf.2018
C2	0.5	32, 64, 64	3, 3	16, 16	mcf.3091
C3	0.6	48, 128, 192	3, 2	64, 64	vortex.9025, vortex.5877, vortex.7432
C4	0.6	32, 64, 64	3, 4	64, 64	perl.781, perl.415, gcc.873
C5	0.6	32, 128, 128	3, 4	64, 64	twolf.8155, twolf.2482, twolf.27755, parser.5201, parser.19972, parser.24856, gap.15783,
C6	0.6	24, 64, 64	3, 5	64, 64	perl.1309
C7	0.6	32, 128, 128	4, 4	64, 64	twolf.15968, gzip.5030, gap.21010
C8	0.6	32, 64, 64	4, 5	64, 16	perl.18
C9	0.7	48, 128, 384	3, 4	128, 128	gap.3229
C10	0.7	48, 128, 384	4, 4	8, 128	parser.10758
C11	0.7	48, 128, 128	4, 4	128, 128	crafty.21513, crafty.12166, crafty.5647, crafty.234
C12	0.7	64, 128, 192	4, 5	8, 128	mcf.3673
C13	0.7	48, 128, 192	4, 5	8, 32	vpr.1350, vpr.6120
C14	0.7	48, 128, 128	4, 5	128, 32	gcc.264
C15	0.7	32, 128, 128	4, 6	8, 32	vpr.3463
C16	0.7	48, 128, 192	5, 5	8, 128	bzip.1873
C17	0.7	32, 128, 128	5, 6	128, 32	gcc.628
C18	0.7	48, 96, 128	6, 5	8, 8	gzip.1361
C19	0.7	32, 128, 128	6, 6	8, 32	gzip.779
C20	0.7	32, 128, 128	6, 6	128, 32	gcc.473
C21	0.8	48, 96, 192	6, 7	32, 16	bzip.9277
C22	0.8	48, 96, 192	8, 7	32, 16	bzip.341
C23	0.9	48, 128, 256	7, 7	64, 16	gap.16913
C24	1	48, 128, 512	7, 8	128, 64	bzip.3089

Table 5.4: Comparison of the genetic-algorithm-based core subsetting (GA) with the configurational-workload-characterization-based core subsetting (CWC).

Number of core-types in HCMP	GA	CWC
1	C5	C5
2	C5, LW	C5, C13
3	C5, LW, C1	C5, C13, C1
4	C5, C9, C1, C20	C5, C9, C1, C20
5	C5, C9, C1, C20, C24	C5, C9, C1, C20, C24

Table 5.5: Ratio of performance of CWC and performance of GA.

Number of core-types in HCMP	Perf-CWC/Perf-GA
1	1
2	0.9969
3	0.9965
4	1
5	1

These positive results are profound for two reasons. First, subsetting approaches (e.g., CWC) have much lower complexity than non-subsetting approaches (e.g., GA) in terms of the second step of selecting an ensemble of core-types (after evaluating phases on core types). The second reason is more subtle, and even more profound. Since subsetting approaches only select from among phases' best core-types, they do not *require* that the first step, of evaluating phases on cores, be exhaustive. In particular, subsetting approaches are compatible with accelerated design space search algorithms for finding so-called best core-types, such as our criticality-driven classical search algorithms. In contrast, because a non-

subsetting approach uses the entire core design space for its palette of core-type candidates, each phase must be exhaustively evaluated on the entire core design space.

CHAPTER 6: SUMMARY AND FUTURE WORK

In this chapter, we will first summarize the thesis and then discuss possible future work.

6.1 Summary

As we enter the dark-silicon era, HCMPs are an attractive substrate for improving single-thread performance and energy efficiency. The most powerful class of HCMP employs *non-monotonic core types,* where each core type is performance-optimized to different instruction-level behavior and hence cannot be ranked – different applications achieve their highest performance on different cores. Although non-monotonic heterogeneous designs offer higher performance potential than either monotonic heterogeneous designs or homogeneous designs, steering applications to the best-performing core is challenging due to performance ambiguity of core types.

We present a unified view of *selecting* non-monotonic core types at design-time and *steering* programs to cores at run-time. After a comprehensive evaluation, we found that with N core-types, the optimal HCMP for single-thread performance is comprised of an "average" core-type coupled with N-1 "accelerator" core-types that relieve distinct resource bottlenecks in the average core-type. This inspires a complementary steering algorithm in which the application is continuously monitored for performance bottlenecks. If the current bottleneck signature of the application does not match with that of the current core-type, the application is migrated to the core-type whose signature matches with that of the application.

In this dissertation, we also address reducing the simulation effort required in the core selection phase of HCMP design. We propose using detailed bottleneck analysis to accelerate design space exploration to find the best core for each application phase in the benchmark

suite. Once the best core information is known, core selection can be quickly performed by employing core-subsetting algorithms. This approach is not only much faster, but also produces excellent results that are very close to more exhaustive searches.

6.2 Future Work

6.2.1 Using Dynamic Voltage and Frequency Scaling in HCMP

Dynamic voltage and frequency scaling (DVFS) is a commonly used power management technique in different computing systems, from embedded systems to desktops to servers. It would be interesting to design an HCMP with a DVFS mechanism built-in for each processor core. Another interesting challenge is to design an application steering algorithm, which not only determines the best core at hand but also its frequency.

6.2.2 Extending the Core Design Space for HCMP Design

To remove resource and other bottlenecks in the processor, researchers have proposed many microarchitectural techniques over the years. Many of these microarchitectural techniques do not have universal applicability and only increase performance for certain applications. Techniques such as trace caches, value prediction, control independence, etc., may be highly beneficial under certain workload behavior and very cost-effective (area and power efficient) in the context of whole-pipeline customization. This presents a good opportunity for HCMP design where certain cores are designed with these techniques implemented. These cores can be powered on only for those applications that yield good performance.

6.2.3 Bottleneck Analysis for Other Architectures

In this dissertation, we have constructed a detailed bottleneck analysis model for the entire superscalar processor. It is easy to extend the bottleneck analysis model to other architectures, like Checkpoint Processing and Recovery (CPR) [2], Continual Flow Pipelines (CFP) [75], Trace Processors [60], Run-ahead execution [53], etc. It would be interesting to see the bottlenecks suffered by different applications in these architectures compared to traditional superscalar architectures.

REFERENCES

[1] M. Agarwal, N. Navale, K. Malik and M. I. Frank. Fetch-Criticality Reduction Through Control Independence, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2008.

[2] H. Akkary, R. Rajwar and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2003.

[3] D. H. Albonesi. Dynamic ipc/clock rate optimization, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 1998.

[4] T. Austin, E. Larson and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling, In IEEE Micro, February 2002.

[5] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2010.

[6] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures, In Proceedings of the International Conference on Computing Frontiers (CF), May 2006.

[7] E. Borch, E. Tune, S. Manne and J. Emer. Loose loops sink chips, In Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), February 2002.

[8] S. Borkar and A. A. Chien. The Future of Microprocessors, In Communications of the ACM, May 2011, Vol.54, No. 5.

[9] D. Brooks, V. Tiwari and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2000.

[10] J. A. Brown, L. Porter and D. M. Tullsen. Fast Thread Migration via Cache Working Set Prediction, In Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), February 2011.

[11] G. Cai, K. Chow, T. Nakanishi, J. Hall and M. Barany. Multivariate power/performance analysis for high performance mobile microprocessor design, In Power Driven Microarchitecture Workshop, June 1998.

[12] J. Chen and L. K. John. Efficient Program Scheduling for Heterogeneous Multi-core Processors, In Proceedings of the Design Automation Conference (DAC), July 2009.

[13] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi and E. Rotenberg. FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template, In Proceedings of the International Symposium on Computer Architecture, June 2011.

[14] N. Choudhary, S. Wadhavkar, T. Shah, S. Navada, H. Hashemi and E. Rotenberg. Fabscalar, In Workshop on Architecture Research Prototyping (WARP), June 2009.

[15] K. Chow and J. Ding. Multivariate analysis of pentium pro processor, In Intel Software Developers Conference, October 1997.

[16] D. Citron. The use and abuse of SPEC: An ISCA panel, In IEEE Micro, July 2003.

[17] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance Implications of Single Thread Migration on a Chip Multi-Core, In SIGARCH Computer Architecture News, vol. 33, no. 4, 2005.

[18] T. Conte, Systematic Computer Architecture Prototyping, PhD. Thesis, Department of Electrical Engineering, UIUC, 1992.

[19] K. V. Craeynest, A. Jaleel, and L. Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE), In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2012.

[20] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2002.

[21] A. Dhodapkar and J. Smith. Comparing program phase detection techniques, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2003.

[22] C. Dubach, T. M. Jones, E. V. Bonilla and M. F. P. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2010.

[23] C. Dubach, T. M. Jones and M. F. P O'Boyle. Microarchitectural Design Space Exploration Using An Architecture-Centric Approach, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2007.

[24] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam and D.C. Burger. Dark Silicon and the End of Multicore Scaling, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2011.

[25] S. Eyerman, L. Eeckhout and K. De Bosschere. Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors, In Proceedings of Design, Automation and Test in Europe (DATE), 2006.

[26] B. Fields, S. Rubin and R. Bodik. Focusing processor policies via critical-path prediction, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2001.

[27] B. Fields, R. Bodik, M. D. Hill and C. J. Newburn. Interaction cost: For when event counts just don't add up, In IEEE Micro, November 2004.

[28] K. Flautner, S. Reinhardt and T. Mudge. Automatic Performance-Setting for Dynamic Voltage Scaling, In Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM-7), 2001.

[29] D. E. Goldberg, (1994). Genetic algorithms in search, optimization, and machine learning, Reading, Mass Addison-Wesley.

[30] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. B. Taylor and S. Swanson, GreenDroid: A Mobile Application Processor for a Future of Dark Silicon, In Proceedings of HotChips, 2010.

[31] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput, In Proceedings of the International Conference on Computer Design (ICCD), October 2004.

[32] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers, In IEEE Micro, June 2011.

[33] M. Hill and M. Marty. Amdahl's law in the multicore era, In IEEE Computer, July 2008.

[34] L Ingber and B Rosen. Genetic algorithms and very fast simulated reannealing: A comparison, In Mathematical and Computer Modelling, 1992.

[35] E. İpek, S.A. McKee, B.R. de Supinski, M. Schulz and R. Caruana. Efficiently Exploring Architectural Design Spaces via Predictive Modeling, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2006.

[36] E. Jacobsen, E. Rotenberg and J. E. Smith. Assigning Confidence to Conditional Branch Predictions, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 1996.

[37] P. J. Joseph, K. Vaswani and M. Thazhuthaveetil. A Predictive Perfomance Model for Superscalar Processors, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2006.

[38] S. Kang and R. Kumar. Magellan: A Framework for Fast Muti-core Design Space Exploration and Optimization Using Search and Machine Learning, In Proceedings of Design, Test and Automation in Europe (DATE), March 2008.

[39] H. Kannan, M. Budiu, J. Davis and G. Venkataramani. Tuning SOCs using the Dynamic Critical Path, In Proceedings of the International System On-chip conference (SOCC), September 2009.

[40] T. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2004.

[41] T. Karkhanis and J. E. Smith. Automated Design of Application-Specific Superscalar Processors, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2007.

[42] T. Karkhanis. Automated Design of Application-Specific Processors, PhD. Thesis, Department of Electrical Engineering, University of Wisconsin-Madison, 2006

[43] U. R. Karpuzcu, B. Greskamp and J. Torrellas. The BubbleWrap Many-Core: Popping Cores for Sequential Acceleration, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2009.

[44] V. Kontorinis, A. Shayan, D. M. Tullsen and R. Kumar. Reducing peak power with a table-driven adaptive processor core, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2009.

[45] D. Koufaty, D. Reddy and S. Hahn. Bias Scheduling in Heterogeneous Multicore Architectures, In Proceedings of the European Conference on Computer Systems (EuroSys), 2010.

[46] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2003.

[47] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi and K. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2004.

[48] R. Kumar, D. Tullsen and N. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), April 2006.

[49] P. J. M. Laarhoven and E. H. L. Aarts. Simulated annealing: theory and applications, Springer, 1987.

[50] B. C. Lee, and D. Brooks. Illustrative design space studies with microarchitectural regression models, In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), February 2007.

[51] B. C. Lee and D. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2006.

[52] AJ Martin and M Nystroem. ET2: A Metric For Time and Energy Efficiency of Computation, In Power-Aware Computing, 2001.

[53] O. Mutlu, J. Stark, C. Wilkerson and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-ofOrder Processors, In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), February 2003.

[54] R. Nagarajan, X. Chen, R. G. McDonald, D. Burger and S. W. Keckler. Critical path analysis of the TRIPS architecture, In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), September 2006.

[55] H. H. Najaf-abadi and E. Rotenberg. Architectural Contesting, In Proceedings of the International Conference on High-Performance Computer Architecture (HPCA), February 2009.

[56] H. H. Najaf-abadi and E. Rotenberg. Configurational Workload Characterization, In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2008.

[57] S. Navada, N. K. Choudhary and E. Rotenberg. Criticality-driven Superscalar Design Space Exploration, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2010.

[58] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2001.

[59] G. Patsilaras, N. K. Choudhary and J. Tuck. Efficiently Exploiting Memory Level Parallelism on Asymmetric Multicore Processors in the Dark Silicon Era, In Proceedings of the ACM Transactions on Architecture and Code Optimization (TACO), Special issue on High-Performance and Embedded Architectures and Compilers (HiPEAC), 2012.

[60] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. E. Smith. Trace Processors, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 1997.

[61] E. Rich and K. Knight. Artificial Intelligence, 2nd Edition. Morgan Kaufmann, 1991.

[62] R. Rodrigues, A. Annamala, I. Koren, S. Kundu and O. Khan. Performance per Watt Benefits of Dynamic Core Morphing in Asymmetric Multicores, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2011.

[63] J. Saez, M. Prieto, A. Fedorova and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Processors, In Proceedings of EuroSys, 2010.

[64] A. Saidi, N. Binkert, T. N. Mudge and S. K. Reinhardt. Full System Critical Path Analysis, In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2008.

[65] A. Saidi, N. Binkert, S. K. Reinhardt and T. N. Mudge. End-To-End Performance Forecasting: Finding Bottlenecks before They Happen, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2009.

[66] P. Salverda and C. Zilles. A Criticality Analysis of Clustering in Superscalar Processors, In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2005.

[67] L. Sawalha, S. Wolff, M. P. Tull and R. D. Barnes. Phase-guided scheduling on Single-ISA Heterogeneous Multicore Processors, In Proceedings of the Euromicro Conference on Digital System Design, 2011.

[68] D. Shelepov, J. C. Saez, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov and V. Kumar. HASS: a scheduler for heterogeneous multicore systems, In ACM Operating System Review 43, 2 (2009).

[69] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns, In Workshop on Power Aware Computing and Systems, October 2010.

[70] T. Sherwood, E. Perelman, G. Hamerly and B. Calder. Automatically Characterizing Large Scale Program Behavior, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2002.

[71] P. Shivakumar and N.P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model, Technical report, 2001.

[72] T. Sondag and H. Rajan. Phase-based Tuning for Better Utilization of Performance-Asymmetric Multicore Processors, In Proceedings of the International Symposium on Code Generation and Optimization (CGO), April 2011.

[73] The Standard Performance Evaluation Corporation. *Welcome to SPEC*, http://www.specbench.org/.

[74] S. Subramaniam, A. Bracy, H. Wang and G.. Loh. Criticality-Based Optimizations for Efficient Load Processing, In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), February 2009.

[75] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi and M. Upton. Continual Flow Pipelines, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2004.

[76] M. Aater Suleman, O. Mutlu, M. K. Qureshi and Y. N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures, In the Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS), March 2009.

[77] E. Tune, D. Tullsen and B. Calder. Quantifying Instruction Criticality, In the Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2002.

[78] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryskin, J. Lugo-Martinez, S. Swanson and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2010.

[79] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, M. B. Taylor and S. Swanson. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores**,** In Proceedings of the International Symposium on Microarchitecture (MICRO), December 2011.

[80] J. Winter, D.H. Albonesi and C.A. Shoemaker. Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-Core Architectures, In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2010.

[81] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling, In Proceedings of the International Symposium on Computer Architecture (ISCA), June 2003.

[82] J. Yi, D. J. Lilja and D. M. Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology, In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), February 2003.