Copyright

by

Shiwen Hu

2005

The Dissertation Committee for Shiwen Hu Certifies that this is the approved version of the following dissertation:

Efficient Adaptation of Multiple Microprocessor Resources for Energy Reduction Using Dynamic Optimization

Committee:

Lizy K. John, Supervisor

Tony Ambler

Mauricio Breternitz Jr.

Stephen W. Keckler

Kathryn S. McKinley

David Z. Pan

Efficient Adaptation of Multiple Microprocessor Resources for Energy Reduction Using Dynamic Optimization

by

Shiwen Hu, B.E.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2005

Dedicated

To my wife Yi Lu.

Acknowledgements

First I would like to thank my advisor, Dr. Lizy K. John, for her support, advice, guidance and good wishes. Her availability at all times including on weekends, and her dedication to work has had a profound influence not only on my academic pursuit, but also on my life. I am also grateful for the freedom and the flexibility she gave me throughout my Ph.D. study.

My gratitude goes to the committee members (in alphabetical order), Dr. Tony Ambler, Dr. Mauricio Breternitz Jr., Dr. Steve Keckler, Dr. Kathryn S. McKinley, and Dr. David Z. Pan, for their invaluable comments, productive suggestions, and the time to read the draft of my thesis.

I would like to thank the students (past and current) at the Laboratory for Computer Architecture -- Tao, Madhavi, Juan, Ravi, Yue, Byeong, Aashish, Ajay, Sean, Hari, and Dimitry. They provided valuable feedback on the drafts of my paper submissions and on my practical talks. I enjoyed working with Ravi and Madhavi in writing the research papers.

Thanks to Debi, Amy, Melanie and other administrative assistants who worked in the Department in the past years.

Last but not least, I am grateful to my wife, Yi, for her consistent love, support, and encouragement throughout my graduate years. This is not something I could accomplish alone.

Efficient Adaptation of Multiple Microprocessor Resources for Energy Reduction Using Dynamic Optimization

Publication No._____________

Shiwen Hu, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Lizy K. John

The continuing advances in VLSI technology have fueled dramatic performance gains for general-purpose microprocessor, but microprocessor energy consumption has been increasing substantially in the past decade. The steady increase of microprocessor energy consumption significantly affects circuit reliability, cooling and package costs, and battery life of embedded systems.

Adaptive microarchitectures are one of the commonly used techniques to dynamically identify configurations that are desirable from performance and power perspectives. By matching hardware resources to a program's runtime requirements, adaptive microarchitectures can effectively reduce energy with minimal performance loss. However, the task of searching for the most energy efficient configurations is complicated by configuration space explosion, which may considerably impair an adaptive microarchitecture's performance and energy efficiency.

This dissertation presents a hardware adaptation framework for efficient management of multiple configurable units, utilizing a dynamic optimization system's inherent capabilities of detecting and optimizing dominant code regions (*hot spots*). The

framework uses hot spot boundaries for phase detection and hardware adaptation. Since hot spots are of variable sizes and are often nested, the framework can decouple the reconfiguration of CUs with diverse adaptation costs by adjusting the granularity of adaptation based on each CU's reconfiguration cost.

This dissertation also studies the interference imposed by one CU's configuration changes on others' adaptation. CUs with minimal mutual interference can be adapted in parallel. In addition, for some CUs, one's size reduction usually prompts the other to choose a smaller size for energy reduction. Hence, the search of those CUs' best configurations biases toward certain paths, and thus prunes the tuning space. Employing the tuning-reduction strategies, the proposed framework significantly improves the energy efficiency of an adaptive microarchitecture.

The energy and hardware adaptation impact of two important dynamic optimization services, JIT optimization and garbage collection, are also investigated in this work. By stressing the data caches, both dynamic optimization services decrease the average power dissipated by a dynamic optimization system. Furthermore, owing to their distinct runtime characteristics and their capabilities to alter program runtime behavior, the two dynamic optimization services change the adaptation preferences of configurable hardware units, and influence the energy efficiency of an adaptive microarchitecture.

Table of Contents

List of Tables xi

List of Figures xiii

Chapter 1. Introduction 1

1.1 Microprocessor energy reduction 1

1.2 Adaptive microarchitectures 2

1.3 Efficient management of multiple configurable units 3

1.4 Thesis statement 5

1.5 Contributions 5

1.6 Organization 10

Chapter 2. Background and Related Work 11

2.1 Configurable hardware units 11

2.1.1 Issue queue 12

2.1.2 Reorder buffer 13

2.1.3 Cache hierarchy 14

2.1.4 Other hardware units 16

2.2 Program phase detection 17

2.3 Resource adaptation strategies 21

2.4 Dynamic optimization systems 22

2.4.1 Jikes Research Virtual Machine (RVM) 23

2.4.2 Other dynamic optimization systems 24

2.4.3 Hot spot detection and optimization 26

2.5 Power-aware compiler optimizations 27

Chapter 3. Hardware Adaptation Framework based on Dynamic Optimization ...29

3.1 Hot spot detection 30

3.2 CU decoupling and hot spot tuning 31

3.2.1 CU decoupling 31

3.2.2 Hot spot tuning 32

3.3 Hardware reconfiguration 33
3.4 Hardware support..33
3.5 Differences with prior approaches...35
3.6 Applicability 36

Chapter 4. Experimental Methodology..38
4.1 Simulation environment...38
4.1.1 Dynamic Simplescalar 38
4.1.2 Basic block vector phase detection 39
4.1.3 Configurable hardware units..40
4.2 Dynamic optimization system...42
4.3 Benchmarks...43

Chapter 5. Evaluation of the Hardware Adaptation Framework 44
5.1 Runtime characteristics of hot spots and BBV phases...........................45
5.1.1 Runtime characteristics of hot spots 45
5.1.2 Runtime characteristics of BBV phases....................................47
5.2 Adaptation of five configurable units 50
5.2.1 Runtime characteristics..51
5.2.2 Energy reduction..52
5.2.3 Performance impact 55
5.3 Adaptation efficiency..57
5.3.1 Adaptation of issue queue and reorder buffer............................58
5.3.2 Adaptation of L1D, L1I, and L2 caches 60
5.3.3 Adaptation efficiency...61
5.3 Summary of advantages...63
5.4 Discussion...64

Chapter 6. Analysis of Factors that Interfere with Adaptation 67
6.1 Impact of runtime characteristics on CUs' tuning decisions 67
6.1.1 Impact measurement 67
6.1.2 Impact intensity analysis..68
6.1.3 Implications to tuning process reduction 73

6.2 Interference between configurable units 75
6.2.1 Interference distance measurement 75
6.2.2 Interference distances between CUs 76
6.2.3 Implications to tuning process reduction 81
6.3 Reduction of multi-CU tuning process 82
6.3.1 Tuning reduction strategies 83
6.3.2 Evaluation of the tuning reduction strategies 85
6.4 Discussion 89
6.4.1 Generalized tuning reduction algorithm for independent CUs 90
6.4.2 Generalized tuning reduction algorithm for CUs with positive interference 93

Chapter 7. The Role of JIT Optimization and Garbage Collection on Microprocessor Energy Consumption and Hardware Adaptation 94
7.1 Energy impact of JIT optimization and garbage collection 94
7.1.1 Impact of JIT optimization 95
7.1.2 Impact of garbage collection 101
7.1.3 Key insights 108
7.2 Interference of JIT optimization and garbage collection on hardware adaptation 108
7.2.1 Interference of JIT optimization 109
7.2.2 Interference of garbage collection 112
7.3 Discussion 117

Chapter 8. Conclusions and Future Research 119
8.1 Conclusions 119
8.2 Directions for Future Research 123

Bibliography 126

Vita 134

List of Tables

Table 1. Baseline configuration of the simulated system. 38

Table 2. Configurable hardware units and their adaptation parameters. 40

Table 3. Characteristics of SPECjvm 98 benchmarks executed by Jikes RVM. ...43

Table 4. Runtime hot spot characteristics of SPECjvm 98 benchmarks...............45

Table 5. Runtime characteristics of hot spots in three different size ranges..........46

Table 6. Runtime characteristics of stable BBV phases with 10K and 1M sampling intervals. 49

Table 7. Runtime characteristics while adapting the five configurable units. 52

Table 8. Adaptation efficiency retained by the 5-CU system versus the 2/3-CU system. 62

Table 9. Coefficients of variations for the characteristic/CU pairs that possess the monotonic property. 74

Table 10. Performance, energy and power of JIT optimized system as fractions of the corresponding results of un-optimized system. 97

Table 11. Runtime characteristics of JIT optimized system as fractions of the corresponding results of un-optimized system. 99

Table 12. Comparison of JIT optimizer and application's energy consumption and power dissipation. 100

Table 13. Number of garbage collections and changes of cache misses due to garbage collection. 105

Table 14. Comparison of garbage collector and mutator's energy consumption and power dissipation. 107

Table 15. Hardware units' energy consumption in JIT optimized system as fractions of those of un-optimized system. 109

Table 16. Fixed size hardware units' energy consumption with small heaps as fractions of those with 200M heap..113

List of Figures

Figure 1. Adaptation of two four-configuration hardware units. (Each pair of numbers is one combinational configuration, with the numbers indicating the two CUs' individual configurations.) 4

Figure 2. Flowchart of the proposed hardware adaptation framework based on a dynamic optimization system. 30

Figure 3. Distribution of stable/transitional BBV phases of SPECjvm 98 benchmarks. (A phase is stable if it has two or more successive sampling intervals; otherwise it is an unstable phase)..48

Figure 4. Reduction in issue queue energy consumption by adapting the five configurable units (5% performance loss threshold). 53

Figure 5. Reduction in reorder buffer energy consumption by adapting the five configurable units (5% performance loss threshold). 54

Figure 6. Reduction in level-one data cache energy consumption by adapting the five configurable units (5% performance loss threshold). 54

Figure 7. Reduction in level-one instruction cache energy consumption by adapting the five configurable units (5% performance loss threshold). 54

Figure 8. Reduction in level-two cache energy consumption by adapting the five configurable units (5% performance loss threshold). 55

Figure 9. Performance degradation due to the adaptation of the five configurable units. 55

Figure 10. Decomposition of the DO-based approach's IPC degradation.............56

Figure 11. Reduction in energy consumption by adapting issue queue and reorder buffer (2% performance loss threshold). 59

Figure 12. Performance degradation due to the adaptation of issue queue and reorder buffer. 59
Figure 13. Reduction in energy consumption by adapting level-one and level-two caches (3% performance loss threshold)....60
Figure 14. Performance degradation due to the adaptation of level-one and level-two caches. 61
Figure 15. Runtime characteristics' impact on reorder buffer's adaptation. (1: 32-entry ROB; 2: 64-entry ROB; 3: 96-entry ROB; 4: 128-entry ROB)70
Figure 16. Runtime characteristics' impact on issue queue's adaptation. (1: 32-entry IQ; 2: 64-entry IQ; 3: 96-entry IQ; 4: 128-entry IQ)....70
Figure 17. Runtime characteristics' impact on L1D cache's adaptation. (1: 16K L1D cache; 2: 32K L1D cache; 3: 48K L1D cache; 4: 64K L1D cache) .71
Figure 18. Runtime characteristics' impact on L1I cache's adaptation. (1: 16K L1I cache; 2: 32K L1I cache; 3: 48K L1I cache; 4: 64K L1I cache) 72
Figure 19. Runtime characteristics' impact on L2 cache's adaptation. (1: 256K L2 cache; 2: 512K L2 cache; 3: 768K L2 cache; 4: 1M L2 cache) 72
Figure 20. Hardware units' interference on reorder buffer's adaptation. 77
Figure 21. Normalized ROB occupancy. 78
Figure 22. Hardware units' interference on issue queue's adaptation. 79
Figure 23. Hardware units' interference on L1D cache's adaptation. 80
Figure 24. Hardware units' interference on L1I cache's adaptation. 80
Figure 25. Hardware units' interference on L2 cache's adaptation. 81
Figure 26. The tuning reduction strategies. 84
Figure 27. Reduction in issue queue energy reduction by using the tuning reduction strategies. 86

Figure 28. Reduction in reorder buffer energy reduction by using the tuning reduction strategies. 86

Figure 29. Reduction in L1D cache energy reduction by using the tuning reduction strategies. 87

Figure 30. Impact of the tuning-reduction strategies on L1I cache energy reduction. 87

Figure 31. Impact of the tuning-reduction strategies on L2 cache energy reduction. 87

Figure 32. Performance degradation due to the tuning reduction strategies..........88

Figure 33. Generalized tuning reduction algorithms for independent CUs...........91

Figure 34. Generalized tuning reduction algorithms for CUs with positive interference. 92

Figure 35. Changes in performance, energy, and power due to garbage collection.103

Figure 36. Impact of JIT optimization on configurable unit energy reduction....110

Figure 37. Comparison of JIT optimizer and application on configurable unit energy reduction. 111

Figure 38. Fixed size hardware units' energy consumption with small heaps as fractions of those with 200M heap. (The results are the same as the ones in the last column of Table 16). 113

Figure 39. Impact of garbage collection and hardware adaptation on configurable unit energy consumption..114

Figure 40. Impact of garbage collection on configurable unit energy reduction.115

Figure 41. Comparison of garbage collector and mutator on configurable unit energy reduction. 117

Figure 42. Percentage of a microprocessor's energy reduced by using the hardware adaptation framework to manage the five configurable units.........120

Chapter 1. Introduction

1.1 MICROPROCESSOR ENERGY REDUCTION

In the past several decades, advances in VLSI technologies and microarchitectural innovations have driven the continuous performance improvement of microprocessors. Today's high performance processors integrate millions of transistors and operate at Giga Hertz frequency. Despite performance achievement, processor architecture designs still face challenges. Specifically, the increasing power dissipation is one of the biggest concerns. Microprocessor power densities have been increasing substantially from one generation to the next, despite the reduced supply voltages and advanced processing technologies. High power densities of today's state-of-the art processors significantly affect circuit reliability, cooling issues, and package costs.

Meanwhile, the market for portable computation devices is growing quickly. In most embedded devices, the computing element accounts for a high portion of the overall energy consumption. Most embedded systems use batteries whose life is determined by the energy consumption of the embedded systems. Consequently, the interest in lowering the power and energy of a processor has grown dramatically over the past several years.

Various techniques are proposed to deal with this problem. First, low power VLSI devices and logic are designed to make significant contribution to power saving. In the circuit level, voltage scaling, low swing buses, and conditional clocking help reduce energy consumption enormously [34][85]. Power-aware compiler optimizations can also help alleviate the energy problem [32][77][83].

1.2 ADAPTIVE MICROARCHITECTURES

Adaptive microarchitectures are one of the commonly used techniques to dynamically identify configurations that are desirable from performance and power perspectives. By matching hardware configurations with changing program requirements, adaptive microarchitectures can reduce energy with minimal performance impact. With fixed size microarchitectural structures such as wide instruction window and large L1 caches, conventional microprocessors are designed to maximize performance for a range of applications. However, a program rarely fully utilizes every microarchitectural resource to achieve high performance. Hence, it is possible to reduce the sizes of those under-utilized microarchitectural resources for energy reduction with minimal performance impact. To achieve energy reduction, the hardware units must have multiple configurations that can be changed at runtime. Reconfiguration of many such *configurable units* (CUs), such as issue queue, reorder buffer, caches, and branch predictors, have been proposed [1][7][27][36][63]. In this dissertation, the term *adaptive microarchitecture* indicates a microprocessor design with one or more such configurable hardware units.

Changing a hardware unit's setting at runtime usually incurs cycle-time *reconfiguration overhead*. For instance, to reduce a cache's size, dirty cache lines must be written back to lower memory hierarchy, which may take thousands of cycles [19]. Hence, a program's performance may be impaired by too frequent reconfigurations where reconfiguration exceeds the benefit gained by the reconfigurations. After a reconfiguration, the configuration should be utilized for a certain minimum time interval, called the CU's *reconfiguration interval*. Depending on its reconfiguration overhead, a configurable unit's reconfiguration interval can vary from thousands (e.g., issue queue [27]) to millions (e.g., caches [2]) of instructions/cycles.

Two important issues in hardware adaptation are *when* to adapt hardware resources and *which* configuration to adapt to. The first issue is addressed by program phase detection. By the definition of Sherwood et al. [75], a *phase* is a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency. Recent research in program runtime behavior confirms that programs execute as a series of phases with varying runtime characteristics and hardware requirements, while still having a fairly homogeneous behavior within a phase [19][20][75][76]. Phase boundaries are thus suitable points for resource reconfiguration. Previously proposed resource adaptation approaches rely on diverse hardware and software schemes to detect distinct program phases that are associated with either successive program sampling intervals or code positions.

The second issue of hardware adaptation, i.e., *which* configuration to adapt to, is addressed by the tuning strategy. Most prior hardware adaptation schemes use the same tuning strategy. Upon a phase change, this tuning strategy tests all hardware configurations and adapts to the *best* one (i.e., the most energy-efficient one that satisfies the performance constraint), and this procedure is called the phase's *tuning process.* The time taken by the tuning process is the phase's *tuning latency,* which is proportional to the number of configurations tested. For instance, the latency of tuning a four-configuration CU is four reconfiguration intervals, with each interval testing one configuration. On the other hand, in a multiple CU environment, the tuning process tests those CUs' combinational configurations.

1.3 EFFICIENT MANAGEMENT OF MULTIPLE CONFIGURABLE UNITS

In adaptive microarchitectures, efficient management of configurable units is vital for maximizing energy reduction. To achieve high energy reduction, an adaptive microarchitecture usually has multiple configurable units. Intuitively, in a multi-CU

environment, changing one CU's configuration may affect the tuning decision of another CU. Hence, hardware units can hardly be adapted individually. This research is motivated by the importance on the efficient adaptation of multiple configurable units on overall energy reduction.

In a multi-CU environment, the straightforward tuning strategy of testing all combinatorial configurations results in a long tuning process. With this tuning strategy, the tuning latency is proportional to the total number of combinatorial configurations, which increases dramatically as more hardware units become configurable. For instance, a four-configuration CU needs only four tests, while adapting five four-configuration CUs needs 1024 tests. Figure 1 illustrates the tuning process of two four-configuration hardware units. Note that it is possible to reduce the tuning process by ruling out some unpromising configurations found via offline or online profiling.

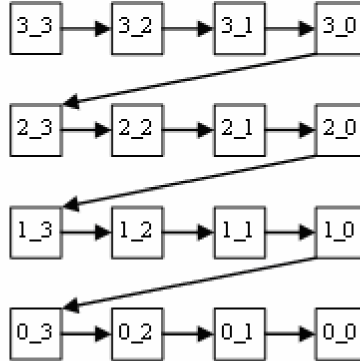

Figure 1. Adaptation of two four-configuration hardware units. (Each pair of numbers is one combinational configuration, with the numbers indicating the two CUs' individual configurations.)

With multiple configurable units, using the straightforward tuning strategy of testing all combinatorial configurations significantly affects the efficiency of hardware adaptation. First, during the tuning process, the program mostly executes at

configurations that yield less energy reduction or too much performance loss compared to the best one. Hence, the rapidly increased tuning latency impairs performance and energy reduction substantially. The performance is further impaired by the increasing tuning overhead that is proportional to the tuning latency. Finally, using the conventional tuning strategy implies that all CUs are adapted at the same pace, which must be chosen to accommodate the largest CU reconfiguration interval. Thus, reconfiguration opportunities are lost for low-overhead CUs. Consequently, it is imperative to find new tuning strategies that improve the adaptation of multiple CUs for better energy reduction.

1.4 Thesis Statement

In adaptive microarchitectures, efficient management of the configurable units is vital for maximizing energy reduction. Utilizing a dynamic optimization system's inherent capabilities of detecting and optimizing hot spots, a hardware adaptation framework can efficiently manage multiple configurable units by adapting them at hot spot boundaries, and thus achieve much better energy reduction than conventional approaches.

1.5 Contributions

This research makes multiple contributions to efficient adaptation of multiple configurable units for energy reduction.

The first contribution of this dissertation is that it demonstrates how a dynamic optimization (DO) system' inherent capabilities of detecting and optimizing hot spots can be synergistically employed for efficient management of adaptive microarchitectures. During the past decade, dynamic optimization systems have grown in popularity. *Dynamic optimization* is a software system's ability to dynamically translate/optimize one type of program code to another form, even in the same ISA. Examples of DO

systems include Transmeta CMS [18], IBM DAISY [24], HP Dynamo [8], Intel IA32EL [10], Java virtual machines [5][90], and Microsoft .NET's common language runtime [91]. To amortize the overhead of runtime translation and further improve performance, most DO systems apply high-cost high-quality optimizations only on frequently executed code sequences (hot spots). It has been shown that hot spots usually have stable runtime characteristics throughout program execution [84], and closely represent program behavior changes [36][56]. Therefore, DO systems are good platforms for adaptive computing environment management.

This dissertation presents a hardware adaptation framework for efficient management of multiple configurable units based on a generic dynamic optimization system. Exploiting the existing hot spot detection mechanism of the dynamic optimization system, the proposed framework adapts microarchitectural resources at hot spot boundaries.

The framework prunes the explosive tuning space of multiple CUs with two techniques. First, a technique, CU decoupling, decouples the adaptation of CUs with diverse adaptation cost. Program hot spots are usually of variable sizes and are nested. Smaller hot spots represent fine-grain phases nested within coarse-grain phases that appear as large hot spots. Intuitively, they closely represent hierarchical phase behavior. Thus, this framework automatically captures the hierarchical phase behavior by identifying nested hot spots. Utilizing this capability, the framework decouples the reconfiguration of different CUs by adjusting the granularity of adaptation based on each CU's reconfiguration cost. This *CU decoupling* strategy significantly reduces the tuning process, and achieves a better balance of benefit/overhead for each configurable hardware resource.

The second technique exploits the different types of interference between CUs' adaptation to prune the tuning space of multiple CUs. In a multi-CU environment, the interference between two CUs exists when one CU's configuration changes alter another CUs' adaptation decisions. For instance, some CUs, such as L1D and L1I caches, rarely interfere with each other, so they can be adapted in parallel. Furthermore, for CUs such as issue queue and reorder buffer, one's size reduction usually prompts the other to keep the same configuration or use a smaller one for energy reduction with minimal extra performance loss. With those types of interference, the search of the CUs' best configuration usually biases towards certain paths, and the tuning space can be pruned by testing only configurations in those paths. Differing with CU decoupling, those tuning reduction strategies are also effective for CUs with similar adaptation costs. Employing the two techniques, the framework manage multiple configurable units efficiently and achieve high energy reduction.

To demonstrate the usefulness and effectiveness of our framework, the proposed hardware adaptation framework is implemented using Jikes Research Virtual Machine [5] and Dynamic Simplescalar simulator [38]. Performance is evaluated on the SPECjvm98 benchmark suite [88]. Five configurable units are implemented: issue queue, reorder buffer, L1 instruction and data caches, and L2 cache. Previous research indicates that those five components dominate the energy consumption of a microprocessor [27]. Hence, improving those CUs' adaptation should considerably reduce the overall microprocessor energy consumption.

Another contribution of the work is the comprehensive investigation of the major adaptation-interfering factors in a multi-CU system, and the findings are applied to the design of new tuning-reduction strategies. The ultimate goal of a resource adaptation scheme is to achieve the lowest energy consumption and the best possible performance. If

energy is the primary constraint, some performance losses may have to be tolerated. Similarly, if performance is the absolute goal, one may have to tolerate high power and energy. Abiding by program requirements and the performance budget, CUs' configurations are adjusted via their tuning processes to achieve the maximum overall energy saving. Hence, in a multi-CU environment, program's runtime characteristics/requirements and the interference imposed by other CUs' adaptation are the two major factors that affect a CU's tuning decision. In this research, those two adaptation-interfering factors are investigated. Following are several findings.

First, by qualitatively analyzing CUs' performance impact, two interference types between CUs are discovered: a) *positive*: one CU's size reduction relieves the performance pressure on the other one, and allows it to use the same or a smaller size; b) *negative*: one CU's size reduction increases the performance pressure on the other one, and prompts it to use a larger size. The interference between issue queue and reorder buffer are positive, which leads to the design of a tuning-reduction strategy for the two units.

Second, the interference of one CU's adaptation on another CU's tuning decisions varies by CUs. In general, high-overhead CUs are less sensitive to other units' interference than low-overhead CUs, which validates the efficiency of multi-grain adaptation employed by the DO-based hardware adaptation framework. There is extensive interference between issue queue and reorder buffer, while L1D and L1I caches have minimal mutual interference. Hence, the L1D and L1I caches can be adapted in parallel, disregarding each other's tuning.

The interference analysis helps us design two tuning-reduction strategies. The first strategy reduces the latency of tuning L1D and L1I caches by adapting them in parallel. Exploiting the positive interference between the issue queue and the reorder

buffer, the second strategy reduces the tuning latency of those two CUs by testing only configurations that comply with the positive interference property. The CU decoupling technique decouples only the adaptation of CUs with diverse reconfiguration overheads, while those two tuning-reduction strategies reduce the adaptation latency of CUs with similar reconfiguration overheads. Hence, employing the tuning reduction strategies and the CU decoupling technique, the proposed framework significantly improves the energy efficiency of an adaptive microarchitecture.

Another contribution of this dissertation is the extensive study of the energy and power impact of two important dynamic optimization services, JIT optimization and garbage collection, on microprocessor energy consumption and hardware adaptation. By reducing instruction counts, JIT optimization significantly reduces a program's energy consumption, while garbage collection incurs runtime overhead that consumes more energy. Interestingly, both JIT optimization and garbage collection decrease the average power dissipated by a program. Detailed analysis reveals that both JIT optimizer and JIT optimized code dissipate less power than un-optimized code. On the other hand, the garbage collector dissipates less power than the mutator, but it rarely affects the mutator's average power.

This research also reveals that DO systems interfere with hardware adaptation. Both JIT optimization and garbage collection alter programs' behavior and runtime requirements. In adaptive microarchitectures, such changes of runtime requirements can considerably affect the adaptation of configurable hardware units, and eventually influence the overall energy consumption. The adaptation preferences of configurable units on the JIT optimizer and the garbage collector are also studied, demonstrating that the units' adaptation decisions on the application code and the dynamic optimization

services can differ substantially. The insights gained in this research point to novel techniques that can further reduce microprocessor energy consumption.

1.6 Organization

The remainder of the dissertation is organized as follows:

Chapter 2 presents the background and related work on hardware resource adaptation, dynamic optimization, and energy and power impact of compiler optimizations. Chapter 3 introduces the proposed hardware adaptation framework. Chapter 4 discusses the experimental methodology used in this research. Chapter 5 evaluates the framework and compares it against one of the best prior resource adaptation schemes on managing multiple configurable units. Chapter 6 examines how runtime program requirements, in terms of measurable characteristics, affect CUs' adaptation decisions. Interference imposed by the adaptation of CUs on other CUs' tuning decisions is also studied in this chapter. Dynamic optimization and garbage collection are two of the major services in many dynamic optimization systems that assist program execution. Chapter 7 analyzes the impact of those two dynamic optimization services on microprocessor energy consumption and hardware adaptation. Chapter 8 concludes the dissertation by summarizing the contributions and suggesting future directions.

Chapter 2. Background and Related Work

This chapter presents the background and the related work on hardware adaptation, dynamic optimization systems, and compiler optimizations for microprocessor energy reduction. First, various configurable units and the terms reconfiguration overheads and intervals of CUs that are used throughout the dissertation are introduced. Most resource adaptation schemes have two components: a phase detection mechanism that identifies when to adapt hardware resources, and a tuning strategy to identify which units to configure and how to configure the units. Section 2.2 describes previously proposed phase detection mechanisms, and Section 2.3 introduces the commonly used tuning strategy and identifies its key limitations on managing multiple configurable units. Several exemplary dynamic optimization systems are presented in Section 2.4. All of those dynamic optimizations systems detect and optimize hot spots to minimize dynamic optimization cost. The impact of compiler optimizations on microprocessor power dissipation and energy reduction is discussed in Section 2.5.

2.1 Configurable Hardware Units

Configurable hardware units are proposed to reduce energy consumed by those units with minimal performance impact. Changing a hardware unit's setting at runtime usually incurs cycle-time *reconfiguration overhead*. For instance, to reduce a cache's size, dirty cache lines must be written back to lower memory hierarchy, which may take thousands of cycles [19]. Hence, a program's performance may be impaired by too frequent reconfigurations where reconfiguration overhead overcomes the benefit gained by the reconfigurations. After a reconfiguration, the configuration should be utilized for a certain minimum time interval, called the CU's *reconfiguration interval*. Depending on its reconfiguration overhead, a configurable unit's reconfiguration interval can vary from

thousands (e.g., issue queue [27]) to millions (e.g., caches [2]) of instructions or cycles. The following subsections studies the following configurable units: issue queue, reorder buffer, caches, branch predictors, and filter cache etc.

2.1.1 Issue queue

In out-of-order superscalar microprocessors, multiple instructions are dynamically reordered and issued to functional units in their data ready order rather than their original program order to achieve high performance. Two hardware units that facilitate dynamic instruction scheduling are issue queue and reorder buffer. Due to their highly associative nature, they are among the most power-hungry and time-critical components in a modern processor. Hence, numerous previous efforts focus on reducing the energy consumption of issue queue and reorder buffer by adjusting their sizes dynamically, or by utilizing low-complexity, low-power alternative ones when suitable.

Many issue queue energy reduction schemes dynamically adjust issue queue sizes to match program requirements for energy saving. To estimate program computational demand and to guide adaptation, they usually sample measurable metrics such as IPC or issue queue occupancy. Folegnani et al. [27] propose an energy effective issue logic design, which saves energy by reducing the issue queue size dynamically, as well as avoiding waking up empty issue queue entries and ready operands. Bahar et al. [6] adapt multiple processor resources, including the issue queue, by balancing the pipeline width with program requirements at runtime. Similar designs are discussed in several other works [1][63][65] as well.

Several other schemes reduce the energy consumption of issue queue by providing low complexity, low power alternatives. Those issue queues can be used solely, or combined with the conventional out-of-order, high power consumption issue queue for better energy saving. Using the low-power issue queues when program has less

available ILP can save energy with minimal or even beneficial performance impact. Canal et al. [15] propose two low-complexity issue queues. The first issue queue design, first-use issue scheme, replaces the conventional out-of-order issue queue with several in-order queues by utilizing the fact that most register values are read at most once. The second scheme, distant issue logic, tries to determine instruction execution order in the decode stage, thus simplifying complex dynamic scheduler used in conventional issue logic. It provides extra hardware to hold instructions that wait for memory accesses. Abella et al. [3] present a low-complexity distributed issue queue design that classifies instructions and dispatches them into a set of FIFO queues depending on their data dependences.

Several other efforts try to simplify the issue queue design by using a less aggressive hardware instruction scheduler. Ernst et al. [25] implement the simple hardware Cyclone scheduler that schedules instructions by predicting the ready latencies of the instructions' operands. The Cyclone instruction queue schedules predicted instructions via a network of locally synchronized datapaths. To ensure correct program execution in the presence of latency and dependence speculation, the Cyclone scheduler incorporates a selective replay mechanism that overloads the register forwarding infrastructure to re-execute only those instructions dependent on incorrectly scheduled instructions. A similar scheme is proposed by Michaud et al. [58].

2.1.2 Reorder buffer

As one of the key datapath structures in modern superscalar, out-of-order processors, the reorder buffer (ROB) maintains the program order and precise states of instructions when them execute out of order. In some microarchitectures, such as the Intel P6, the physical registers are implemented as slots within the ROB entries, which, combined with the large number of ROB ports, contribute a significant percentage to the

overall chip power dissipation [27]. Hence, it is important to reduce the ROB power consumption with minimal performance loss. Several prior efforts in the area propose to dynamically adjust the size of ROB for better energy efficiency [6][42]. Besides the reorder buffer, Wu et al. [83] also adjust the fetch width of the microprocessor dynamically.

Kucuk et al. [47] exploit the observation that in a typical superscalar datapath, almost all of the source operand values are obtained either through data forwarding of the recently generated results or from the reads of the committed register values. Only a small percentage of the operands need to be read from the ROB; delaying such reads have little impact on the overall performance. This allows the ROB ports for reading the source operands at the time of dispatch to be completely eliminated with little performance loss. Another paper [46] extends the scheme by implementing a ROB with distributed queues. Each ROB queue serves a single functional unit, with a single write port instead of multiple ones needed by a centralized implementation.

2.1.3 Cache hierarchy

Due to their large sizes, on-chip caches contribute much to a microprocessor's power dissipation and energy consumption. Various adaptable cache schemes have been proposed in past research. Some of them disable parts of cache ways or cache lines, or access only the cache ways that are most likely to hit. Cache line width and hierarchy can be adjusted to adapt to program requirements.

Albonesi [2] proposes to a technique, called *selective cache ways*, to reduce cache energy consumption by dynamically disabling a subset of cache ways in a set associative cache. Selective cache ways exploits the fact that large on-chip caches are often partitioned into multiple subarrays to reduce the long word and/or bitline delays of a single large cache array. Hence, only minor changes, e.g., gating hardware and a control

register, are required to implement selective cache ways in a conventional cache. A similar adaptive complexity-adaptive cache hierarchy is proposed [1] as a part of the so-called complexity adaptive processors.

During a cache access, usually both the tag and data arrays are activated. A cache in the phased mode [31] is a set associative cache where an access first activates only the tags. If there is a match, only the matching data array is subsequently activated, reducing the amount of bitline activity and sense amplification in the data array. Consequently, a phased cache saves energy at the cost of extra delay. Min et al. [59] present a low power cache system that compares tags in two phases. The first phase determines the data ways that the memory access falls into, which is then verified by the second phase. With those two phases, most activities in accessing a conventional cache tag can be avoided, and thus achieve energy saving. The approaches proposed by Powell et al. [64] predict the matching ways and probes only the predicted ways and not all the ways to achieve energy saving. A similar approach is proposed by Zhang et al. [86], in which a small way-halting cache stores partial tags and is accessed in parallel with set-index decoding. Thus, the ways with mismatching tags can be disabled during the cache assess for energy saving.

Kaxiras et al. [45] exploit the fact that cache lines are usually not accessed for a long period before they are evicted. Hence, tuning off the cache lines during the dead period can reduce power leakage, without introducing any additional cache misses that hurt performance. The paper proposes several policies for determining when to turn a cache line off. Based on the same principle, the drowsy cache [26] reduces cache leakage by exploiting the fact that during a fixed period of time, only a small subset of cache lines are accessed. Hence, inactive cache lines can transfer to a low-power drowsy mode with minimal performance loss.

Balasubramonian et al. [9] propose a cache hierarchy reconfiguration scheme that behaves as a virtual two-level, physical one-level non-inclusive cache hierarchy. In the scheme, multiple levels of the cache hierarchy share a large cache. The boundaries between L1 and L2 caches, and thus their sizes, are adapted at runtime by monitoring phase changes using miss rates and branch frequencies. The scheme is also applicable to TLBs. Based on the same principle, Ranganathan et al. [68] propose a reconfigurable cache in which a portion of the cache could be used for another function, such as an instruction reuse buffer. Veidenbaum et al. [82] describe an adaptive cache design in which cache line size is dynamically adapted based on program requirements monitored by hardware, although it aims at improving cache miss rate, not reducing energy consumption.

The Filter cache [48] is proposed to reduce the power consumption of an L1 instruction cache. The Filter cache is a small, direct-mapped cache that resides between processor core and the L1 instruction cache. If it is deactivated, instructions are fetched directly from the L1 instruction cache; otherwise, the processor checks the filter cache first. Filter cache hits save energy and improve performance, while filter cache misses consume more energy and lose performance.

2.1.4 Other hardware units

Branch prediction is essential for increasing instruction-level parallelism. However, a large amount of unnecessary work results from wrong-path instructions entering the pipeline due to misprediction. Previous work [61] shows that in general, such speculation cost is worthwhile. Yet, it is still possible to improve the energy efficiency of branch prediction without hurting its accuracy. The work also proposes two techniques, banking and the prediction probe detector, to reduce the power dissipation of branch predictors. Banking allows partial activation of the branch predictor during a predictor

access. The prediction probe detector uses pre-decode bits to entirely eliminate unnecessary predictor and branch target buffer accesses. Manne et al. [53] use confidence estimation to gate the execution of branches that are most likely to be mispredicted, and thus save energy.

Wide-issue processors typically have many functional units (FUs). Since few applications need all the FUs all the time, processors typically clock-gate unused FUs, which can significantly reduce the energy consumed by FUs. However, performance will be impaired when program requires more FUs than available ones. This scheme is used to reduce microprocessor energy consumption [6][36].

Besides turning off whole or part of hardware resources, it is also possible to scale down the voltage and frequency of processor core and DRAM array for energy saving. Dynamic energy is proportional to the square of the supply voltage, while dynamic power is proportional to the frequency and to the square of the voltage. Hence, reducing both the voltage and the frequency reduces energy, and is thus widely used in modern microprocessors [39]. Lowering the voltage of DRAM array can be conducted by changing the reference voltage used in an on-chip voltage converter according to the outputs of a detector [43].

2.2 Program Phase Detection

Recent research in program runtime behavior confirms that programs execute as a series of phases, where each phase may be very different from the others, while still having a fairly homogeneous behavior within a phase [19][20][75][76]. By the definition of Sherwood et al. [75], a *phase* is a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency. Accurate and timely identification of program phases is essential for improving the effectiveness of adaptation. For instance, in systems that contain CUs with large reconfiguration

overheads, recurring phases may reuse configuration information to avoid repeated tunings to improve performance.

Balasubramonian et al. [9] use a conditional branch counter to detect program phase changes. The counter keeps track of the number of dynamic conditional branches executed over a fixed execution interval. Phase changes are detected when the difference in branch counts of consecutive intervals exceeds a threshold. Their scheme does not use a fixed threshold. Rather, the detection strategy dynamically varies the threshold throughout the execution of the program.

Sherwood et al. [76] propose to use an array of hardware counters, called basic block vector (BBV), to capture and classify phase-based program behavior on large time scales. This scheme is an online variant of the offline phase analysis and profiling tool SimPoint [74]. BBVs keep track of execution frequencies of basic blocks touched in a particular execution interval. Phase changes are detected when the Manhattan distance between consecutive BBVs exceeds a preset threshold. The paper demonstrates that sampling intervals of the same phase are homogeneous over multiple performance metrics. Program phases are usually hierarchical, i.e., they are of variable lengths and nested. Lau et al. [49] extend the SimPoint tool to detect hierarchical phases by using k-means clustering to group intervals with similar BBVs into clusters. However, the technique is complex and cannot be performed at runtime. Hence, it is hard to apply it to the online BBV phase detection technique.

Dhodapkar et al. [19] propose to use instruction working set signatures, i.e., highly compressed working set representations, to identify program phase behavior. The rationale is that as program phase behavior changes, the phases' instruction work sets should change also. Hence, by identifying the working set changes, one can detect phase changes and trigger hardware re-tuning. The working set signature is an n-bit vector

formed by mapping addresses of cache line accesses into the vector buckets using a randomizing hash function.

All of the above phase change detection schemes can classify intervals with similar characteristics into phases, and allows hardware reconfiguration techniques to adapt for each phase. This is mainly due to their use of microarchitecture independent phase detection metrics. Dhodapkar et al. [20] compare the three phase detection techniques in terms of sensitivity to phase changes and false positive, phase stability and length. It finds out that the BBV technique performs better than other techniques.

Several other phase detection schemes identify phases using microarchitecture dependent metrics such as IPC [27] and occupancy [63]. One deficiency of such schemes is that the characteristics used to trigger hardware retuning are microarchitecture-dependent, and are affected by configuration changes conducted by the schemes. To minimize such interference, the configurable units have to be adapted to their maximum sizes once in a while.

The above phase detection schemes are all temporal ones, in that phases consist of sampling intervals with fixed or varying sizes. In temporal approaches, phases can be classified into stable and transitional ones [20]. A phase is *stable* if it lasts two or more successive sampling intervals; otherwise, it is *transitional*. Transitional phases have short lifetimes, rarely recur, and are thus difficult to tune. Recognizing only stable phases can improve the phase detection hardware utilization and increase phase detection accuracy considerably [20]. Lau et al. [50] propose to filter out transitional phases also.

In temporal approaches, a phase change cannot be identified immediately. It can only be detected after the phase change lasts one or more sampling intervals, which is called the identification latency. Recurring phases in those temporal approaches incur phase identification latencies, regardless the length of program execution. Recurring

phase identification latencies can be reduced by next phase detection mechanisms [50][76], which predict what the next phase will be and when it will occur. However, incorrect predictions cause unnecessary or wrong adaptation and subsequent rollback of hardware configurations, thus affecting performance considerably. Hence, high prediction accuracy is imperative for such mechanisms. Based on the BBV phase detection technique, Sherwood et al. [76] propose two phase prediction mechanisms, Markov and Run-length-encoding Markov predictors. Lau et al. [50] also examine the issue, and propose several improved phase prediction schemes using hybrid predictors and confidence. Nevertheless, it is hard to accurately predict phase changes. Hence, current complex predictors usually provide marginal benefits over no prediction [50].

Differing from temporal approaches that detect phases at successive sampling intervals, the positional approach [36][83] captures phase changes at certain positions such as procedure boundaries, relying on the observation that program phase behavior is closely associated with program structures. Since it is hard to find procedure calls that start new phases by hardware at runtime, the positional approach simply adapts at boundaries of large procedures [36]. It has been shown that phase detection techniques based on large procedure boundaries do not perform as well as those based on the temporal approaches due to their inability to adapt to changes within the procedures [20].

Recently, Shen et al. [72] propose to construct memory phases based on memory localities. The scheme first profiles program execution using variable-distance sampling, wavelet filtering, and best phase partition via tracing a training input. It then identifies the memory phase hierarchy through grammar analysis. Finally, the phase information is inserted into program code by a static compiler. The technique is mainly used for cache adaptation. Its efficiency on adapting other hardware units is yet to be proven.

Iyer et al. [42] propose a microarchitecture-level power management scheme to detect program phase changes and triggers the adaptation of multiple hardware resources. It uses a hardware profiling scheme to identify tightly coupled regions of code and a hardware-based power estimation method to judge the power requirements for each region of code and scale or resize resources at runtime depending on these estimates. The hardware hot spot detection scheme is based on the work of Merten et al. [56]. However, the limited scope of hot spot detection hardware and its frequent flushes indicate that the scheme can only detect small, locally-hot code regions, and is unable to detect large, global hot spots.

2.3 RESOURCE ADAPTATION STRATEGIES

The tuning strategies used in previous resource adaptation schemes [9][36][37][42] are similar in nature. In general, after a phase is found, the tuning strategy tests different configurations of a CU in successive sampling intervals (the temporal approaches), or successive invocations of the same code (the positional approach). It tests the least energy efficient configuration first. The tuning process completes when all the configurations have been tested. The most energy efficient configuration that satisfies the performance constraint is then selected for the phase. The tuning latency, the time taken to find the most energy-efficient configuration, is the number of sampling intervals required to test all the configurations.

With multiple configurable units, using the straightforward tuning strategy of testing all combinatorial configurations significantly affects the efficiency of hardware adaptation. First, during the tuning process, the program mostly executes at configurations that yield less energy reduction or too much performance loss compared to the best one. Hence, the exponentially increased tuning latency impairs performance and energy reduction substantially. Furthermore, the performance is also impaired by the

increasing tuning overhead that is proportional to the tuning latency. Finally, using the conventional tuning strategy implies that all CUs are adapted at the same pace, which must be chosen to accommodate the largest CU reconfiguration intervals. Reconfiguration opportunities are thus lost for low-overhead CUs. Consequently, resource adaptation schemes always prefer short tuning processes to long ones.

Several prior efforts aim at efficient adaptation of multiple configurable units. Ponomarev et al. [63] propose a scheme that periodically samples occupancies of data path buffers to adapt them independently. Dropsho et al. [21] minimizes the overhead of managing multiple configurable units by using local information, such as a cache's LRU states and a buffer's occupancy statistics, to guide each configurable unit's tuning decision. The paper notes that configurations of interdependent CUs are coupled, and cannot be recognized by local tuning strategy. Hence, a global tuning strategy might exploit the coupled effects for more energy reduction. The two hardware adaptation schemes are extended in a scheme that trades off clock frequencies with hardware structure sizes to achieve high throughput [22]. For multimedia applications, Sasanka et al. [73] propose multiple local hardware adaptation schemes to exploit intra-frame execution variability. Those local schemes are integrated with the global scheme, exploiting per-frame execution slack, to achieve better overall energy reduction.

2.4 DYNAMIC OPTIMIZATION SYSTEMS

During the past decade, dynamic optimization (DO) systems have grown in popularity. Dynamic optimization is a software system's ability to dynamically translate/optimize one type of program code to another form, even in the same ISA. Examples of DO systems include IBM DAISY [24], Transmeta CMS [18], HP Dynamo [8], Intel IA32EL [10], Java virtual machines [5][90], and Microsoft .NET's common language runtime [91]. Some of those systems are full system emulators that emulate a

whole system, and execute under the operating system. The others execute at the user space and above the operating system.

Dynamic optimization systems provide several hardware and software engineering advantages over statically compiled binaries, including flexible hardware design, portable program representations, some safety guarantees, build-in automatic memory and thread management. This section reviews five typical dynamic optimization systems.

2.4.1 Jikes Research Virtual Machine (RVM)

Jikes RVM [5] is a research Java virtual machine (JVM) developed in IBM T. J. Watson Center, and is the first Java virtual machine written in Java. In addition to providing a high-level strongly-typed development environment, this design decision allows the optimization techniques to apply not only to application code, but also to the JVM itself.

Jikes RVM employs a compile-only strategy. It includes two, baseline and optimizing, compilers. The baseline compiler translates bytecodes directly into native code without any optimization. It is also used to validate the optimizing compiler. The optimizing compiler first translates bytecodes into an intermediate representation (IR). It then optimizes the IR before translating it into native code. The optimizing compiler of Jikes RVM has three levels of optimizations (JIT0, JIT1, and JIT2), each one consisting of its own group of optimizations as well as the optimizations that belong to lower levels. The lower two levels (JJT0 and JIT2) perform optimizations that are fast and high-payoff. JIT0 performs inlining and register allocation. JIT1 contains optimizations, such as such as common sub-expression elimination, copy and constant propagation, and dead-code elimination. JIT2 contains more expensive ones, such as those based on static single assignment (SSA) form.

Jikes RVM's adaptive optimization system is responsible for choosing the right compilation level for a given method. It uses a low-overhead sampling method to detect program hot spots. Approximately every 10 milliseconds, Jikes RVM increments a counter associated with the currently active procedure. For all methods that have been sampled, Jikes uses a cost/benefit model to determine whether it is profitable to recompile the method, and if so, what level of optimization to use.

2.4.2 Other dynamic optimization systems

The IBM DAISY system [24] emulates a PowerPC processor and executes on a DAISY VLIW processor. When the system powers up, the DAISY code is executed to initialize itself and the system. Then, it begins PowerPC emulation by interpreting all PowerPC instructions belonging to PowerPC firmware, AIX operating system, or user applications. It uses a counter based scheme to detect hot spots and translate newly detected hot spots into native VLIW code. Hence, when the hot spot is encountered again, its native VLIW code is executed. During the JIT compilation, DAISY performs a variety of optimizations including ILP scheduling with data and control speculation, loop unrolling, alias analysis, load-store telescoping, copy propagation, combining, unification, and limited dead code elimination.

The Transmeta Crusoe [18] emulates an x86 system on a VLIW chip specifically designed to support binary translation for x86. It is similar to IBM DAISY in that they are both full system emulators. However, Crusoe is aimed at low power and mobile applications, with narrow issue width and smaller memory budgets than IBM DAISY. Crusoe maintains a shadow copy of the x86 portion of its register set, and a gated store buffer to efficiently recover from exceptions. Hence, within the scope of a group, instructions can be arbitrarily reordered or deleted. Thus, Crusoe can perform optimizations such as strength reduction and aggressive dead code elimination. In other

ways, Crusoe optimization is similar to that of DAISY. Code is first interpreted and profiled. If a fragment turns out to be frequently executed (more than 50 times), it is translated to native Crusoe instructions.

HP Dynamo [8] is a dynamic optimization system that takes PA-RISC code and produces optimized PA-RISC code. Implemented entirely in software, HP Dynamo runs above the HPUS operating system. Its operation is transparent: no preparatory compiler phase or programmer assistance is required. Dynamo improves execution time on some SPEC benchmarks by up to 22% and by an average of more than 10%. Initially, Dynamo interprets the instruction stream until a hot instruction sequence (trace) is identified. To identify hot traces, Dynamo associates counters with loop headers. If the counter exceeds a preset threshold value, the instruction starts a hot trace. Dynamo records the sequence of instruction as they are being interpreted until an end-of-trace condition is reached. At that point, Dynamo generates an optimized version of the hot trace (fragment) into a software code cache. Subsequent encounters of the hot trace will invoke the optimized fragment in the fragment cache. Due to the significant overheads of operating at runtime, Dynamo has to maximize the impact of any optimizations that it performs. Dynamo primarily looks for performance opportunities such as redundancies that cross static program boundaries. Another performance opportunity is instruction cache utilization since a dynamically contiguous sequence of frequently executing instructions may often be statically non-contiguous in the application binary.

IA-32 Execution Layer (EL) [10] is a two-phase dynamic translator that executes IA-32 application on Itanium-based systems. Similar to HP Dynamo, it executes only applications, and is available on both MS Windows and Linux operating systems. Initially, basic blocks are translated by IA-32 EL by using prepared translation template, and are instrumented to collect information for the second phase. When a block is found

to be invoked enough times, the subsequent blocks executed are selected as a hyperblock. Based on the information collected during the first phase, various optimizations are applied on the hyperblock to improve its performance.

2.4.3 Hot spot detection and optimization

Dynamic optimization systems usually focus costly optimizations on program hot spots to reduce runtime compilation overhead and improve performance. Program *hot spots* are frequently executed code sequences, such as procedures [5] or basic block groups [8][18][24]. To amortize the overhead of runtime translation and further improve performance, most DO systems apply high-cost, high-payoff optimizations only to hot spots. A DO system usually includes the following steps to detect and optimize hot spots. Initially, a program code block is interpreted [8][18][24] or quickly translated and instrumented [5]. The execution frequency information of the code block is then gathered by the interpreter or the profiling code instrumented at hot spot boundaries, and saved in the code block's corresponding entry in the DO database that stores runtime profiling information for the DO system. The information is then examined to find frequently executed code blocks as hot spots, and advanced optimizations are applied on them.

The hot spot detection mechanism in the DO system can be used directly for phase identification. Wu et al. [84] indicate that the runtime characteristics of hot spots are usually stable throughout program execution. Huang et al. [36] and Merten et al. [56] observe that program phase behavior is closely related with hot spot invocations. Hence, tuning and reconfiguring CUs at hot spot boundaries should accurately adapt to program changes.

2.5 Power-aware compiler optimizations

With the importance of energy reduction, previous work examines the impact of static compiler optimizations on program energy consumption and power dissipation. Kandemir et al. [41] investigate the influence of several compiler optimizations, such as linear loop transformations, blocking, loop unrolling and loop fusion, on programs' energy consumption. They find that those optimizations usually reduce the memory energy consumption at the cost of rising core energy consumption. Valluri et al. [80] note that compiler optimizations for locality and instruction counts usually optimize program energy consumption, while optimizations improving ILP increase average power dissipation. In their study of Pentium 4 power consumption, Seng et al. [71] note that compiler optimizations affect programs' energy consumption mainly by reducing their execution time. Chakrapani et al. [16] examine the interaction between compiler optimizations and processor components, and draw similar observations as Valluri [80].

Power-aware compiler optimizations improve program energy consumption and power dissipation. Simunic et al. [77] perform several compilation optimizations, including loop merging, unrolling, software pipelining, loop invariant extraction to optimize the performance and energy consumption of a MPEGAUDIO video decoder in an embedded system, with the assistance of profiling information. Software pipelining is used by Yang et al. [85] in a power-aware instruction scheduling scheme to reduce power variation throughout program execution.

Compiler optimizations can also assist hardware energy reduction proposals. Valluri et al. [81] use the compiler to assist instruction scheduling, thus achieving low power, low-complexity instruction issue. Compared with run-time scheduling, compile-time scheduling features fast and simple hardware, but at the expense of conservative schedules [32]. The paper implements a compile-time analyzer to identify basic blocks

free of memory misses, false-dependences, or unresolved alias edges, and schedules them statically. This issue queue design includes two types of queues, a FIFO queue for statically scheduled instructions and a fully associative queue for instructions requiring dynamically scheduling. The scheme consumes less energy than the conventional issue queue since the fully associative queue is much smaller, and has a smaller issue width than the conventional one.

A compiler-directed strategy [34] identifies non-critical program code regions, and dynamic voltage and frequency can be scaled down on those regions for energy reduction. The compiler can also detect program regions that do not use certain hardware components, e.g., hard disk, for certain long enough periods so that the hardware components can be in hibernation for energy reduction [32].

All of the above research focuses on static compiler optimizations. Recently, Wu et al. [83] propose a lightweight dynamic compilation framework for dynamic voltage and frequency scaling (DVFS). This framework does not include the JIT optimization or garbage collection capability. It detects hot code regions that are memory-bound, and inserts DVFS mode set instructions at the boundaries of those regions so that they can be executed in energy-efficient modes.

Chapter 3. Hardware Adaptation Framework based on Dynamic Optimization

In the presence of multiple CUs, especially those that have diverse reconfiguration overheads, existing resource adaptation schemes have considerable limitations. To efficiently manage multiple CUs, a hardware adaptation framework based on a generic dynamic optimization system is developed and presented in this chapter. Although the idea of integrating hardware adaptation with a virtual machine is not new [19], to the best of our knowledge, this research is the first one that utilizes a DO system's inherent capabilities of detecting and optimizing hot spots to efficiently manage multiple configurable hardware resources.

Figure 2 shows the flowchart of the proposed hardware adaptation framework. Thin lines indicate program control flows, and thick lines represent data flows. Three main tasks are performed. Initially, the DO system monitors program execution and detects hot spots. After a hot spot is detected and JIT optimized, the DO system inserts tuning code at hot spot boundaries to identify the most energy-efficient hardware configuration for the hot spot during its subsequent invocations. After the tuning finishes, the JIT compiler replaces the tuning code with the code that automatically adapts to the hot spot's most energy-efficient configuration whenever it is invoked. The details of the framework are explained in the following sections.

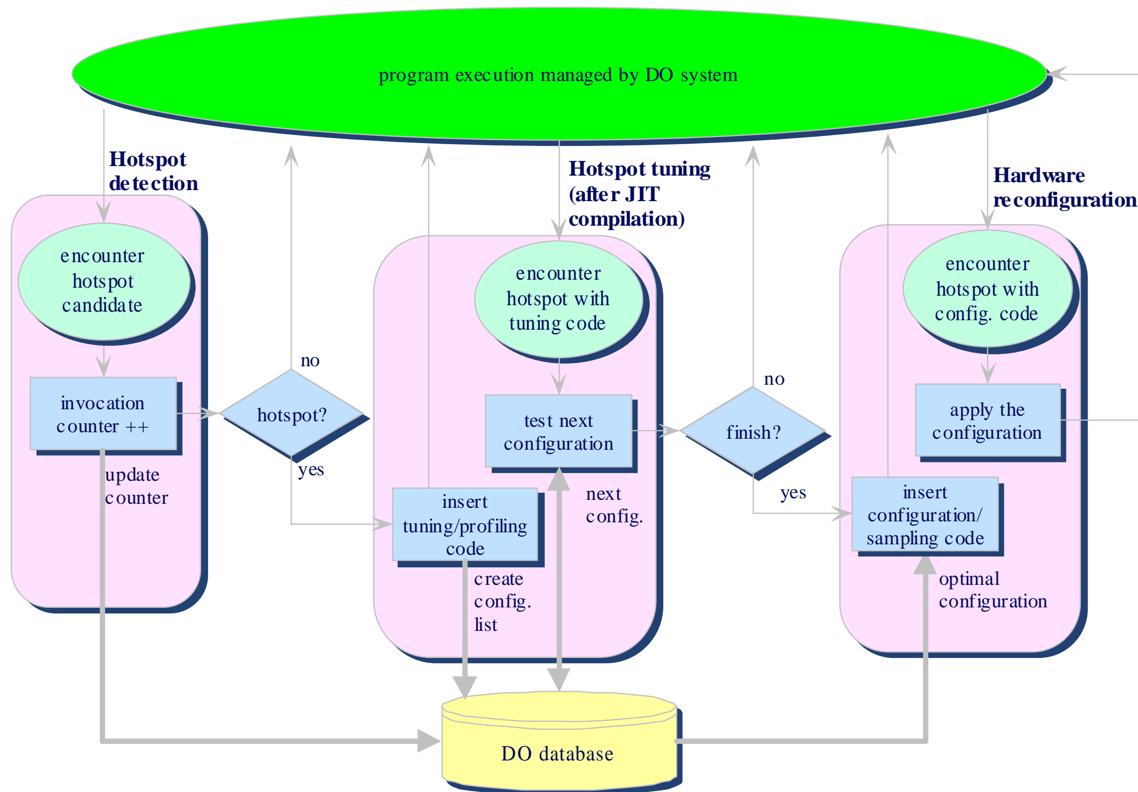

Figure 2. Flowchart of the proposed hardware adaptation framework based on a dynamic optimization system.

3.1 Hot spot detection

Program *hot spots* are frequently executed code sequences, such as procedures [5] or basic block groups [8][18][24]. To amortize the overhead of runtime translation and further improve performance, most DO systems apply high-cost, high-payoff optimizations only to hot spots. A DO system usually includes the following steps to detect and optimize hot spots. Initially, a program code block is interpreted [8][18][24] or quickly translated and instrumented [5]. The execution frequency information of the code block is then gathered by the interpreter or the profiling code instrumented at hot spot boundaries, and saved in the code block's corresponding entry in the DO database that

stores runtime profiling information for the DO system. The information is then examined to find frequently executed code blocks as hot spots, and advanced optimizations are applied on them.

The hot spot detection mechanism in the DO system can be used directly for phase identification. Wu et al. [84] indicate that the runtime characteristics of hot spots are usually stable throughout program execution. Huang et al. [36] and Merten et al. [56] observe that program phase behavior is closely related to hot spot invocations. Hence, tuning and reconfiguring CUs at hot spot boundaries should accurately adapt to program changes.

3.2 CU DECOUPLING AND HOT SPOT TUNING

After a hot spot is detected, the CU decoupling technique is applied to the hot spot to reduce its tuning process.

3.2.1 CU decoupling

Hot spots have variable sizes. In the same time, configurable units have different adaptation costs and reconfiguration intervals. Hence, low-overhead CUs are adapted at the boundaries of small hot spots, while high-overhead CUs are adapted at the boundaries of large hot spots. To do so, CUs are matched with hot spots that are similar in size to the CUs' reconfiguration intervals. This technique is called *CU decoupling* since it decouples the reconfiguration of different CUs in a multiple-CU system, and allows multi-grain configuration of CUs.

The effectiveness of CU decoupling relies on the properties of hot spots. Hot spots are nested, i.e., a large hot spot usually contains many small hot spots. Hence, when the small hot spots tune low-overhead CUs, those CUs are automatically tuned for the outside large hot spot. Consequently, adapting different CUs at different hot spots

boundaries does not sacrifice the CUs' reconfiguration opportunities. In fact, CU decoupling allows low-overhead CUs to capture small-grain phase changes and adapt accordingly, and thus improves performance.

3.2.2 Hot spot tuning

After a new hot spot is detected and JIT optimized, the subset of CUs is chosen for the hot spot, such that the CU's reconfiguration interval sizes match the hot spot size (an implementation will be given in Section 4.1). Then, a list of configuration combinations of the selected CUs is created and added to the hot spot's DO database entry, with an index initially pointing to the first list item. Next, the tuning code is inserted at the entry point of the hot spot and the profiling code at all exit points of the hot spot. Immediately after the hot spot's invocation, the tuning code fetches the configuration pointed to by the list index and increments the index, and then adapts the hardware according to the fetched configuration. When leaving the hot spot, the hot spot's performance characteristics under the current configuration are gathered. The configurations applicable to the hot spot are thus tested one by one until all configurations are tested. The most energy-efficient configuration is then selected to complete the hot spot's tuning process.

Contrary to the conventional strategy that tests all interdependent CUs, this tuning strategy adapts only a subset of all CUs for each hot spot. Consequently, this technique significantly reduces the tuning process, and allows multi-grain adaptation that further improves performance.

With CU decoupling, it is possible that during the tuning process for one hot spot, a new hot spot is detected so that a different set of CUs is selected for the hot spot and needs to be tuned. In this case, the newly detected hot spot's adaptation stalls until the former finishes its tuning process. Intuitively, changes of CU's configurations may alter

the adaptation preferences of other CUs. Hence, by allowing only one set of CUs to be adapted each time, the interference between two sets of CUs' adaptation is minimized.

3.3 HARDWARE RECONFIGURATION

Once the most energy-efficient configuration of a hot spot is found, the JIT compiler is invoked to perform the following two tasks. First, the tuning code at the beginning of the hot spot is replaced by the configuration code that sets the adaptive microarchitecture to the hot spot's most energy-efficient configuration. For each tuned hot spot, CUs will be changed to the hot spot's most energy-efficient configuration just prior to the hot spot's execution. In contrast to temporal approaches, there will be no further tuning latency or phase identification latency incurred by the hot spot.

Additionally, the profiling code at a hot spot's exit is replaced by the sampling code that occasionally gathers performance statistics to detect any performance change between the hot spot's current and prior invocations. A large performance change indicates that the hot spot's behavior may have altered. Consequently, the hot spot is tuned again. As observed by Wu et al. [84], runtime characteristics of hot spots are usually stable throughout program execution, and thus such re-tunings should be rare.

3.4 HARDWARE SUPPORT

The proposed scheme is mainly a software approach. It relies on the underlying DO system to detect and adapt program hot spots. However, some minimal hardware support is needed. This dissertation assumes that each CU has a control register, and the CU's configuration can be changed by setting the register value. To allow resource adaptation by software, a special instruction is required to change the values of the control registers. Note that in this framework, the DO system configures hardware, which is potentially less error-prone than allowing user applications to control hardware

adaptation directly. The above functionality can also be implemented as system calls. Comparing with the reconfiguration overheads of hundreds or thousands of cycles, the overheads of invoking those system calls are negligible.

For each CU, a hardware counter holds its most recent reconfiguration time. Each time a CU's configuration is changed, its last-reconfiguration counter is updated with the current time. When a CU reconfiguration request arrives, the time elapsed since the CU's last reconfiguration is calculated. If the interval is shorter than the CU's reconfiguration interval, the request is ignored without modifying the CU's configuration. Frequent reconfiguration requests could happen when two nested hot spots adapt the same CU with different adaptation preferences, and the nested hot spot is invoked immediately after the nesting one. With the last-reconfiguration counters, the proposed framework is freed from the burden of maintaining the minimal reconfiguration interval for each CU.

The research assumes that performance and power are measured using hardware performance counters. Note that performance counters cannot directly measure power. Nevertheless, they can be used to accurately estimate power [51]. Measuring performance and power with performance counters incurs error, especially when the sampling interval is short. The error can be reduced in different ways. First, we can choose larger reconfiguration interval sizes for the CUs to improve the measurement accuracy. Furthermore, we can adopt a sampling tuning approach so that the measurement happens after multiple, instead of one, invocations of a hot spot, which effectively increase the sampling interval length and improve the accuracy of the measurement. Those two approaches also compensate for the cost associated with accessing the performance counters.

If a system does not have performance counters, we can estimate power using the CUs' configuration information. Each configuration's power can be estimated accurately

via extensive offline experiments. Such configuration-power information is stored in a table accessed at runtime by the DO-based framework to make accurate adaptation decisions.

3.5 DIFFERENCES WITH PRIOR APPROACHES

In temporal approaches, a phase change cannot be identified immediately. It can only be detected after the phase change lasts one or more sampling interval, which is called the identification latency. Recurring phases in those temporal approaches incur phase identification latencies, regardless the length of program execution. Recurring phase identification latencies can be reduced by next phase detection mechanisms [50][76], which predict what the next phase will be and when it will occur. However, incorrect predictions cause unnecessary or wrong adaptation and subsequent rollback of hardware configurations, thus affecting performance considerably. Hence, high prediction accuracy is imperative for such mechanisms.

In comparison, in a DO-based system, only new hot spots need to be detected, and recurring hot spots are identified immediately. Hence, a hot spot's detection overhead is a one-time cost, and thus can be diminished by long program execution. Moreover, since hot spots are often nested, detections of new hot spots are often overlapped, further reducing the overall hot spot identification cost. Table 4 show hot spots execute frequently. Hence, although hot spots take more time than BBV phases to be recognized, the hot spot identification cost does not pose a big burden to the hot spot approach.

This hot spot-based framework is essentially a software positional approach. Differing from the original positional approaches [36], the proposed framework detects hot spots instead of large procedures. The frequent-invocation nature of hot spots ensures that the most energy-efficient hardware configuration of a hot spot can be applied enough times for high benefit, while the positional approach can not enjoy this feature from the

large procedures it uses. Furthermore, the original positional approach requires considerably efforts to detect hierarchical phase changes and adapt hardware accordingly, which, in contrast, is accomplished by our framework in a natural and elegant way.

Magklis et al. [54] use offline profile-driven binary rewriting tools to insert reconfiguration instructions into the applications to avoid the complexity of online hardware reconfiguration. Differing with our framework that obtains the profiling information on the fly, this scheme needs several training runs of the applications to obtain the profiling information. Both approaches have their advantages and disadvantages. For instance, using offline tools incurs no cost for runtime profiling and is applicable to most programs. In contrast, DO-based framework incurs runtime profiling overhead, although the overhead corresponding to hardware adaptation is minimal since the framework leverages the existing infrastructure of the underlying dynamic optimization system. On the other hand, the DO-based framework requires minimal human intervention achieve reconfigurations, and all applications executed by the framework can enjoy the benefits without the painful work of tuning each application for each hardware platform.

3.6 APPLICABILITY

In recent years, dynamic optimization systems are deployed in various software levels, such as operating systems (e.g., Microsoft .NET Common Language Runtime [91]), or virtual machines emulating certain hardware platforms (e.g., IBM Daisy [24] and Transmeta Crusoe [18]). In this research, the DO-based framework is implemented and evaluated in Jikes RVM, which is a user-level virtual machine. Nevertheless, the framework is not confined to Jikes RVM, or user-level dynamic optimization systems. Most dynamic optimization systems can be easily augmented to integrate the proposed hardware adaptation framework.

One difference between some DO systems and Jikes RVM is that they detect frequently executed basic blocks, instead of procedures, as hot spots (e.g., IBM Daisy [24] and Transmeta Crusoe [18]). Differing to procedure-based hot spot detection, basic-block-based hot spots are not nested and are usually fine grain. Since CU decoupling requires the hot spots to be nested and of variable sizes, it cannot be directly used in those DO systems. This issue can be solved. In those DO systems, branch execution frequencies are monitored to detect hot basic blocks. The same mechanism can keep track of branches between hot spots, and group those hot spots into larger ones to provide size variable and nested hot spots. This simple enhancement allows the DO systems to use CU decoupling for efficient management of multiple CUs with diverse adaptation costs as the proposed framework does.

Chapter 4. Experimental Methodology

In this research, the proposed framework is implemented with the Dynamic Simplescalar simulator [38] and the Jikes Research Virtual Machine (RVM) [5]. The proposed hardware adaptation framework is evaluated with the SPECjvm98 benchmark suite [88].

Table 1. Baseline configuration of the simulated system.

CPU	
Instruction window	128-IFQ, 128-RUU, 64-LSQ
functional units	4 intALU, 2 IntMult/Div,4 FPALU, 2 FPMult/Div
Branch predictor	2K-entry combined predictor, 3-cycle misprediction penalty
Issue/Commit width	4 instructions per cycle
Memory Hierarchy	
L1 I-cache	64KB, 64B blocks, 4-way, LRU, 1 cycle hit latency
L1 D-cache	64KB, 64B blocks, 4-way, LRU, 1 cycle hit latency,
L2 unified cache	1MB, 128B blocks, 4-way, LRU, 10 cycles hit latency
DTLB/ITLB	128 entries, fully set associative

4.1 Simulation Environment

4.1.1 Dynamic Simplescalar

The Dynamic Simplescalar (DSS) [38] simulator used in our work adds a series of major extensions to Simplescalar/PowerPC version 3.0, and permits simulation of a full Java run-time environment on a detailed simulated hardware platform. The original Simplescalar framework cannot simulate Java programs due to its inability to handle self-modifying code. DSS resolves the problem and implements support for dynamic code generation, thread scheduling and synchronization, as well as a general signal mechanism that supports exception delivery and recovery. The newer version of DSS incorporates a

power model that is based on Wattch [13]. The operating frequency and voltage of the target processor are 1GHz and 2V respectively. The baseline processor configuration is presented in Table 1.

4.1.2 Basic block vector phase detection

We also implement the basic block vector (BBV) approach [76] in DSS, and compare it with our framework. BBV is shown to be one of the best phase detection techniques [20]. It partitions the dynamic instruction stream into contiguous fixed-length intervals, and gathers dynamic basic block distributions through an array of counters, called the accumulator table, for each interval. Hence, at the end of each interval, the accumulator table obtains the code signature for the interval. A large Manhattan distance between two consecutive intervals' code signatures of indicates a phase change [76]. If the Manhattan distance between two code signatures is smaller than a predetermined phase classification threshold, the two intervals are classified into the same phase.

In our BBV implementation, the hardware accumulator table has 32 entries, as in [76]. Each table entry is 20 bits, so it will never overflow with 1 million sampling intervals. For each branch instruction executed, the lower 5 bits (excluding 2 least significant bits) of the branch PC indexes an accumulator table entry and increments the entry by the number of instructions executed since the last branch. The phase classification threshold is 20%. We also evaluate phase classification thresholds of 12.5% and 25% that are used in previous BBS literature [76]. A 25% threshold yields similar results to those using a 20% threshold. On the other hand, using 12.5% threshold increases the number of detected phases and affects the code coverage of BBV phases considerably.

To give an advantage to the BBV approach, This BBV implementation uses the values from the accumulator table directly to form a phase's signature, and thus loses no

information due to signature compression [76]. Furthermore, this BBV implementation allows unlimited number of BBV phase signatures. However, this BBV implementation does not contain a next phase predictor since both [50] and our preliminary results confirm that using costly next phase predictors provide only marginal benefits over no prediction.

4.1.3 Configurable hardware units

In this work, five size-adaptable hardware units are implemented. They are issue queue (IQ), reorder buffer (ROB), level-one data (L1D) and instruction (L1I) caches and level-two (L2) cache (Table 2). Each configurable unit has four different sizes. The power model is augmented to reflect the runtime size reduction of the CUs and the power consumed for reconfiguring the hardware (i.e., power consumed for writing dirty cache lines into the lower level of memory hierarchy). Changing a hardware unit's setting at runtime usually incurs a reconfiguration overhead. A program's performance may be impaired by too frequent reconfigurations where reconfiguration overhead exceeds the benefit gained by the reconfigurations. After a reconfiguration, the configuration should be utilized for a certain minimum time interval, called the unit's reconfiguration interval (presented in the third column of Table 2).

Table 2. Configurable hardware units and their adaptation parameters.

Configurable units	Configurations (units)	Reconfiguration interval (instructions)	Hot spot size (instructions)	BBV sampling interval size (instructions)
IQ/ROB	128/96/64/32 (entries)	1000	1K – 100K	10K
L1I/L1D caches	64K/48K/32K/16K (bytes)	100K	100K – 1M	100K
L2 cache	1M/768K/512K/256K (bytes)	1M	> 1M	1M

The hardware adaptation framework proposed in this work adapts configurable units at program hot spot boundaries. To avoid testing all CUs at the same time, each hot spot tests only a subset of CUs. The fourth column of Table 2 lists the size ranges of hot spots at whose boundaries corresponding CUs are adapted. Hot spots between 1K and 100K instructions long configure the issue queue and the reorder buffer (IQ-ROB hot spots), and hot spots adapting the L1I and L1D caches (L1 hot spots) are between 100K and 1M instructions long, while L2 hot spots are at least 1M instructions long.

To evaluate how well the DO-based framework reduces energy, we compare it with a hardware adaptation scheme using the BBV phase detection approach. The detail and parameters of the BBV phase detection approach are given in Section 4.1.2. To allow a fair comparison of the two hardware adaptation approaches, the BBV approach also decouples the tuning of different CUs, which can be implemented in multiple ways. This work uses multiple accumulator tables, each corresponding to a sampling interval size and detecting BBV phases of different granularities. Similar to the DO-based framework, a CU's adaptation happens only at the boundaries of phases whose granularities (i.e, sampling interval sizes) match the phase's reconfiguration interval size. The last column of Table 2 shows the sampling interval sizes that correspond to the CUs.

Note that in Table 2, issue queue and reorder buffer are adapted at boundaries of 10K-instruction BBV phases, instead of 1K-instruction BBV phases, although the CUs have a reconfiguration interval size of 1K instructions. This is mainly due to that we find that for the programs evaluated in this work, 1K-instruction stable BBV phases usually have much poorer code coverage than 10K-instruction phases. Consequently, 10K-instruction phases achieve higher energy reduction on IQ and ROB, than 1K-instruction phases.

4.2 DYNAMIC OPTIMIZATION SYSTEM

Jikes RVM is a research Java virtual machine (JVM) developed at IBM T. J. Watson Center [5]. It is written in Java. This enables the optimization techniques to be applied to both the application code and the JVM itself. The 2.0.2 version of Jikes RVM is used since Dynamic Simplescalar is not compatible with the latest version of Jikes RVM.

Jikes RVM employs a compile-only strategy (i.e., no interpreter mode). It includes a baseline and an optimizing compiler. The optimizing compiler has three levels of optimizations, each one consisting of its own group of optimizations as well as the optimizations that belong to lower levels. Initially, code sequences are compiled by the baseline compiler.

The Jikes RVM uses a low-overhead sampling method to detect program hot spots. Approximately every 10 milliseconds, Jikes increments a counter associated with the currently active procedure. For all methods that have been sampled, Jikes uses a cost/benefit model to determine whether it is profitable to recompile the method, and if so, what level of optimization to use.

By default, the optimizing compiler with the level-one compilation level is used to compile all methods. The simplification allows us to focus on the implementation and evaluation of the proposed hardware adaptation framework. Furthermore, in all experiments except Chapter 7, a large heap (200MB) is chosen to reduce garbage collection activities. By doing so, execution is dominated by the application rather than Jikes RVM.

As described in Chapter 3, hardware tuning and reconfiguration are performed after hot spots are detected and JIT optimized. The functionality is implemented in the

Jikes optimizing compiler, and Jikes' global data structure is also modified to store the necessary information for the hardware tuning and reconfiguration.

Table 3. Characteristics of SPECjvm 98 benchmarks executed by Jikes RVM.

	compress	**jess**	**db**	**javac**	**mpegaudio**	**mtrt**	**jack**
Instruction count (billions)	9.83	5.73	8.78	8.90	10.93	4.17	8.18
Cycle count (billions)	5.30	4.93	14.15	7.66	6.98	2.76	5.88
IPC	1.85	1.16	0.62	1.16	1.56	1.51	1.39
Energy consumption (J)	141.70	98.08	219.96	153.39	174.55	63.46	127.33

4.3 BENCHMARKS

The industry standard SPECjvm98 benchmarks are used to evaluate the proposed framework. Among the programs in the SPECjvm98 suite, 200_check is not considered in this study since its only purpose is to check the functionality of a JVM. The experiments use the largest s100 data sets of SPECjvm 98 benchmarks. Table 3 provides a summary of these SPECjvm98 benchmarks. By default, each benchmark runs only one iteration, and the results are obtained while Jikes RVM uses the level-one JIT compilations and a 200M heap. Comparing with server-side Java benchmarks, such as SPECjbb 2000 [89], SPECjvm 98 benchmarks measure the performance of client-side Java virtual machines, and are short running.

Chapter 5. Evaluation of the Hardware Adaptation Framework

To evaluate the proposed framework, it is compared with an adaptive system that uses the BBV phase detection technique proposed by Sherwood et al. [76], which has been proven to be one of the best phase detection schemes. Similar to the DO-based framework, the BBV-based approach decouples the adaptation of CUs with diverse adaptation costs by matching them with BBV phases whose sampling interval sizes equal to the CUs' reconfiguration interval sizes.

The BBV technique used in this research does not use the phase prediction mechanisms presented in [50] and [76]. Theoretically, accurate phase prediction tells what the next phase will be and when it will occur, and thus can improve the coverage of resource adaptation. However, implementing complex next phase predictors in hardware is costly. Both Lau et al. [50] and our preliminary results confirm that using next phase predictors provide only marginal benefits over no prediction. Hence, in this research, the BBV results are obtained without next phase prediction. In contrast, the hot spot approach always identifies upcoming phases immediately, and does not need next phase prediction.

In this experiment, the two resource adaptation approaches adapts all the five CUs described previously. BBV phases with three different sampling interval sizes are detected. 1M-instruction BBV phases manage the L2 cache. 10K-instruction phases adapt issue queue and reorder buffer, and 100K-instruction phases manage L1D and L1I caches. For the DO-based framework, hot spots that configure the issue queue and the reorder buffer (*IQ-ROB hot spots*) are between 1K and 100K instructions long, and hot spots adapting the L1I and L1D caches (*L1 hot spots*) are between 100K and 1M instructions long, while *L2 hot spots* are at least 1M instructions long. The IPC

degradation thresholds are 5% for both approaches, essentially each CU with a 1% IPC loss budget.

A preferable feature of a resource adaptation approach is its capability to retain most of the energy reduction for existing CUs as more hardware units become adaptable. In this dissertation, this feature is called *adaptation efficiency*, which is investigated in Section 5.3.

5.1 RUNTIME CHARACTERISTICS OF HOT SPOTS AND BBV PHASES

In this section, the runtime characteristics of hot spots are obtained and compared with those of BBV phases.

5.1.1 Runtime characteristics of hot spots

Table 4 presents the runtime characteristics of hot spots in SPECjvm98 benchmarks. In this research, only the hot spots whose average sizes are longer than 1000 instructions are considered. Each program usually has hundreds of such hot spots. The average sizes of those hot spots are more than 14000 instructions, indicating that many hot spots are very long.

Table 4. Runtime hot spot characteristics of SPECjvm 98 benchmarks

	compress	**db**	**jack**	**javac**	**jess**	**mpegaudio**	**mtrt**
dynamic instruction count	9.83E+09	8.78E+09	8.22E+09	8.92E+09	5.72E+09	1.09E+10	5.10E+09
number of hot spots	299	316	470	685	434	386	363
average hot spot size	81,645	75,648	14,941	23,774	77,841	70,231	18,617
average invocations per hot spot	823	1,105	13,091	5,983	2490	4,747	3,284
per-hot spot IPC CoV	9.17%	9.97%	6.74%	9.33%	7.79%	5.37%	8.09%
Inter-hot spot IPC CoV	43.78%	42.99%	49.38%	46.47%	52.49%	49.05%	46.69%

On average, the resulting hot spots execute at least 823 times. One disadvantage of the proposed DO-based framework over temporal approaches is the former's long initial hot spot identification latency. The invocation results clearly shows that although hot spots take more time than BBV phases to be recognized, it does not pose a big burden to the hot spot approach.

To characterize how well hot spots represent program phase behavior, Table 4 also provides the per-phase and inter-phase IPC coefficient-of-variations (CoVs) of hot spots. CoV equals the percentage of the standard deviation divided by the average, and measures data dispersion compared to the mean. The per-phase IPC CoVs are IPC variations among different invocations of the same hot spot, characterizing the homogeneity among different invocations of a hot spot. The inter-phase IPC CoVs are the variations of average IPCs of different hot spots, and larger inter-phase IPC CoVs signify large differences between the characteristics of detected hot spots. More than 34% disparities between hot spots' per-phase and inter-phase CoVs demonstrate that the differences between hot spots are more prominent than those between invocations of the same hot spot, further confirming that hot spots are closely related with program behavior changes.

Table 5. Runtime characteristics of hot spots in three different size ranges.

	compress	**db**	**jack**	**javac**	**jess**	**mpegaudio**	**mtrt**
number of IQ-ROB hot spots	213	229	358	544	336	299	269
IQ-ROB coverage	98.56%	73.12%	81.26%	88.57%	78.52%	89.77%	80.28%
number of L1 hot spots	64	58	81	108	68	64	73
L1 coverage	71.70%	87.90%	85.00%	81.20%	92.70%	91.00%	81.40%
number of L2 hot spots	22	29	31	33	30	23	21
L2 coverage	73.80%	87.90%	56.90%	80.00%	84.30%	95.70%	82.60%

Table 5 counts the number of IQ-ROB, L1 and L2 hot spots. As shown in the table, there are many more small hot spots than large ones. Since low-overhead CUs are adapted at the boundaries of small hot spots, this property, as well as the fact that small hot spots are invoked more frequently than large hot spots, guarantees that low-overhead CUs are adapted more frequently than high-overhead CUs.

Table 5 also gives the coverage (i.e., the portion of dynamically executed instructions within the hot spots) results for the three types of hot spots. As shown in the table, most IQ-ROB, L1D and L2 hot spots have coverage larger than 80% across benchmarks. The exceptions are compress (72% L1 coverage and L2 coverage), db (73% IQ-ROB coverage), jack (57% L2 coverage), and jess (79% IQ-ROB coverage). Good coverage and numerous hot spot invocations indicate that CU decoupling does not sacrifice each CU's reconfiguration opportunity.

5.1.2 Runtime characteristics of BBV phases

As explained in Section 2.2, in temporal approaches, phases can be classified into stable and transitional ones. Transitional phases have short lifetimes. They last only one sampling interval, rarely recur, and are thus difficult to tune. Recognizing only stable phases that last two or more continuous intervals can improve the phase detection hardware utilization and increase phase detection accuracy considerably. Figure 3 shows the distributions of stable and transitional BBV phases of SPECjvm 98 benchmarks. The phases are identified by the BBV method with 1M-instruction sampling intervals. As demonstrated by *javac*, one major drawback of ignoring transitional phases is the considerably reduction of the coverage of resource adaptation. By default, BBV phases studied in this paper are all stable ones. Hence, the results cannot be directly compared with the results in prior BBV literature [49][50][76].

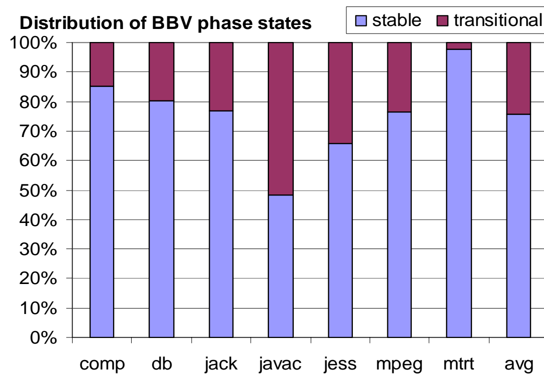

Figure 3. Distribution of stable/transitional BBV phases of SPECjvm 98 benchmarks. (A phase is stable if it has two or more successive sampling intervals; otherwise it is an unstable phase)

Programs phases are usually hierarchical, i.e., nested and with multiple granularities. Hence, BBV phases with different sampling interval sizes may exhibit distinct runtime characteristics. Programs phases are usually hierarchical, i.e., nested and with multiple granularities. Hence, with a fixed sampling interval size, the BBV approach can detect only phases in a range of granularities. For instance, BBV phases with large sampling intervals may overlook short and sharp phase changes that can be identified by BBV phases with small sampling intervals. On the other hand, the later, with its limited scope, may be unaware of long and incremental changes of program behavior, and thus captures distinct runtime characteristics to the large BBV phases. Table 6 presents the runtime characteristics of stable BBV phases with 10K-instruction and 1M-instruction sampling intervals respectively.

All benchmarks have hundreds of 10K-instruction BBV phases, but only tens of 1M-instruction BBV phases. Since the tuning process is an overhead in terms of energy saving and each phase needs to be tuned, too many phases will inevitably impair performance of the BBV approach when using 10K-instruction sampling intervals.

Table 6. Runtime characteristics of stable BBV phases with 10K and 1M sampling intervals.

		compress	db	jack	javac	jess	mpegaudio	mtrt
10K	number of BBV phases	622	576	890	1529	884	794	773
	% of code in BBV phases	82.01%	72.72%	47.09%	42.37%	53.38%	46.98%	53.79%
	per-phase IPC CoVs	8.04%	8.72%	9.41%	8.96%	8.91%	7.54%	8.19%
	inter-phase IPC CoVs	48.03%	50.91%	44.31%	35.59%	44.35%	49.28%	49.08%
1M	number of BBV phases	70	50	70	84	80	58	75
	% of code in BBV phases	85.02%	80.26%	77.29%	48.15%	65.61%	76.47%	77.42%
	per-phase IPC CoVs	4.07%	9.10%	7.35%	6.59%	5.20%	4.91%	6.24%
	inter-phase IPC CoVs	20.05%	33.32%	20.07%	24.87%	26.11%	38.26%	23.96%

Similar to the hot spot results in Table 5, Table 6 also contains stable BBV phases' coverage results (i.e., portions of dynamic program code in those stable BBV phases). Since CUs are adapted at stable BBV phase boundaries, the coverage results estimate how mach of dynamic program code can benefit from CU adaptation. Overall, hot spots have better coverage than BBV phases.

For the two BBV configurations, 10K-instruction phases have worse code coverage than 1M-instruction phases. Intuitively, in terms of BBV phase detection, a fluctuation of branch frequency affects small intervals more than large intervals, since large intervals contain many more branches to minimize the impact of a single fluctuation of branch frequency on phase detection. Hence, a small interval is more likely to be classified as a different phase to its neighboring intervals, attributing to the poor coverage of stable 10K-instruction BBV phases. For the 10K-instruction BBV phases, three out of seven benchmarks have code coverage less than 50%. On the other hand, except for javac, all benchmarks have good code coverage using 1M-instruction BBV phases. On average, 10K and 1M-instruction BBV phases achieve 57% and 73% code coverage respectively.

Table 6 also gives the CoVs for the BBV phases. 10K-instruction BBV phases have larger per-phase CoVs than 1M-instrcution BBV phases, indicating that small BBV phases are less stable than large BBV, i.e., there are more variations among different sampling intervals of the same phase. Meanwhile, 10K-instruction BBV phases have also larger inter-phase CoVs than 1M-instruction phases, which may be because large phases are insensitive to fine-grain phase changes within large sampling intervals; such fine-grain phase changes can be detected by small BBV phases. Comparing hot spots (Table 4) with BBV phases, both the per-phase and inter-phase CoVs of hot spots are between the corresponding ones of small and large BBV phases, indicating that hot spots, with their variable sizes, can detect both fine-grain and coarse-grain program phase changes.

In summary, hot spots' IPC CoV results indicate that differences among hot spots are much larger than those among invocations of the same hot spot. Furthermore, hot spots' CoV results are comparable with those of BBV phases. The results clearly demonstrate that program hot spots closely represent phases. Since hot spots are of variable sizes and are often nested, they capture hierarchical program phase behavior in a natural and elegant way. As demonstrated in the next two sections, this property can be used for efficient management of multiple configurable units for high energy reduction.

5.2 ADAPTATION OF FIVE CONFIGURABLE UNITS

To demonstrate the scalability and efficiency of the hot spot-based approach on adapting multiple CUs, the BBV and hot spot schemes manage all the five CUs for energy reduction, and their runtime characteristics and energy reduction results are compared. For the hot spot approach, the five CUs are adapted at the boundaries of IQ-ROB, L1 and L2 hot spots respectively. To find the most energy-efficient configuration, each L2 hot spot corresponds to only one CU, the L2 cache, and needs to test all the four L2 configurations to find the best performing one. On the other hand, since an IQ-ROB

hot spot or a L1 hot spot adapts two CUs, it has to test sixteen combinatorial configurations of the two CUs to find the best performing one. Similarly, a 10K or a 100K BBV phase needs to test up to 16 combinatorial configurations, while a 1M BBV phase tests only 4 configurations. Note that it is possible to improve both approaches' energy reduction efficiency by avoiding testing certain unpromising configurations. For both approaches, the IPC degradation threshold is 5%.

Other differences exist between the two approaches. First, the BBV approach is more general than the DO-based framework in that it is applicable to any applications, while the DO-based framework can only optimize dynamically executed applications. On the other hand, as a software approach, the DO-based framework requires much simpler hardware support than the BBV approach.

5.2.1 Runtime characteristics

Table 7 presents the number of tuning attempts made (tunings) and the number of times the most energy-efficient configuration is applied (reconfigs) for both the hot spot and the BBV schemes. Owing to the use of the decoupling method to adapt the CUs, both approaches have more reconfigurations than tunings, indicating that both approaches devote more time on applying the optimal configurations for energy reduction than on testing the configurable units. Furthermore, since both approaches adapt low-overhead CUs at boundaries of fine-grain hot spots or BBV phases, low-overhead CUs are reconfigured more frequently than high-overhead CUs. Hence, low-overhead CUs can exploit fine-grain phase changes for energy reduction. On the other hand, IQ-ROB hot spots have many more reconfigurations than 10K-instruction BBV phases, indicating that hot spots have more opportunities to adapt issue queue and reorder buffer for better performance.

Table 7. Runtime characteristics while adapting the five configurable units.

		compress	db	jack	javac	jess	mpegaudio	mtrt
Hot spot	**IQ-ROB tunings**	38771	59494	983878	855701	176977	41555	177392
	IQ-ROB reconfigs	182219	244330	4844773	3148274	793500	197442	876231
	L1 tunings	1479	1876	6799	12377	2740	8697	10277
	L1 reconfigs	11302	11160	23958	34645	7713	34724	38141
	L2 tunings	99	132	189	268	155	366	104
	L2 reconfigs	821	1236	4284	2761	1245	8173	843
BBV	**10K tunings**	24721	9380	31824	43615	16815	29492	14308
	10K reconfigs	93481	18037	114671	94228	33557	91167	26651
	100K tunings	2096	1375	2500	3976	2778	1413	2394
	100K reconfigs	7301	2643	5348	10175	7031	1563	5928
	1M tunings	183	112	156	316	184	115	166
	1M reconfigs	623	239	324	839	290	335	147

Note that the impact of a long tuning process on long-running applications will not be as prominent as on short-running applications. However, with the ability of multi-grain adaptation, the CU decoupling technique helps a hardware adaptation scheme to reach better balance of adaptation benefit/overhead for each CU. This advantage does not diminish with longer execution.

5.2.2 Energy reduction

The energy reduction results are given in Figure 4 to Figure 8. For both the BBV and the hot spot approaches, the energy reduction results are obtained by comparing with the baseline ones that use the maximum sizes of the CUs throughout program execution. In each picture, the portions of energy reduced for a CU are presented for the two resource adaptation approaches.

Comparing with the DO-based framework on energy reduction, the BBV approach is slightly better on L1I and L2 caches, slightly worse on L1D cache, but considerably worse on issue queue and reorder buffer. The BBV approach's poor performance on issue queue and reorder buffer is mainly due to the poor code coverage

the BBV approach with 10K-instruction sampling intervals (Table 6). Furthermore, adapting issue queue and reorder buffer at a coarse granularity (10K instructions) than their optimal reconfiguration interval size (1K instructions), the BBV approach loses tuning opportunities for the two CUs. However, we find that using 1K-instruction sampling intervals yields poorer performance than using 10K-instruction sampling intervals, mainly due to that the code coverage of 1K-instruction BBV phases deteriorate significantly. On average, the BBV approach reduces the energy consumed by issue queue, reorder buffer, L1D and L1I caches, and L2 cache by 23%, 22%, 34%, 31% and 46% respectively. In comparison, the hot spot approach reduces the CUs' energy consumption by 33%, 28%, 37%, 31% and 45%.

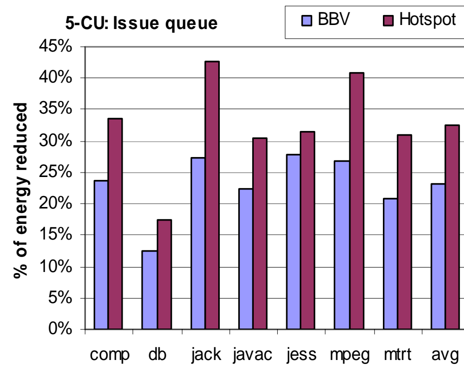

Figure 4. Reduction in issue queue energy consumption by adapting the five configurable units (5% performance loss threshold).

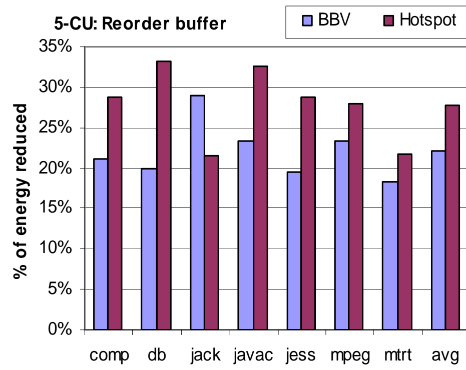

Figure 5. Reduction in reorder buffer energy consumption by adapting the five configurable units (5% performance loss threshold).

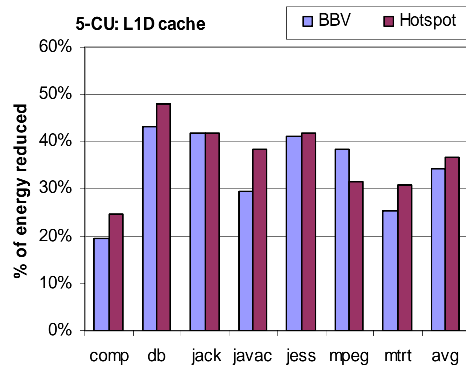

Figure 6. Reduction in level-one data cache energy consumption by adapting the five configurable units (5% performance loss threshold).

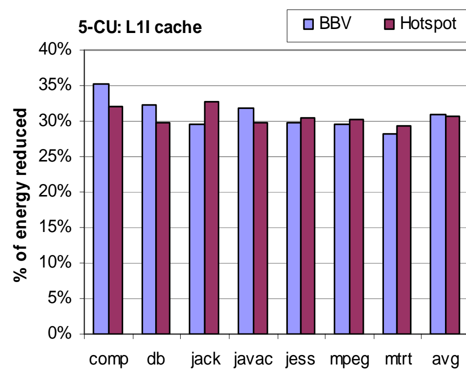

Figure 7. Reduction in level-one instruction cache energy consumption by adapting the five configurable units (5% performance loss threshold).

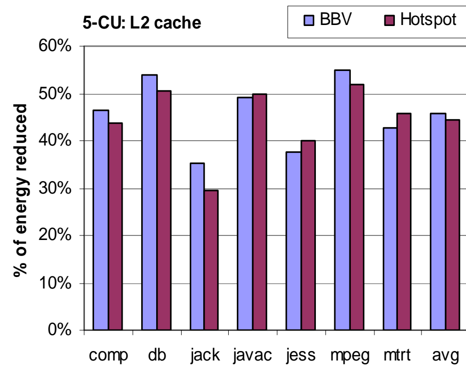

Figure 8. Reduction in level-two cache energy consumption by adapting the five configurable units (5% performance loss threshold).

5.2.3 Performance impact

The performance degradation results caused by adapting the five configurable units are presented in Figure 9. In this experiment, the target IPC degradation threshold is 5% for both approaches, and. Hence, both approaches incur similar performance degradations. On average, the BBV and the hot spot approaches yield IPC degradations of 5.0% and 4.9% respectively.

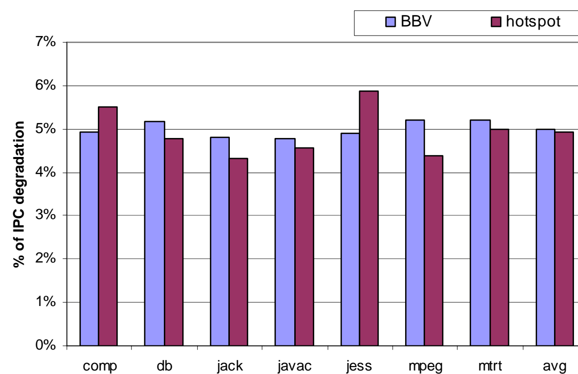

Figure 9. Performance degradation due to the adaptation of the five configurable units.

To understand how different factors attribute to the performance loss, we further separate different sources of IPC degradation for the hot spot approach, and obtain each one's contribution to the overall performance loss in Figure 10. Hardware adaptation related performance loss happens at both tuning and reconfiguration stages. There are two major sources for the performance loss. First, before downsizing the L1D cache and the L2 cache, dirty cache lines must be written back to the lower level of memory. Those adaptation overheads are called L1D and L2 costs in Figure 10. Size reduction of the configurable units is the second source of performance loss (IPC drop in Figure 10) when the size reduced configurable cannot fulfill program requirements. Figure 10 shows that on average, configurable units' size reduction accounts for 72% of performance loss, and the rest 28% of performance loss is due to the adaptation costs of L1D and L2 caches. The performance loss in the reconfiguration stage consists of 68% of overall performance loss.

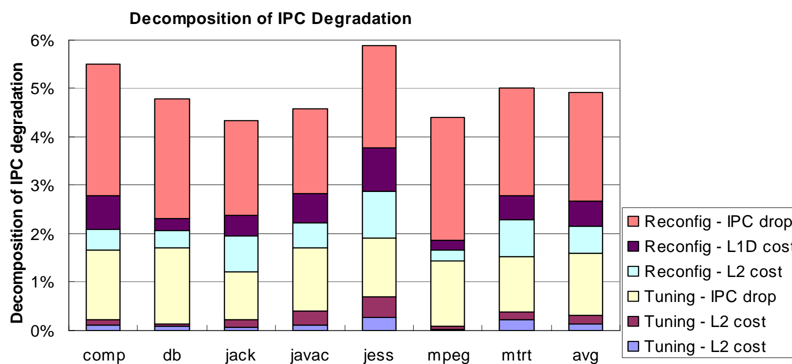

Figure 10. Decomposition of the DO-based approach's IPC degradation.

The proportions of different sources of adaptation overhead to the overall performance loss are affected by different factors. First, long-running applications rarely creates new phases constantly through their execution. Hence, the tuning costs are usually

mush smaller for the long-running applications than short-running applications. On the other hand, if an adaptive microarchitecture has more configurable units or each unit has more configurations, the tuning costs will increases dramatically as the tuning space explodes. Finally, tuning reduction techniques, such as CU decoupling, prunes the tuning space and effectively reduce the tuning costs.

The total adaptation costs for L1D and L2 caches consist of near 30% of the overall performance loss. Differing from tuning overhead, L1D/L2 adaptation costs do not diminish for long running applications since they are associated with the downsizing of the caches, which happens also at the reconfiguration stage. Those adaptation costs reduce the effective performance budget for hardware downsizing that reduces the energy consumption. Hence, high adaptation costs hurt the energy efficiency of an adaptive microarchitecture. For this purpose, a hardware adaptation framework can prioritize the CUs by their adaptation costs, and allow low-overhead CUs to be adapted more frequently. This prioritization can be implemented by further increase the reconfiguration intervals of high-overhead CUs.

In summary, this section demonstrates that the DO-based hardware adaptation framework can efficiently manage multiple configurable units, and achieves comparable energy reduction results to one of the best prior approaches. The results, in turn, further confirm that programs hot spots accurately capture program phase behavior. More importantly, being able to detect hot spots of multiple granularities, the framework can efficiently manage multiple configurable units with diverse reconfiguration costs, and achieve high overall energy reduction.

5.3 ADAPTATION EFFICIENCY

To achieve high overall energy reduction, an adaptive microarchitecture usually contains several configurable units. Intuitively, the configuration changes of a CU should

affect the adaptation preferences of another CU. Hence, when an adaptive microarchitecture contains more CUs, each CU's energy reduction should drop. This section examines the impact of the interference between CUs on each CU's energy reduction. In this work, we call a CU's capability to retain most of its energy reduction in a multi-CU system the configurable unit's *adaptation efficiency*.

In two experiments, the hot spot approach manages different sets of configurable units. The first experiment has two CUs, issue queue and reorder buffer. The second experiment uses three CUs, level-one data and instruction caches, and level-two cache. Then each CU's adaptation efficiency retained by the 5-CU system (Section 5.2.2) over the 2/3-CU system is computed by dividing the CU's energy reduction in the 5-CU system with the corresponding result in the 2/3-CU system. To allow a fair comparison of the experiments, the IPC degradation thresholds are 2% and 3% respectively, essentially each CU with a 1% IPC loss budget.

5.3.1 Adaptation of issue queue and reorder buffer

In this section, the hot spot approach adapts two CUs, issue queue and reorder buffer, at the boundaries of IQ-ROB hot spots. The IPC degradation threshold is 2%. The percentages of energy consumption of the two CUs reduced by the hot spot resource adaptation approach are presented in Figure 11. On average, the hot spot approach reduces the energy consumption of IQ and ROB by 39% and 34% respectively. Note that since issue queue and reorder buffer have the same reconfiguration interval size, CU decoupling is not utilized by the hot spot approach, and each hot spot needs to test 16 configurations.

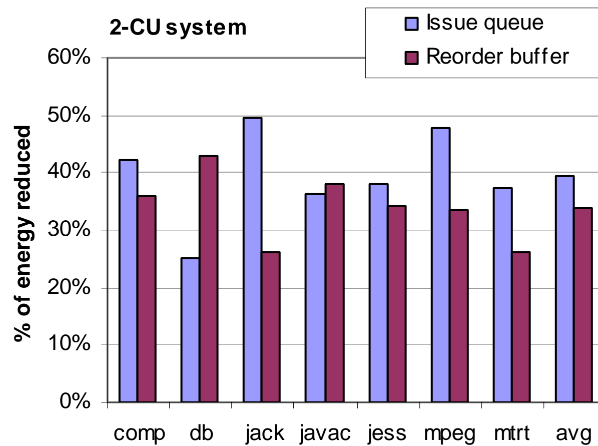

Figure 11. Reduction in energy consumption by adapting issue queue and reorder buffer (2% performance loss threshold).

In the 2-CU experiment, the IPC degradation threshold is 2%. Figure 12 presents the performance impact of the two approaches. Among the seven benchmarks, mpegaudio incurs the least performance loss of 0.8%. On average, the performance losses of the hot spot approach is 2.2%.

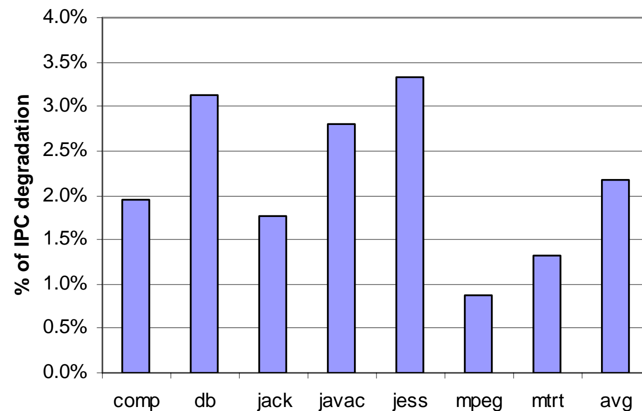

Figure 12. Performance degradation due to the adaptation of issue queue and reorder buffer.

5.3.2 Adaptation of L1D, L1I, and L2 caches

In this section, the two resource adaptation schemes are compared on adapting three configurable units: L1D, L1I, and L2 caches. Similar to the prior experiments, the hot spot approach also exceeds the BBV approach on achieving higher energy reduction on the CUs. The IPC degradation threshold for both approaches is 3%.

Figure 13 shows the portions of energy consumption reduced by the resource adaptation scheme for the three caches. The hot spot approach performs especially well on *db* with 58% L1D cache energy reduction. In *db*, less than 10 procedures are responsible for more than 95% of data cache misses [78]. Consequently, the average cache sizes can be considerably reduced for the hot spots that have very few data misses and good data locality. Moreover, a hot spot with poor data locality may also choose a smaller cache configuration if the cache size reduction does not significantly deteriorate the performance. On average, the hot spot approach achieves 42%, 37% and 48% energy reduction over the baseline configuration on the L1D, L1I, and L2 caches. The baseline configuration and the configurable units are listed in Table 1 and Table 2 respectively.

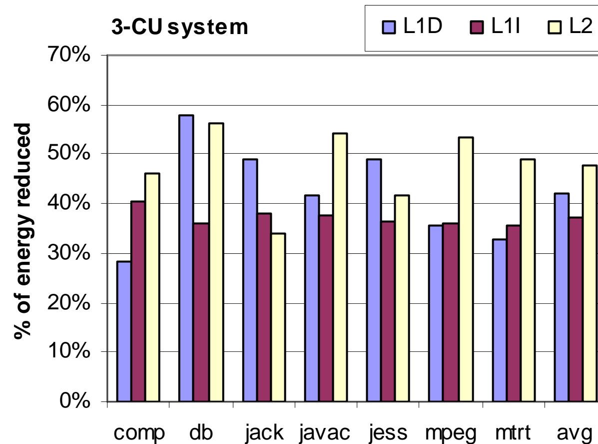

Figure 13. Reduction in energy consumption by adapting level-one and level-two caches (3% performance loss threshold).

Since this experiment has three CUs, the IPC degradation threshold is 3%. The performance impact of using the hot spot resource adaptation scheme is illustrated in Figure 14. For the hot spot technique the performance penalty ranges from 2.8% to 3.3%. On average the performance penalty for the hot spot technique is 2.9%.

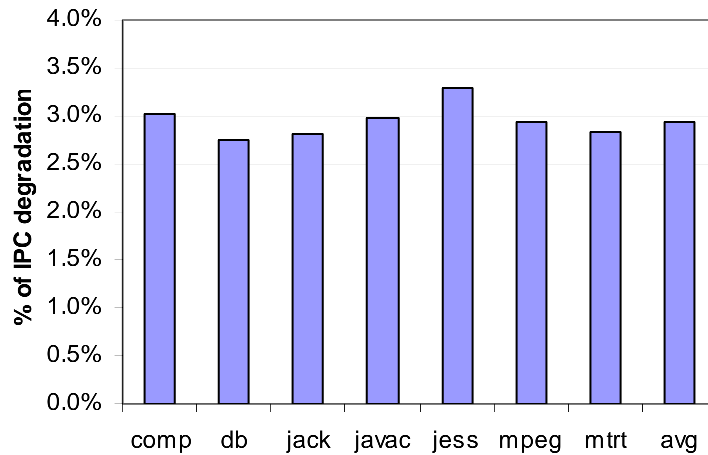

Figure 14. Performance degradation due to the adaptation of level-one and level-two caches.

5.3.3 Adaptation efficiency

A CU's adaptation efficiency indicates how well the CU's energy reduction is affected by the other CUs' adaptation, which can be obtained by comparing the CU's energy reduction results in systems with different number of CUs. The adaptation efficiencies of issue queue and reorder buffer are calculated by dividing the CU's energy reduction in the 5-CU system (Section 5.2.2) with the corresponding results in the 2-CU system (Section 5.3.1). For the three caches, their efficiencies are calculated by dividing the CUs' energy reduction rates in the 5-CU system (Section 5.2.2) with the

corresponding results in the 3-CU system (Section 5.3.2). The results are presented in Table 8.

Table 8. Adaptation efficiency retained by the 5-CU system versus the 2/3-CU system.

	compress	**db**	**jack**	**javac**	**jess**	**mpegaudio**	**mtrt**	**avg**
IQ	79.3%	70.3%	86.1%	83.9%	83.3%	85.6%	82.9%	81.6%
ROB	79.5%	77.5%	82.2%	85.3%	84.0%	84.0%	83.1%	82.2%
L1I	87.2%	82.7%	85.7%	91.6%	85.7%	88.7%	93.9%	87.9%
L1D	79.6%	82.4%	86.3%	79.2%	83.9%	84.2%	82.5%	82.6%
L2	95.1%	89.5%	87.1%	92.0%	95.9%	96.9%	93.9%	92.9%

For the hot spot approach, adding more CUs into either the 2-CU or the 3-CU system does not affect the tuning latency of the existing CUs. Hence, the drop of a CU' adaptation efficiency is mainly due to the interference of other CUs' configuration changes on the CU' tuning. For instance, the change of a CU's configuration may render another CU's previous most energy-efficient configuration sub-optimal, or make it incur too much performance loss. In both cases, the second CU needs to be re-tuned. In this respect, the adaptation efficiencies for the hot spot approach in Table 8 measure the interference between IQ/ROB and the caches.

The three caches have higher adaptation efficiencies than issue queue and reorder buffer. L2 cache has the highest adaptation efficiencies among the five configurable units. The results indicate that IQ and ROB are more likely to be interfered by the caches, and L2 cache is insensitive to other CUs configuration changes.

A better understanding of the interference among CUs' adaptation may assist the design of more sophisticated tuning-reduction strategies that can further improve the efficiency of adapting multiple CUs, which is studied in the next chapter.

5.3 SUMMARY OF ADVANTAGES

This chapter compares the two resource adaptation approaches in three experiments that manage various numbers of CUs. The results clearly demonstrate that with its ability to accurately capture phases with diverse granularity, the DO-based scheme is more efficient than the BBV approach on managing multiple CUs to achieve high overall energy reduction.

Comparing with hardware-based or static hardware adaptation schemes, the DO-based framework has its advantages as well as limitations. By leverage the existing hot spot detection and optimization infrastructure of a dynamic optimization, the DO-based framework requires much simpler hardware support than hardware-based schemes, or less human-intervention than static approaches. As a software approach, the framework can be easily tuned to achieve high energy efficiency for a wide range of adaptive execution environments, and all applications executed by the framework can enjoy the benefits without the painful work of tuning each application for each hardware platform upon which it will execute. On the other hand, statically compiled programs, such as C/C++ programs, cannot benefit from this DO-based approach. Nevertheless, exemplified by Java virtual machines and Microsoft .NET, dynamic optimization systems become increasingly popular, and more applications, especially server and commercial ones, are executed on DO systems. Those existing DO systems can utilize our framework for better hardware/software integration and optimizations.

In summary, utilizing existing DO components, this framework incurs minimal overhead while providing accurate phase detection and configuration tuning. Reconfiguring at hot spot boundaries identified by DO systems has the following advantages:

- **Prompt recurring phase identification.** By instrumenting hot spot headers, the framework can identify all previously seen hot spots with zero latency, and thus needs no phase prediction at all.
- **Reduced tuning latency.** Since each hot spot configures only a subset of CUs, the tuning latency is greatly reduced.
- **Adapting to hierarchical phase changes.** By detecting nested and multi-grain hot spots, tuning and reconfiguring CUs at those hot spots' boundaries accurately adapts to hierarchical phase changes.
- **Differentiating low-overhead and high-overhead CUs.** Because of CU decoupling, reconfiguration of CUs with different reconfiguration overhead occurs at different time intervals. Hence, low-overhead CUs are adapted more frequently than high-overhead CUs.
- **Versatility and scalability.** By detecting hot spots of any sizes, this approach works efficiently for workloads with diverse runtime characteristics and CUs with disparate reconfiguration overheads.

5.4 DISCUSSION

Currently, the DO-based hardware adaptation framework manages only configurable units. It can further improve a system's energy efficiency by incorporating other energy reduction techniques. Recently, Wu et al. [83] propose a lightweight dynamic compilation framework for dynamic voltage and frequency scaling (DVFS). This framework does not include the JIT optimization or garbage collection capability. It detects hot code regions that are memory-bound, and inserts DVFS mode set instructions at the boundaries of those regions so that they can be executed in energy-efficient modes. This mechanism can be implemented in the DO-based hardware adaptation framework,

which provides the opportunities to combine both DVFS and hardware adaptation for higher energy reduction.

Two principal sources of power dissipated by today's microprocessors are dynamic and static power. Dynamic power arises from the repeated capacitance charge and discharge of transistors. Static power is current leaking through transistors. Since shrinking processor technology reduces dynamic power, but exacerbates leakage, static power is becoming an increasingly larger component of power. Due to the limitation of the underlying power model, this work examines only the dynamic power reduced by hardware adaptation. Nevertheless, since each configurable unit can shut off the unused parts to reduce static power, the DO-based hardware adaptation framework is inherently effective on reducing static power also.

In a future system, each configurable unit will have more configurations, due to both technology and performance reasons. First, advances in the semiconductor technology constantly shrink devices geometries. In those systems, microprocessor performance is increasingly affected by local line delay instead of circuit delay. For performance improvement, buffers are used to reduce the local line delay, which is a natural way to partition hardware units and provides multiple configurations. In the future, more buffers will be used to partition hardware units in finer granularities and provides more configurations for each CU. Second, a hardware unit with more configurations will be able to precisely match its configuration with program runtime requirements, which reduces both adaptation-related performance loss and energy consumption. However, the increase of each CU's available configurations explodes the adaptive microarchitecture' configuration space, rendering the search of the globally best configuration more costly. Hence, for such a future system, configuration space explosion is one of the most important issues that a hardware adaptation scheme must deal with.

Another challenge for the hardware adaptation framework is to fully utilize the available fine-grain configurations to improve energy efficiency, which requires accurate detection of program runtime requirements and adjusts hardware configuration to fine grain program changes.

A DO-based hardware adaptation framework has certain features to help it handle those challenges effectively. First, the CU decoupling technique can manages multiple CUs efficiently. To prune the explosive tuning space, the DO-based framework can partition the hot spots size ranges into more groups so that the number of configurations each hot spot tested does not increase dramatically. Moreover, the DO system possesses the capabilities, such as runtime code analysis, to predict the best configuration or avoiding testing unpromising configurations, which further reduces the tuning space. Finally, with the runtime profiling infrastructure, a DO-based framework can accurately detects behavioral changes of hot spots, and re-tune the hot spots when necessary. This capability allows it to detect runtime characteristic changes within hot spots and adjust configurable units accordingly.

Chapter 6. Analysis of Factors that Interfere with Adaptation

A better understanding of a multi-CU environment's tuning process and the factors affecting the CUs' tuning decisions may aid the reduction of the tuning process and improve the efficiency of hardware adaptation. For example, if two CUs' tuning does not interfere with each other, those two CUs can be adapted individually. This can significantly reduce the number of tests for adapting the two CUs. This chapter studies two of the most important adaptation interfering factors, i.e., runtime program characteristics and interference of multiple CUs' adaptation.

6.1 Impact of runtime characteristics on CUs' tuning decisions

During a CU's tuning, the choice of its best configuration for a phase is determined by both the phase's runtime requirements, in terms of measurable characteristics, and the CU's negotiation with other configurable units to share an overall performance loss budget. In this section, the impact of runtime requirements/characteristics on CUs' tuning decisions is analyzed. The motivation of this study is to find out whether it is possible to accurately predict a CU's best configuration via runtime characteristics.

6.1.1 Impact measurement

The *impact intensity* of a runtime characteristic on a CU represents the impact of a phase's runtime characteristic value on its adaptation choice for the CU. It is measured in the following steps.

1) Each phase's best configuration for a CU is obtained with all other units keeping their largest configurations throughout program execution.

2) The mean values of various runtime characteristics are obtained for each phase. To avoid the interference of hardware configuration changes on detected characteristic values, the characteristic results are gathered with all hardware units retaining their largest configurations throughout program execution.
3) Phases are then classified into groups by their tuning decisions on the CU. For each phase group, the arithmetic means of the runtime characteristics of the phases in the group are calculated. The impact intensities are calculated by normalizing the means to the ones corresponding to the CU's smallest configuration. Those results indicate the trend of the mean characteristic values obtained in the baseline system configuration varies according to the tuning decisions.

Two types of trends between a characteristic and a CU can be potentially utilized to predict the CU's tuning decisions from the characteristic values. In the first type of trend, the characteristic's impact intensities, gathered in the steps described above, increase or decrease monotonically. The second type of trend exists when a mean value for a given CU configuration is much larger or smaller than the ones for other CU configurations, implying that the characteristic weighs more than other characteristics on choosing the configuration. Hence, in the next section, the impact analysis is conducted by identifying the two types of trends between runtime characteristics and CUs.

6.1.2 Impact intensity analysis

The impact intensities of runtime characteristics on CUs are presented in Figure 15 to Figure 19. The results are averaged over all seven benchmarks. For each characteristic, each of the four resulting bars represents the average impact intensity of all the phases that choose the same configuration, with the first bar always being one. The CU's configurations are presented with values 1 to 4, with 1 and 4 being the smallest and the largest ones respectively, and the smallest size is one quarter of the largest one. The

characteristics examined in this section are IPC, L1D/L1I/ L2 misses (L1D/L1I/L2_miss), branch misses (br_miss) and IQ and ROB occupancies (IQ/ROB_occu). For simplicity, the results in Figure 15 to Figure 19 do not include characteristics, such as IFQ occupancies and TLB misses, that rarely affect CUs' tuning decisions.

6.1.2.1 Reorder buffer and issue queue

Comparing Figure 15 with Figure 16, one notes that programs have very similar requirements/characteristics on the issue queue and the reorder buffer, owing to their close cooperation in extracting instruction-level parallelism (ILP) from programs. Hence, the two CUs are investigated together.

First, the impact intensities of L1D/L2 misses to IQ/ROB configuration_4 are higher than other values, implying that when a phase has many L1D/L2 misses, it is more likely to choose the largest IQ/ROB configuration. With abundant L1D/L2 misses, the missed loads/stores and their dependent instructions occupy the ROB and the IQ respectively, leaving fewer IQ/ROB entries for ILP extraction. With high IQ/ROB occupancies, using smaller IQ/ROB will inevitably hurt their ability to extract ILP, and impairs performance. Consequently, a phase with many L1D/L2 misses tends to use large IQ/ROB. On the other hand, when a phase has fewer L1D/L2 misses, the phase's tuning decisions on IQ/ROB are affected by more program requirements/characteristics other than L1D/L2 misses.

The tuning decisions of IQ/ROB increases monotonically with IQ/ROB occupancies, indicating that statistically, the IQ/ROB occupancies are closely related with the tuning decisions of IQ/ROB. Although not presented in the figures, our results show that a phase's ROB occupancy is almost always larger than its IQ occupancy.

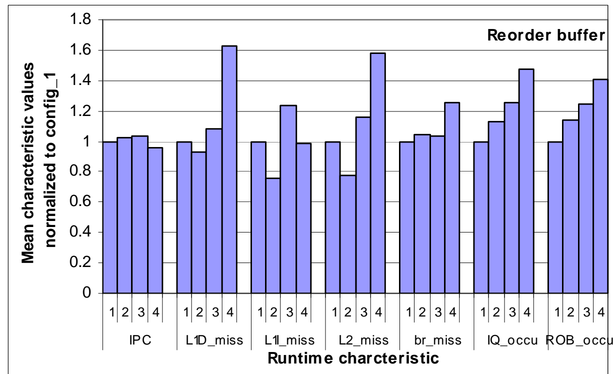

Figure 15. Runtime characteristics' impact on reorder buffer's adaptation. (1: 32-entry ROB; 2: 64-entry ROB; 3: 96-entry ROB; 4: 128-entry ROB)

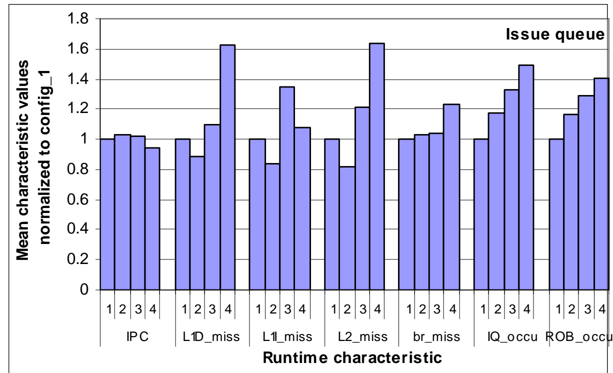

Figure 16. Runtime characteristics' impact on issue queue's adaptation. (1: 32-entry IQ; 2: 64-entry IQ; 3: 96-entry IQ; 4: 128-entry IQ)

6.1.2.2 Level-one data cache

The impact intensities for the level-one data cache are in Figure 17. First, phases with large amount of L1D misses and branch misses tend to select the largest L1D configuration. If a phase has fewer L1D and branch misses, having fewer L2 misses may lead the phase to choose the most aggressive L1D configuration. Finally, the branch misses exhibits the monotonic property to the L1D cache. However, the differences

between the branch misses' impact intensities, excluding the one for configuration_4, are very small, less than 0.2, indicating that the impact of branch misses on L1D cache, except on configuration_4, is not strong enough for prediction. Interestingly, L1D misses does not exhibit the monotonic property for the L1D cache, although the amount of L1D misses characterizes the L1D cache.

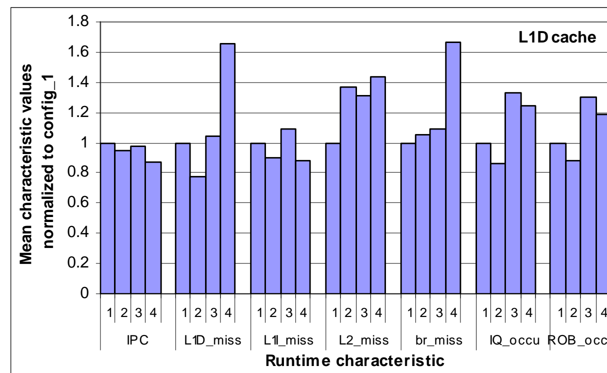

Figure 17. Runtime characteristics' impact on L1D cache's adaptation. (1: 16K L1D cache; 2: 32K L1D cache; 3: 48K L1D cache; 4: 64K L1D cache)

6.1.2.3 Level-one instruction cache

The impact intensities for the L1I cache are illustrated in Figure 18. For the L1I configuration_4, the impact intensity for L1I misses is 5.2, much larger than all the other values, indicating that if a phase has a large amount of L1I misses, it is highly likely to adapt to the largest L1I configuration. Similarly, phases adapted to L1I configuration_4 also have more branch misses than other phases. Large amount of branch misses indicate that the program code has poor instruction locality, and is one of the primary causes for the increase of L1I misses. Finally, no characteristic exhibits the monotonic trend for the L1I cache.

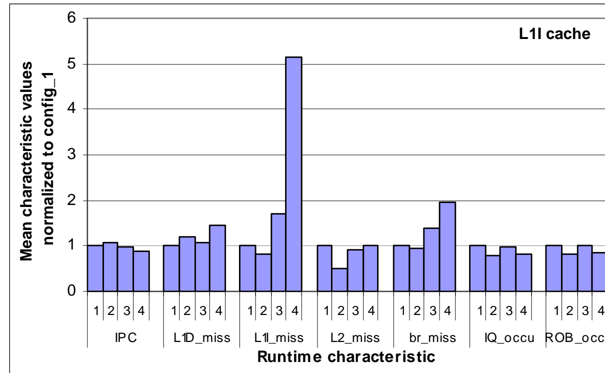

Figure 18. Runtime characteristics' impact on L1I cache's adaptation. (1: 16K L1I cache; 2: 32K L1I cache; 3: 48K L1I cache; 4: 64K L1I cache)

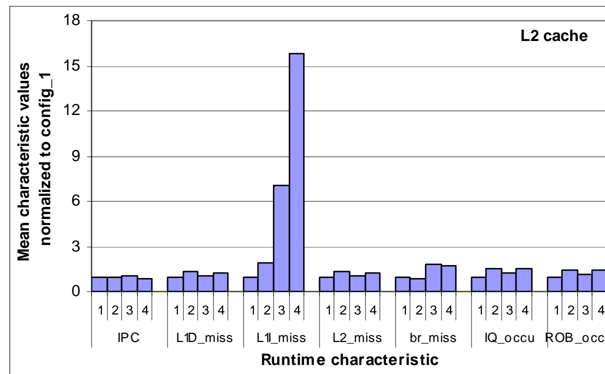

Figure 19. Runtime characteristics' impact on L2 cache's adaptation. (1: 256K L2 cache; 2: 512K L2 cache; 3: 768K L2 cache; 4: 1M L2 cache)

6.1.2.4 Level-two cache

Figure 19 gives the results for the L2 cache. The impact intensities of L1I misses increases monotonically and are much higher than other characteristics. The results imply that statically, the amount of L1I misses determines the L2 cache's adaptation. This is mainly due to the fact that the performance is more sensitive to instruction misses, which, unlike data misses, cannot be hidden via techniques such as out-of-order execution.

Reducing L2 cache's size renders more L2 accesses, some being L1I misses, to be misses, which, in turn, considerably increases the latency to fetch a missed instruction, and thus impairs the performance. Consequently, L1I misses determine the L2 cache's tuning choices.

6.1.3 Implications to tuning process reduction

First, IPC rarely correlates with the tuning decisions of the CUs studied in this paper, although it is used in a previous scheme [7] to decide the proper pipeline width. IPC does not exhibit the two types of trends with any CU. A characteristic affecting a CU's tuning decision should closely represent program's requirements on the CU. However, as a composite runtime characteristic, IPC reflects multiple program requirements that may conflict on a CU's tuning decisions. Hence, it is inaccurate to predict the CUs' tuning decisions based on IPC values.

In the above section, several characteristic/CU pairs possess the monotonic trend. Nevertheless, this type of trend exists statistically. To be able to accurately predict a phase's best configuration, the observed trend must be validated for each phase. To examine whether this premise is true, Table 9 measures the coefficients of variations of the characteristic/CU pairs. In the table all monotonic pairs have very high CoVs, indicating that for most phases, it is highly possible that the values of those characteristics vary a lot from their means. Consequently, for all the CUs studied in this work, it will be inaccurate to predict a CU's best configuration based on the value of a runtime characteristic. Nevertheless, it is still possible to predict a CU's best configuration based on the values of multiple characteristics, which is a suitable topic for future study.

Statically, some characteristics are good indicators for CUs' choosing of their largest configurations. The examples are L1I misses for the L1I and L2 caches, and L1D

and branch misses for the L1D cache. This property can be utilized to avoid the tunings for some phases. For instance, if a new phase is found to have a lot of L1I misses, the tuning of L1I and L2 caches for the phase can be avoided and the largest L1I/L2 configurations can be directly chosen for the phase. The results indicate that the phenomenon holds for all the benchmarks evaluated. On the other hand, the thresholds for those characteristics vary by programs, and need to be finely tuned to achieve better phase coverage and avoid false positive.

Table 9. Coefficients of variations for the characteristic/CU pairs that possess the monotonic property.

Characteristic/CU pair	**config_1**	**config_2**	**config_3**	**config_4**
IQ occupancy/IQ	102%	75%	68%	74%
ROB occupancy/IQ	100%	74%	68%	74%
IQ occupancy/ROB	83%	62%	57%	64%
ROB occupancy/ROB	81%	61%	58%	64%
L1I misses/L1I cache	160%	122%	105%	97%
L1I misses/L2 cache	149%	123%	55%	101%

This research shows that it is usually inaccurate to predict a hardware unit's best configuration based on a runtime characteristic. The reason is that a runtime characteristic is usually configuration dependent, and only indicates how well the current hardware configuration performs, but rarely indicates how well other configurations should work. For instance, given a phase with a 60K data working set, a 64K data cache should perform very well since it can hold all the data. On the other hand, reducing the data cache size beyond 60K will results in a surge of capacity misses. Hence, the number of data misses to the 64K data cache, by themselves, provides no hint to the best cache configuration for the phase. Nevertheless, as the above example shows, examining the

changes of the runtime characteristics for different configurations can help find the best configuration, which is exactly what a tuning algorithm does.

6.2 INTERFERENCE BETWEEN CONFIGURABLE UNITS

In a multi-CU environment, CUs interfere with each other during their tunings, reflecting the adjustment of each CU's configurations to maximize overall energy saving while retaining an overall performance loss. For simplicity, the interference between two units' tuning is called *the interference between the two units*. In this section, the interference between any two of the five configurable hardware units (presented in Table 2) is investigated.

6.2.1 Interference distance measurement

Multiple sets of experiments are conducted to quantify the interference between any two CUs. Each set of experiments examines the interference of one unit (*the interfering unit*) on the tuning decisions of another CU (*the interfered unit*). Only the interfered CU is adapted on the fly, while all other units keep their configurations throughout program execution. The interfering unit changes its configuration from experiment to experiment, while all other units stay constant using the largest configurations.

The *interference distance* between two CUs is measured in the following steps.

1) For each pair of CUs, three experiments are conducted for a benchmark, with the interfering unit varying its size to full, a half, and a quarter of its maximal size.
2) Each experiment obtains the best configuration (*BC*) array that contains all phases' best configurations for the CU, with each array element valued between 1 and 4. The BC array obtained with the largest interfering unit configuration is the baseline one. The difference between two arrays indicates the changes of phases' best

configurations as the configuration of the interfering unit varies, and represents the interference imposed by the interfering unit on the interfered unit.

3) The *interference distance* between the baseline array and a non-baseline array is represented by the arrays' normalized Manhattan distance, i.e., the sum of the absolute differences of the corresponding array entry values divided by the number of entries. Consequently, a large interference distance indicates that the CU is more likely to be affected by the interfering unit's setting, while a zero distance means that the CU is not interfered by the other unit.

6.2.2 Interference distances between CUs

The interference distances between any two of the five configurable units are illustrated in Figure 20 to Figure 25. Each figure, corresponding to a CU, contains four groups of results that represent the four other units. Each group consists of two subgroups of distances that represent the Manhattan distances between the CU's half-size/quarter-size BC arrays the full-size BC array. Finally, each subgroup consists of eight bars that represent the seven SPECjvm98 benchmarks and their arithmetic mean respectively.

6.2.2.1 Reorder buffer versus other units

Issue queue has the largest interference distances to the reorder buffer, demonstrating that IQ extensively interferes with ROB's tuning. This is mainly because the size reduction of the IQ impairs its ability to extract ILP and slows down the pipeline. As a result, a phase can adapt to a smaller reorder buffer with minimal extra performance loss.

Among the three caches, the L1D and the L2 caches interfere with the reorder buffer's tuning more than the L1I cache. Both the L1D and the L2 caches affect the ROB's tuning decisions by occupying the ROB with missed loads/stores. Two factors

determine the ROB occupancies: the number of missed loads/stores and their lifetimes in the ROB. Although a program has fewer L2 misses than L1D misses, L2 misses take much longer (100s cycles) to fetch than L1D misses (10s cycles). To measure those two conflicting factors' impact on the ROB, the average ROB occupancies are counted and normalized to the ones obtained with the caches' corresponding baseline configurations. The results are presented in Figure 21, and indicate that ROB occupancy changes owing to L2 cache size reduction is more prominent than those due to L1D cache size reduction. In contrast, owing to the good instruction locality, L1I cache's size reduction rarely incurs a dramatic increase of instruction misses that significantly affect the adaptation of ROB. Hence, the interference of the L1I cache on the ROB is the smallest among the four units.

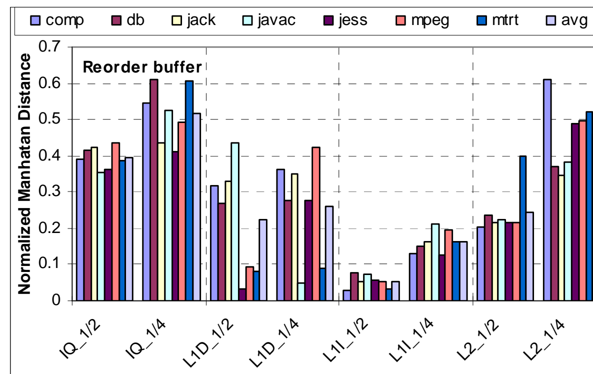

Figure 20. Hardware units' interference on reorder buffer's adaptation.

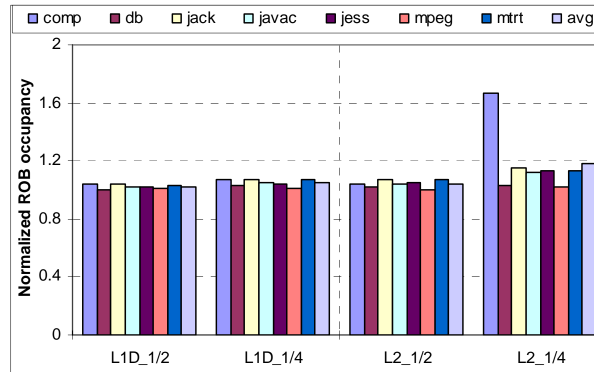

Figure 21. Normalized ROB occupancy.

6.2.2.2 Issue queue versus other units

Figure 22 shows the interference of the other four hardware units on the issue queue. Similar to the observation in the prior subsection, the reorder buffer interferes with the issue queue's adaptation, and prevails other units as the reorder buffer size reduces to one quarter of its maximal size.

The L1D and the L2 caches interfere with the reorder buffer's tuning more than the L1I cache. The L1D and the L2 caches interfere with the adaptation of the IQ via occupying the IQ with instructions that depend on the missed loads/stores. Note that those loads/stores do not stay in the IQ since they are already dispatched, but are still in the ROB since they are not committed yet.

Although the L1I cache has the smallest interference distances on IQ's tuning than other units, its impact on IQ is larger than its impact on ROB. One major reason for the difference is that being closer to the L1I cache, the IQ can absorb some of the performance impact caused by L1I cache's size reduction before it affects the tuning of the ROB.

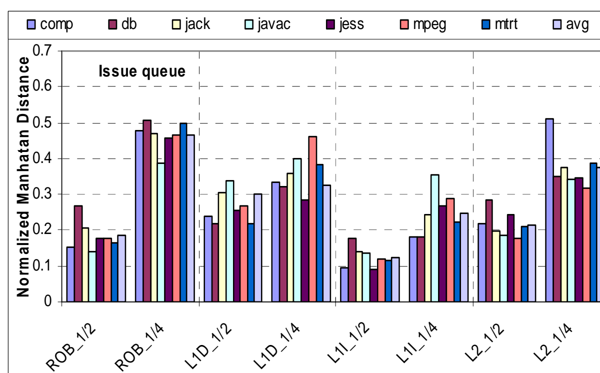

Figure 22. Hardware units' interference on issue queue's adaptation.

6.2.2.3 Level-one data cache versus other units

Figure 23 shows the interference distances of the hardware units on the level-one data cache. As shown in the figure, the L2 cache interferes with the L1D cache's adaptation extensively, which, to some extent, validates the observation in Section 4.2.4 that the L2 cache's adaptation is sensitive to the amount of L1 misses. In contrast, the L1I cache's size reduction rarely interferes with the L1D cache's adaptation. Residing at the two end of the pipeline, the L1I cache's performance impact is absorbed by the execution energy before it affects the L1D cache's adaptation. Finally, the size-reductions of both the IQ and the ROB affect the adaptation of the L1D cache via slowing down the pipeline. Consequently, a smaller L1D cache can be used with minimal extra performance loss.

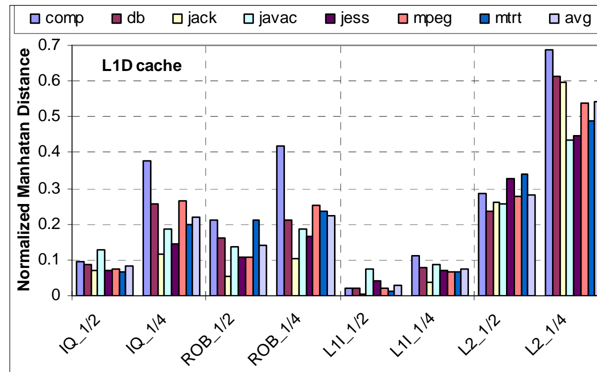

Figure 23. Hardware units' interference on L1D cache's adaptation.

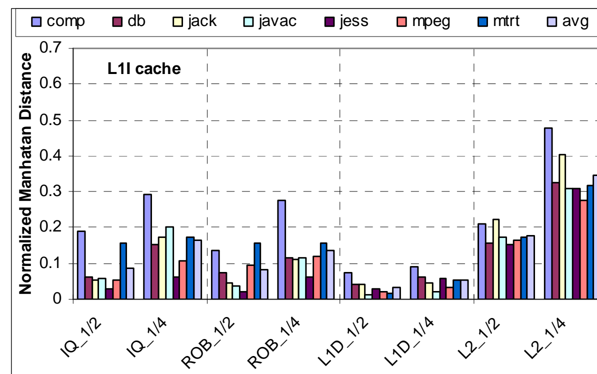

Figure 24. Hardware units' interference on L1I cache's adaptation.

6.2.2.4 Level-one instruction cache versus other units

Figure 24 provides the interference distances between the L1I cache and the other four hardware components. The adaptation of the L1I cache is more likely to be interfered by the L2 cache than other units. Comparing with those in Figure 23, the other three units' interference distances on the L1D cache are larger than on the L1I cache, implying that those units' interference on the L1D cache is more intense than on the L1I cache. Unable to affect L1I cache's performance, L1D cache rarely interferes with the adaptation of the L1I cache.

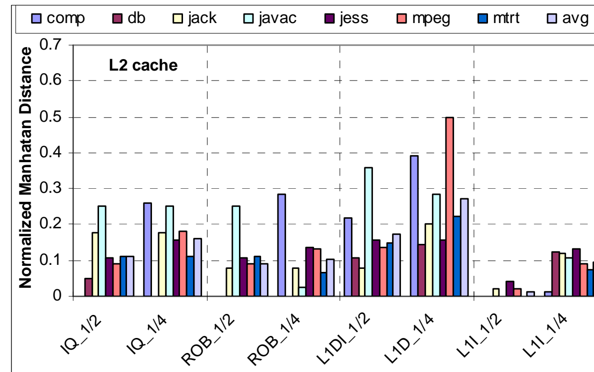

Figure 25. Hardware units' interference on L2 cache's adaptation.

6.2.2.5 Level-two cache versus other units

The interference distances between the level-two cache and the other hardware components are presented in Figure 25. According to the figure, the L2 cache's adaptation is more likely to be interfered by the L1D cache than the other units. In contrast, when the L1I cache size is reduced to the half of its maximal size, the L2 cache's adaptation is rarely affected. Finally, the interference of both the IQ and the ROB on the adaptation of the L2 cache varies by benchmarks.

6.2.3 Implications to tuning process reduction

Two trends on the interference between CUs partly validate the efficiency of multi-grain adaptation employed by the DO-based hardware adaptation framework proposed in Chapter 3. First, units in the execution engine, such as IQ and ROB, are more sensitive to other units' adaptation than those outside the execution engine, such as the caches. The units in the execution engine have lower reconfiguration overhead than those outside the execution engine. Furthermore, the interference of the L2 cache on the L1D

and the L1I caches are much larger than the interference of the level-one caches on the L2 cache. Hence, high-overhead CUs are less sensitive to other units' interference than low-overhead CUs, and their adaptation can be decoupled from low-overhead CUs.

Representing only the intensities, the interference distances presented in Section 6.2.2 do not fully characterize the interference between two CUs. It is also interesting to know how the size reduction of the interfering unit affects the interfered CU's configuration selection. Two types of interference between CUs are of special interest since they may be used to reduce the tuning space exploration. The interference between two CUs is *positive* when the size reduction of one CU relieves the performance pressure on the other one and allows the later to use the same or a smaller configuration with minimal performance loss. The interference between two CUs is *negative* when the size reduction of one CU increases the performance pressure on the other one and prompts the later to use the same or a larger configuration to prevent an unacceptable performance loss. The interference results for the benchmarks indicate that the interference between L1D and L2 caches is negative, while the interference between IQ and ROB is positive. Knowing the intensities and types of interference between CUs can aid the reduction of the tuning process, which is discussed in the next section.

The interaction between hardware units is complex. Besides positive and negative interference, there could be other types of interference. An extensive examination of all those interference types helps us gain a better understanding of the interaction between CUs' adaptation, and may lead to novice strategies to prune the tuning space.

6.3 Reduction of multi-CU tuning process

Although statistically, runtime characteristics may closely relate with some CU's tuning decisions, Section 6.1 shows that for the CUs examined in this work, it will be inaccurate to predict a CU's best configurations based on the value of a runtime

characteristic. On the other hand, knowing the intensities, i.e., the Manhattan distances between best configuration arrays, and types of interference between CUs can aid the reduction of the tuning process. In this section, two tuning reduction strategies are proposed based on the above findings.

6.3.1 Tuning reduction strategies

The hardware adaptation framework proposed in Chapter 3 improves the efficiency of managing multiple CUs by allowing multi-grain adaptation. With CU decoupling, CUs with different reconfiguration intervals can be tuned at different hot spot boundaries. On the other hand, for the CUs with the same reconfiguration interval sizes, the framework still needs to test all the combinatorial configurations of the CUs. Among the five CUs studied in this work, the issue queue and the reorder buffer have the same reconfiguration interval size are adapted at the boundaries of the same group of hot spots, and so do the L1D and the L1I caches. Hence, to further improve the framework's efficiency, strategies that can reduce the tuning of the CUs with identical reconfiguration interval sizes are needed. In this section, two such schemes, based on the findings in Section 6.1 and Section 6.2, are proposed.

6.3.1.1 Tuning the level-one caches

The minimal interference between the two caches implies that they can be adapted individually (Section 6.2.2.3 and 6.2.2.4). Hence, it is possible to reduce the latency of adapting those two caches. Initially, the most energy consuming combinatorial configuration (i.e., 3_3, the two numbers represent the two caches' configurations respectively) is tested first to obtain the baseline performance. Then the twin combinatorial ones (i.e., 2_2, 1_1, 0_0) are tested gradually (indicated by the solid lines in Figure 26(a)) until the most aggressive one (0_0) is reached or the performance

degradation is too large. In the first case, the configuration 0_0 is the best configuration for the tuned phase. Otherwise, one can find whether the performance degradation is caused by the L1D cache, the L1I cache, or both by examining the changes of L1D and L1I cache misses. If the performance is caused by both caches, the tuning processes finishes. Otherwise, the performance-impairing cache keeps its prior configuration, while the other cache continue tests its smaller configurations until the performance degradation is reached or all of its configurations are tested (indicated by the dotted lines in Figure 26(a)). Then, the last tested combinational configuration within the performance threshold is the most energy efficient one for the phase. This strategy takes up to five, instead of sixteen tests, to adapt the two level-one caches.

Figure 26. The tuning reduction strategies.

6.3.1.2 Tuning the issue queue and the reorder buffer

The interference between IQ and ROB is extensive (Section 6.2.2.1 and 6.2.2.2), and cannot be tuned separately as the level-one caches. On the other hand, the interference type between IQ and ROB is positive, i.e., when one uses a smaller configuration, the other one can use the same or a smaller configuration with minimal

performance losses (Section 6.2.3). Another observation is that the ROB's size should be the same as the IQ's size, or larger (Section 6.1.2.1). These two findings inspire us to design the following reduced tuning strategy (shown in Figure 26 (b)).

In the reduced tuning process, the most energy consuming combinatorial configuration (i.e., 3_3) is tested first to obtain the baseline performance. Then the twin combinatorial ones (i.e., 2_2, 1_1, 0_0) are tested gradually (indicated by the solid lines in Figure 26 (b)) until the most aggressive one (0_0) is reached or the performance degradation is too large. In the first case, the configuration 0_0 is the best configuration for the tuned phase. In the second case, the combinatorial configurations in the same row as the last tested twin configuration are tested gradually (indicated by the dotted lines in Figure 26 (b)), and one with acceptable performance degradation and the highest energy saving is the best one. The configurations in the same column as the last tested twin configuration need not to be tested since they represent the configurations that IQ is larger than ROB. Consequently, the tuning process for the IQ and the ROB is dramatically reduced.

6.3.2 Evaluation of the tuning reduction strategies

With the above two tuning reduction strategies incorporated in the hardware adaptation framework, they are evaluated by adapting all the five configurable units using the framework, and the target performance loss is 5%. To The results are given in Figure 27 to Figure 31. As a comparison, the energy and performance results of the hardware adaptation scheme without those two tuning reduction strategies are presented as baseline results. They are the same hot spot results in Figure 4 to Figure 8. For each benchmark, the two bars correspond to the percentages of energy reductions with (tuning-reduction) and without (baseline) the tuning reduction strategies. On average, using the tuning reduction strategies reduce the energy of issue queue, reorder buffer, L1D cache, L1I

cache, and L2 cache by 39%, 33%, 42%, 35% and 45%, respectively. Differing to the other CUs, the L1I cache's energy reductions are similar for all the benchmarks, which reveals that the instructions streams of those benchmarks have similar characteristics due to the extensive virtual machine activities.

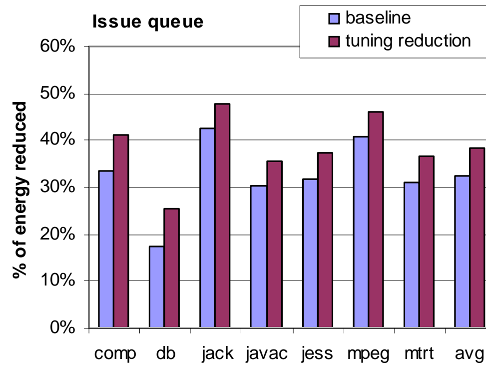

Figure 27. Reduction in issue queue energy reduction by using the tuning reduction strategies.

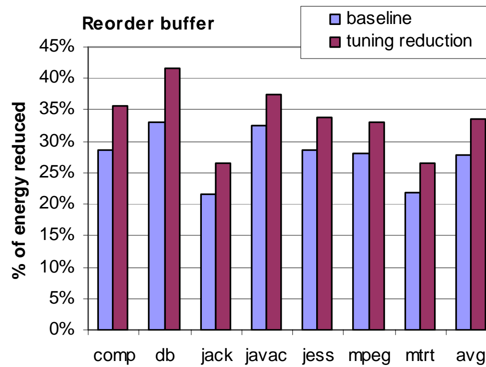

Figure 28. Reduction in reorder buffer energy reduction by using the tuning reduction strategies.

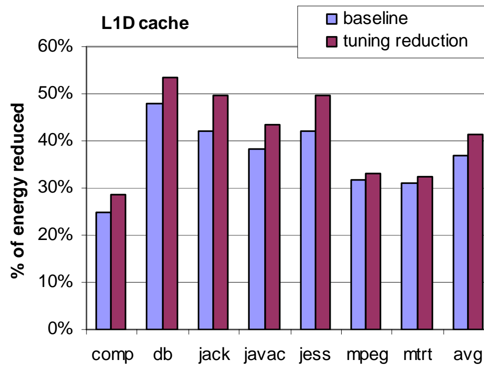

Figure 29. Reduction in L1D cache energy reduction by using the tuning reduction strategies.

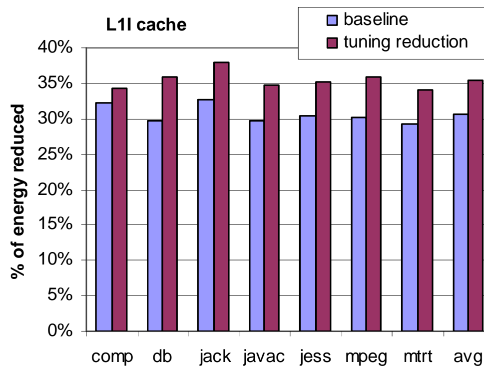

Figure 30. Impact of the tuning-reduction strategies on L1I cache energy reduction.

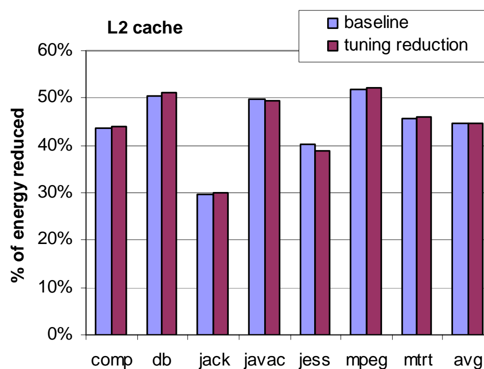

Figure 31. Impact of the tuning-reduction strategies on L2 cache energy reduction.

Without the two tuning reduction strategies presented in this work, the DO-based hardware adaptation scheme reduces issue queue, reorder buffer, L1D cache, L1I cache, and the L2 cache's energy consumption by 33%, 28%, 37%, 31% and 45% respectively. Hence, incorporating the two tuning reduction strategies in the DO-based hardware adaptation framework further reduces the energy consumed by IQ, ROB, L1D and L1I caches.

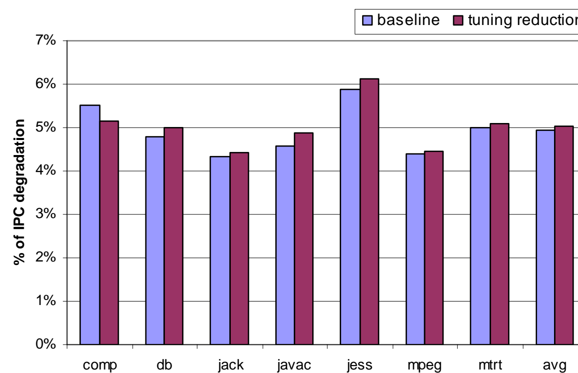

Figure 32. Performance degradation due to the tuning reduction strategies.

The performance degradations of the resource adaptation are illustrated in Figure 32 as well. The energy reduction achieved by adapting a CU is usually at the cost of performance, in terms of IPC degradation. In the experiments, the simulation environment is tuned to achieve the target IPC degradation of 5%. The performance degradation seen by the BBV technique ranges from 4.5% (*mpegaudio*) to 6.1% (*jess*). On average the performance penalty for the hot spot technique is 5.0%. In comparison, without the tuning reduction strategies, the hardware adaptation framework causes a 4.9% performance drop on average. The results clearly demonstrate that despite the growing complexity on adapting multiple CUs, the two tuning reduction strategies further

improve the efficiency of the DO-based framework on managing multiple configurable units.

6.4 DISCUSSION

Hardware adaptation involves tradeoffs. First, the best configuration achieves the highest energy reduction for a given performance budget, and thus is always preferred. On the other hand, due to the explosion of the configuration space, traversing the configuration space to find the best one is costly. Hence, it is unreasonable for a real system to choose a tuning strategy that tests all combinational configurations. However, without doing so, a system usually detects sub-optimal configurations, impairing the system's energy efficiency.

This chapter demonstrates that with a better understanding of the interference between CUs, the tuning space can be significantly reduced without affecting the accuracy of the best configuration search. The interference analysis is conducted using SPECjvm 98 benchmarks. However, the study examines interactions between hardware units, and do not relying on any specific features of either the programs or Jikes RVM. Hence, the findings should be applicable to other applications and hardware adaptation schemes.

This research discovers three phenomena that can be used to prune the tuning space traversed by a tuning algorithm to find the best configuration. First, high-overhead configurable units (e.g., L2 cache) are less sensitive to other CU's adaptation than low-overhead CUs (e.g., issue queue). Hence, high-overhead CUs' adaptation can be decoupled from low-overhead CUs. Second, some CUs', such as L1I and L1D caches, have minimal mutual interference to each other's adaptation, and can be adapted in parallel. Finally, interference between some CUs (e.g., issue queue and reorder buffer) is coupled so that when one's size is reduced, the other either keeps its current

configuration or chooses a smaller one with minimal extra performance loss. Hence, during the traverse of their tuning space to search for the best configuration, those CUs bias towards certain paths. The tuning space is essentially pruned when the tuning algorithm focuses on only those promising paths. To utilize those properties in an adaptive microarchitecture, we need to obtain the information in advance.

Exploiting the first phenomenon, we can decouple the adaptation of CUs with different reconfiguration costs by associating them with phases of different granularities. Each phase adapts only those CUs with similar adaptation costs, which significantly reduces the tuning space.

Then, the tuning space of the CUs with similar adaptation costs can be pruned if the CUs are either independent or have positive interference. Two generalized tuning reductions algorithms are given in the following subsections.

6.4.1 Generalized tuning reduction algorithm for independent CUs

Figure 33 shows the tuning reduction algorithm for two or more independent CUs. The algorithm first evaluates the configuration with the highest energy consumption and performance to obtain the baseline IPC. Since the goal of hardware adaptation is to achieve high energy reduction with the performance budget, the allowed performance loss is subtracted from the baseline IPC to get the target IPC. The best configuration must yield an IPC equal to or larger than the target IPC to fulfill the performance constraint.

In Step 1 and 2, we test the configurations one by one, reducing the sizes of the tested CUs by one from their current sizes. Those two steps test those CUs in parallel since those CUs are independent. By doing so, the algorithm can quickly reach either the best configuration or one that incurs a performance loss exceeding the threshold. The current configuration is recognized as current_configuration.

0. The available CU list includes all CUs.
    a. Set current_configuration to the most energy consuming configuration, i.e., the one with each CU using its largest size.
    b. Record the baseline IPC.
1. Test the configuration with all the available CUs' sizes reduced by one from current_configuration, and set the new configuration as current_configuration. (E.g., if current current_configuration is (C1, C2, …, Cn), the new current_configuration is (C1-1, C2-1, …, Cn-1)).
2. Repeat 1 until performance threshold is exceeded.
3. Choose one CU from the available CU list. Test the configuration with this CU's size increased by one from current_configuration (e.g., (C1+1, C2, …, Cn) or (C1, C2+1, C3, …, Cn)) to find whether the CU that is responsible for the performance loss.
4. If the performance is within the threshold, the tested CU is removed from the available CU list. This CU keeps its current configuration afterwards. Go to 1.
5. Repeat 3 until all such configurations are tested.
6. Choose two CUs from the available CU list. Test the configuration with the CUs' sizes increased by one from Current_configuration (e.g., (C1+1, C2+1, …, Cn) or (C1, C2+1, C3+1, …, Cn)) to find any two units that is responsible for the performance loss.
7. If the performance is within the threshold, the tested CUs are removed from the available CU list. Those CUs keep their current configurations afterwards. Go to 1.
8. Repeat 6 until all combinations of two CUs are tested.
9. Repeat 6-8 to test incremental number of CUs to find the set of CUs that are responsible for the performance loss.
10. If all CUs are responsible for the performance loss, stop.

Figure 33. Generalized tuning reduction algorithms for independent CUs.

Steps 3 to 9 are used to identify the configurable units that cause the performance loss. This is performed by testing configurations in which each CU's configuration either remains the same, or increases by one from its configuration in current_configuration. In

those tests, the CUs whose sizes increase are evaluated to find out whether they cause the performance loss. Such tests are possible since those CUs are independent with each other and the performance loss is due to CUs' size reduction, instead of the interaction between CUs.

After one or more CUs are detected to be responsible for the performance loss, those CUs keep their configurations afterwards. Other CUs continue steps 1 and 9 to test configurations that further save energy until the best configuration is found.

0. Initially,
    a. Set Current_configuration to the most energy consuming configuration, i.e., the one with each CU using its largest size.
    b. Record the baseline IPC.
1. Quickly reach a combinatorial configuration that reduces energy and satisfies the performance constraint.
    a. Test the configuration with all the available CUs' sizes reduced by one from current_configuration, and set the new configuration as current_configuration. (E.g., if current current_configuration is (C1, C2, …, Cn), the new current_configuration is (C1-1, C2-1, …, Cn-1)).
    b. Repeat 2 until performance threshold is exceeded.
    c. Set current_configuration to the previous one that satisfies the performance constraint.
2. Fine-grain adjust each CU's configuration to achieve higher energy reduction
    a. Select one CU from all available CUs. Test the configuration with this CU's size decreased by one from Current_configuration (e.g., (C1-1, C2, …, Cn) or (C1, C2-1, C3, …, Cn)).
    b. If Current_IPC > Overall_Target_IPC, the current configuration is set as current_configuration; Otherwise, keep current current_configuration.
    c. Repeat 2.a until no CU changes its configuration.

Figure 34. Generalized tuning reduction algorithms for CUs with positive interference.

6.4.2 Generalized tuning reduction algorithm for CUs with positive interference

Figure 34 shows the tuning reduction algorithm for two or more CUs with positive interference. The algorithm first evaluates the configuration with the highest energy consumption and performance to obtain the baseline IPC. Since the goal of hardware adaptation is to achieve high energy reduction with the performance budget, the allowed performance loss is subtracted from the baseline IPC to get the target IPC. The best configuration must yield an IPC equal to or larger than the target IPC to fulfill the performance constraint.

In Step 1, we test the configurations one by one, reducing the sizes of all CUs by one from their current sizes. This step tests the CUs in parallel in accordance with the positive interference between the CUs. Testing those configurations allows the algorithm to quickly reach either the best configuration or one that incurs a performance loss surpassing the budget. The current configuration is recognized as current_configuration.

Starting from current_configuration, Step 2 searches locally to find the best performing configuration. Each test examines the configuration with one CU reducing its size by one from current_configuration while other CUs keeping their current sizes. If the new configuration still satisfies the performance constraint, it becomes the new current_configuration. Since the CUs interfere positively, the size reduction of one CU may prompt other CUs to use smaller sizes that are still satisfy the performance constraint. Hence, the tests continue until the size of no CU can be reduced without incurring a performance loss exceeding the threshold.

Chapter 7. The Role of JIT Optimization and Garbage Collection on Microprocessor Energy Consumption and Hardware Adaptation

The above two chapters demonstrate that efficient management of multiple configurable units is vital for maximizing energy reduction, and a dynamic optimization system's inherent capabilities of detecting and optimizing hot spots can be employed to efficiently manage multiple configurable units for energy reduction.

With the growing popularity of DO systems and importance to reduce surging microprocessor energy consumption, it is important to understand the impact of DO systems on programs' energy consumption and power dissipation. This chapter characterizes the energy and power impact of two most important services of DO systems, Just-in-time (JIT) optimization and garbage collection. Moreover, the interference of those two DO services on hardware adaptation is examined for five configurable hardware units: issue queue, reorder buffer, level-one instruction and data caches, and level-two unified cache.

This chapter also compares application, JIT optimizer and garbage collector on energy reduction and adaptation preferences. To identify the affiliation of dynamic code sequences, we instrument the code boundaries of the JIT optimizer and the garbage collector. The instrumentations are recognized when simulated in Dynamic Simplescalar, and indicate that the subsequent code sequences belong to either application, JIT optimizer, or garbage collector.

7.1 ENERGY IMPACT OF JIT OPTIMIZATION AND GARBAGE COLLECTION

The runtime characteristics of a program's *dynamic execution* (i.e., execution in a dynamic optimization system) vary significantly from its statically compiled counterpart due to the following reasons. First, optimizations used in offline and JIT compilers may

differ substantially. Although a JIT optimizer usually contains some commonly used offline compiler optimizations, many other offline optimizations, such as loop transformations, are hard to be adopted in DO systems owing to their high runtime overheads. Meanwhile, novel adaptive optimizations utilizing feedback information are designed specifically for DO systems [6]. Hence, dynamically optimized programs usually exhibit characteristics that differ to their statically optimized counterparts [67]. Second, as an important part of many DO systems, the garbage collector may also change a program's runtime behavior by compacting and rearranging heap objects. Finally, a program's dynamic execution comprises the application, as well as numerous assisting DO services, such as JIT optimization and garbage collection. Being integral parts of the dynamic execution, JIT optimization and garbage collection have distinct characteristics to applications, and thus affect the overall runtime characteristics [29]. In this section, the energy and power impact of two most important DO services, JIT optimization and garbage collection, are examined.

7.1.1 Impact of JIT optimization

Jikes RVM has two compilers. Jikes RVM's baseline compiler is a fast non-optimizing compiler that converts Java bytecodes to machine code. The optimizing compiler of Jikes RVM has three levels of optimizations (JIT0, JIT1, and JIT2), each one consisting of its own group of optimizations as well as the optimizations that belong to lower levels. The lower two levels (JJT0 and JIT2) perform optimizations that are fast and high-payoff. JIT0 performs inlining and register allocation. JIT1 contains optimizations, such as such as common sub-expression elimination, copy and constant propagation, and dead-code elimination. JIT2 contains more expensive ones, such as those based on static single assignment (SSA) form. In this research, we do not use Jikes RVM's sampling based adaptive optimization system. All methods are compiled only

once, and are optimized if the optimizing compiler is used. Lee et al. [52] evaluate the performance impact of various Jikes RVM' optimizations on PowerPC and IA-32 machines.

In this section, JIT1 optimizations are compared with the baseline compiler (BASE). We did not show results for JIT optimization level zero since they are similar to JIT1 results [52]. Unfortunately, JIT2 optimizations cannot be tested in this work since Dynamic Simplescalar fails when high level optimizations generate instructions or call system calls that are not implemented in Dynamic Simplescalar. If DSS does not fail, we expect that using high level optimizations will yield incremental energy reduction over the first level. The main source of energy reduction has been performance improvement. In a compiler with multiple levels of optimizations, lower level optimizations usually achieve the largest performance improvements over no optimization, while high level optimizations yield diminishing improvements. Hence, the energy reduction achieved by high level optimization should also diminish.

The impact of JIT optimization on a program's dynamic execution is two-folded. First, JIT optimization alters a program's runtime behavior. Second, as an integral part of programs' dynamic execution, the JIT optimizer normally possesses distinct characteristics to the applications. The following subsections first studies the overall impact of JIT optimization. Then, the performance and energy impact of each of the above two factors is evaluated. In the experiment, the 200M heap is used to minimize garbage collection activities in SPECjvm 98 benchmarks.

7.1.1.1 Impact on performance, energy, and power

Table 10 presents the instruction counts, cycle counts, energy consumption, and average power of SPECjvm 98 benchmarks with JIT1 optimizations as fractions of the corresponding BASE results. Using the JIT optimizations significantly reduces a

program's instruction count, execution time, and energy consumption. On average, JIT1 optimizations reduce a program's instruction count, cycle count, and energy consumption to 32%, 36%, and 33% of those without JIT optimization.

Table 10. Performance, energy and power of JIT optimized system as fractions of the corresponding results of un-optimized system.

JIT1/BASE	compress	jess	db	javac	mpegaudio	mtrt	jack	avg
Instr. count (%)	17.1	32.2	40.4	40.1	27.1	33.3	32.0	31.7
Cycle count (%)	19.1	39.1	47.1	44.1	33.7	33.3	37.0	36.2
Energy (%)	17.7	35.9	41.5	42.3	28.8	33.0	34.3	33.4
Power (%)	92.6	91.8	88.3	96.0	85.6	99.1	95.4	92.7

JIT1 optimizations reduce both dynamic instruction counts and execution time. Nevertheless, those optimizations are more effective on reducing instruction counts than execution time. JIT optimizations can be broadly classified into two groups: those that reduce instruction counts, and those that improve ILP without reducing instruction counts, such as instruction scheduling. Most JIT1 optimizations, such as common sub-expression elimination, copy and constant propagation, and dead code elimination, belong to the first group. Hence, for all programs studied, the reduction of execution time is not as much as the reduction of instruction counts. The increases of L1D miss rates (Table 11) also contribute to the growing disparities between programs' instructions and cycles.

A program's energy consumption is related to both the total amount of work done by the program (instruction count) and the time taken to finish the work (cycle count). Apparently there is more idling when the optimized code is executed. Consequently, SPECjvm 98 benchmarks' average power dissipation decreases due to the JIT

optimizations. On average, a program's JIT-optimized execution dissipates 93% of the power as its non-JIT-optimized execution.

Among the seven benchmarks, *compress* benefits the most from JIT optimization. The benchmark *compress* spends most of its execution on two loops. The first loop compresses input files, which are then decompressed in the second loop. Hence, *compress* has a few hot methods within the loops that dominate its execution. Optimizing those hot methods can significantly improve the performance and energy consumption while incurring minimal overhead. In contrast, *javac* and *jack* achieve the least performance improvement and energy reduction among all the workloads. The benchmark *javac* is a Java compiler that translates Java source code into binary code, while *jack* is a Java parser generator with lexical analysis. Comparing with other benchmarks, they contain more methods that require more time to optimize.

7.1.1.2 Impact on instruction window and caches

To examine the impact of JIT optimization on program execution, the characteristics for the instruction window and the cache hierarchy are obtained. Those metrics include issue queue and reorder buffer occupancies, and L1D/L1I/L2 cache miss rates, which are closely related with the CUs examined in the next subsection, and help us understand how those CUs' adaptation is interfered by JIT optimization. The results presented in Table 11 are the fractions of the JIT1 results over the corresponding BASE ones.

All fractions, except L2 miss rate, increase as the JIT optimizations are applied. During the JIT optimizations, large data structures are used to store intermediate representations of methods and other information. Since the L1D cache can rarely hold all the data structures, the JIT optimizer's traversing of the data structures incurs many L1D misses. Hence, programs' overall L1D miss rates increase. As the pipeline is slowed

down to wait for the missed data accesses, many dependent instructions stall in the issue queue and the reorder buffer and result in high IQ and ROB occupancies.

Table 11. Runtime characteristics of JIT optimized system as fractions of the corresponding results of un-optimized system.

JIT1/BASE	compress	jess	db	javac	mpegaudio	mtrt	jack	avg
IQ occupancy (%)	123.1	99.4	109.0	100.9	115.0	117.7	117.7	111.8
ROB occupancy (%)	126.2	104.0	111.1	103.3	121.3	113.3	123.2	114.6
L1D miss rate (%)	110.9	109.6	106.3	117.6	113.7	107.1	107.3	110.4
L1I miss rate (%)	103.0	100.2	105.2	101.8	103.3	100.2	98.9	101.8
L2 miss rate (%)	102.2	94.0	97.8	91.9	105.6	95.6	93.9	97.3

The JIT optimizations' impact on the L1I cache is two folded. First, JIT optimizations generate more compact code, resulting in fewer L1I capacity misses. On the other hand, after the optimized code is generated, the instruction cache must be flushed to bring in the newly generated code, incurring more cold misses. Determined by those two conflicting factors, L1I miss rates increasing slightly. For the L2 cache, JIT optimization greatly reduces the sizes of programs' instruction working sets, improving the L2 cache miss rates. In contrast to the L1D cache, the L2 cache is much larger to hold most of the intermediate data structures, and thus is barely affected by the JIT optimizer's accesses to those data structures. Hence, JIT optimization reduces the L2 miss rates.

Note that all characteristics examined in Table 11 are fractions by themselves. The biggest impact of JIT optimization is the instruction and cycle count reduction (Table 10). Comparing the fractional characteristics allows us identifies program behavior changes hidden by instruction and cycle count reduction. For instance, a program has more instructions (Table 10), and thus more L1D misses and accesses in the BASE

configuration than in the JIT1 configuration, although the L1D miss rates increase after the JIT optimization.

7.1.1.3 JIT optimizer versus application

JIT optimization's impact on a program's dynamic execution is two-folded. First, JIT optimization alters a program's runtime behavior. Second, as an integral part of programs' dynamic execution, the JIT optimizer normally possesses distinct characteristics to the applications. Both factors attribute to the overall characteristics of a program's dynamic execution, which is investigated in the previous subsection.

This subsection evaluates the performance and energy impact of each of the above two factors. To do so, the Jikes RVM is instrumented to identify code regions belonging to the JIT optimizer from the rest of the programs' dynamic execution, which is dominated by the execution of the application code. Table 12 gives the portions of overall execution time (% of JIT cycle count), instruction count (% of JIT instr. count), and energy consumption (% of JIT energy) corresponding to the JIT optimizer. The differences between 100% and the JIT percentages are the portions taken by applications. On average, the JIT optimizer accounts for 9.8% of total instructions, 11% of total cycles, and 10.7% of total energy consumed in a program's JIT-optimized execution.

Table 12. Comparison of JIT optimizer and application's energy consumption and power dissipation.

.	**compress**	**jess**	**db**	**javac**	**mpegaudio**	**mtrt**	**jack**	**avg**
% of JIT instr. count	5.1	9.7	6.3	15.3	8.7	10.9	12.8	9.8
% of JIT cycle count	5.7	10.7	7.4	17.5	10.9	11.1	14.0	11.0
% of JIT energy	5.6	10.7	7.3	16.2	10.4	11.0	13.6	10.7
Norm. JIT power %	91.5	91.2	87.7	88.8	82.4	98.9	92.8	90.5
Norm. App. power %	92.6	91.8	88.3	96.0	85.6	99.1	95.4	92.7

The last two columns of Table 12 represent the normalized power results for the JIT compiler and the applications respectively. Those power results are normalized to the ones corresponding to the programs' non-JIT-optimized execution, and represent the power changes of the JIT optimizer and the applications as the JIT optimizations are applied. A normalized power value smaller than 100% means that the average power drops due to JIT optimization. Table 12 implies that both the JIT optimizer and the JIT-optimized applications dissipate less power than the non-JIT-optimized applications, and contribute to the decrease of overall power due to JIT optimization (Table 10).

In this section, we observe that Jikes RVM's JIT1 optimizations reduce average power since both JIT optimizer and optimized application dissipate less power than the un-optimized system. A different dynamic optimization system may not have the same trends. First, JIT optimizer has relative lower power than SPECjvm 98 benchmarks. It is possible that a different application may dissipate much less power than both SPECjvm 98 benchmarks and the JIT optimizer. Second, as explained in Section 7.1.1.1, optimized applications dissipate less power than un-optimized system since the JIT1 optimizations are more effective on reducing instruction counts than cycle counts. Another DO system with different JIT optimizations may act otherwise. In such a system, optimized applications may dissipate more power than un-optimized one.

7.1.2 Impact of garbage collection

The garbage collector used in this research is Appel's generational collector [4] from the GCTk toolkit [12]. The Appel collector uses a size-adaptable nursery, and initially it is the whole heap. Each collection reduces the nursery size by the survivors. A full heap triggers a full heap collection. To test various garbage collection activities, three heap sizes, 25M, 42.5M, and 60M, are used in our simulations, and the results are

compared with those obtained with no GC activities (using a 200M heap). For several benchmarks, 25M is the minimal size in which the Appel collector works [38].

In this subsection, the performance, energy, and power results of programs executed by Jikes RVM with different heap sizes are obtained. Those results are then compared with the ones obtained with a 200M heap that is large enough to have no garbage collection in all benchmarks. The energy impact of the garbage collector is also examined and compared with that of the mutator (i.e., a program's execution other than the garbage collections). The experiment uses JIT1 optimizations.

7.1.2.2 Impact on performance, energy, and power

For each benchmark, Figure 35 shows the normalized instruction count, execution time, and energy consumption by dividing the results for the small heaps by the corresponding ones with the 200M heap.

The energy consumed is again directly proportional to the number of instructions. *javac*'s performance and energy consumption are significantly affected by garbage collection. With a 25M heap, *javac*'s instruction count, cycle count, and energy all increase by 65% over no GC, which is in accordance with the fact that *javac* has more garbage collections than other benchmarks (Table 13). In contrast, with 42.5M and 60M heaps, *mpegaudio*'s performance and energy consumption almost equal to the corresponding ones with no garbage collection, reflecting the fact that *mpegaudio* allocates very little heap space [78]. All the other benchmarks' performance degradation and energy increases are between those of *javac* and *mpegaudio*.

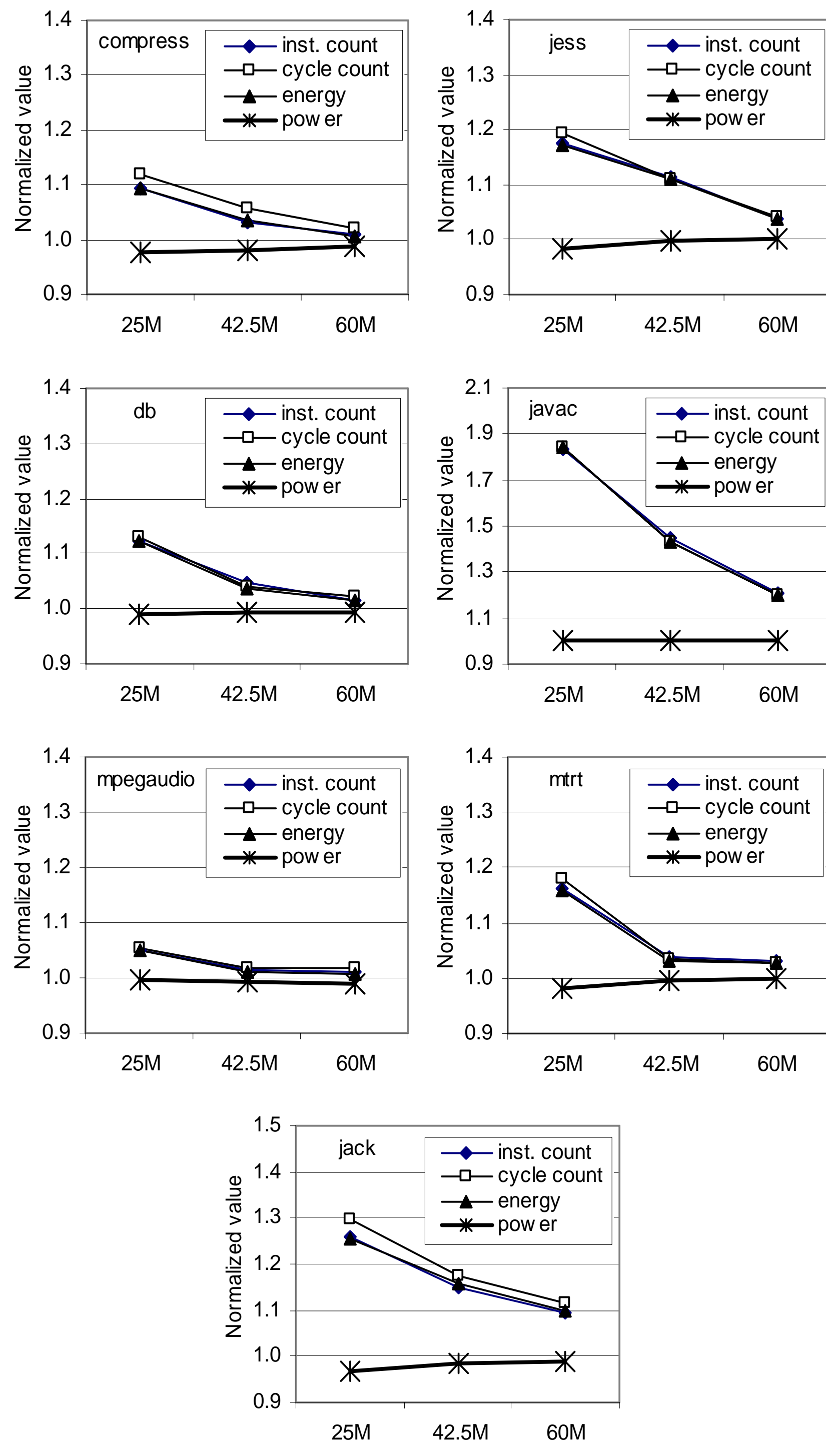

Figure 35. Changes in performance, energy, and power due to garbage collection.

Garbage collection slightly reduces programs' average power dissipation. As Table 13 shows, many benchmarks' L2 misses increase very fast as heap size decreases. Such cache misses can often stall the pipeline, and considerably affect the execution time. On the other hand, since idle units consumes less power, energy is less sensitive to pipeline stalls than execution time, resulting in the lower power dissipation during garbage collection. Among the evaluated workloads, *compress* and *javac*'s average power results drop most, varying between 2% and 4%. According to Table 13, both benchmarks' L2 cache performance and *jack*'s L1 cache performance are impaired by garbage collection.

Note that the energy results in Figure 35 account for only the microprocessor, not the main memory. In our experiment, different heap sizes are used. Heaps reside in the virtual memory, which is usually larger than the physical memory. Several factors determine an application's memory energy consumption. First, accessing a small physical memory consumes less power than accessing a large one. One study on SDRAM energy consumption shows that as the memory size doubles, its energy consumption increases by up to 10% [40]. On the other hand, if the application's data working set is larger than the physical memory, page swaps may occur frequently between the physical memory and storage devices, hurting both performance and energy consumption. Third, the heap size can also affect an application's memory energy consumption. A small heap has to be collected more frequently, and thus has more memory accesses than a larger heap. As shown in Table 13, reducing heap size dramatically increases the amount of L2 misses, i.e., memory accesses. The overall memory energy consumption is determined by those factors as well as the memory access patterns. Unfortunately, a detailed study on memory energy consumption is out of the scope of this research.

7.1.2.1 Impact on caches

Table 13 lists the number of garbage collections (Number of GCs) conducted throughout program execution with heap sizes varying from 25MB to 60MB. A program's number of GCs is highly dependent on the program's memory requirements and lifetimes of the objects allocated. Both *compress* and *mpegaudio* have few garbage collections than other benchmarks, confirming that those two benchmarks allocate very little heap space [78]. In contrast, *javac* performs many more garbage collections than the other benchmarks.

Since the task of garbage collection is to compact and reallocate heap objects, its major performance impact is on the memory subsystem. Table 13 provides the normalized L1D/L1I/L2 misses for each benchmark, obtained by dividing the numbers of misses with the 25M/42.5M/60M heaps by the ones with the 200M heap. The difference between a normalized value and 100% represent the percentage of misses increased due to garbage collection. For comparison, the average instruction counts increase by 21% (25M heap), 12% (42.5M) and 6% (60M) respectively over no GC, and each benchmark's normalized instruction counts are drawn in Figure 35.

Table 13. Number of garbage collections and changes of cache misses due to garbage collection.

	Number of GCs			**L1D misses (%)**			**L1I misses (%)**			**L2 misses (%)**		
	25M	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**
compress	38	18	8	103.4	101.4	100.1	103.3	102.2	100.6	152.0	125.0	102.0
jess	91	87	23	117.4	110.0	102.5	101.0	100.9	99.1	137.0	119.0	104.0
db	73	32	9	103.2	101.8	102.2	104.9	103.3	100.9	103.0	100.0	100.0
javac	606	133	76	152.1	130.4	114.4	100.2	99.6	100.8	186.0	153.0	124.0
mpegaudio	23	9	5	101.8	100.4	100.1	101.8	100.9	103.6	114.0	107.0	104.0
mtrt	153	46	11	108.5	105.3	101.6	104.8	102.7	100.8	120.0	110.0	103.0
jack	171	101	81	129.3	118.2	112.2	103.6	101.3	104.8	147.0	131.0	120.0
avg	122	61	30	116.5	109.6	104.7	102.8	101.5	101.5	137.0	121.0	108.0

As heap size decreases, the numbers of L1D and L2 misses increase faster than L1I misses. Since the garbage collector is small enough to be held in the L1I cache, its execution incurs only a few L1I misses. On the other hand, during the garbage collections a series of memory locations are traversed to find surviving objects, yielding many cold data misses. Furthermore, the garbage collector's accesses to heap objects usually evict data required by the mutator, inflicting many conflict data misses. Although those two factors impair the performance of both L1D and L2 caches, being much larger, the L2 cache's performance is more sensitive to the factors than the L1D cache. Furthermore, since the misses to the L1I and the L1D caches access the L2 cache, more L1I/L1D misses also contribute the increases of L2 misses. Consequently, with more garbage collections, most programs' L2 misses increase much faster than their L1D/L1I misses and instruction counts.

7.1.2.3 Garbage collection versus mutator

Similar to JIT optimization, garbage collection's impact on a program's dynamic execution is two-folded. First, garbage collector alters a program's runtime behavior by compacting and rearranging heap objects. Second, being an integral part of the applications' dynamic execution, the garbage collector normally possesses distinct characteristics to the applications. The previous subsection investigates the overall impact of garbage collection. It is also important to evaluate the performance and energy impact of each of the above two factors.

To do so, the Jikes RVM is instrumented to separate activities of the garbage collector from those of the mutator. Table 14 presents the portions of overall execution time (% of GC cycle count), instruction count (% of GC inst. count), and energy consumption (% of GC energy) spent on the garbage collections. The differences

between 100% and the GC percentages are the portions taken by the mutator. The results confirm that with smaller heaps, the garbage collector is more active, and is responsible for more of the overall work done and energy consumed by a program.

Table 14. Comparison of garbage collector and mutator's energy consumption and power dissipation.

	% of GC cycle count			% of GC inst. count			% of GC energy			GC power %			Mutator power %		
	25M	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**
compress	8.9	3.5	1.1	8.6	3.1	1.0	8.6	3.1	1.0	93.8	86.9	88.2	98.2	98.4	98.9
jess	16.2	11.6	4.3	14.9	10.1	3.7	14.9	10.1	3.7	90.2	87.3	85.8	99.8	101.5	100.6
db	11.0	4.7	1.4	11.0	4.7	1.4	11.0	3.4	1.4	99.1	71.8	97.6	99.1	100.8	99.3
javac	44.4	31.5	20.9	45.4	30.9	17.3	41.8	29.9	18.8	94.0	94.9	89.8	101.8	101.5	101.6
mpegaudio	5.0	1.5	1.2	4.7	1.3	1.1	4.7	1.3	1.1	94.4	87.2	90.3	99.8	99.6	99.1
mtrt	16.7	4.1	3.2	14.0	3.8	2.9	14.0	3.8	2.9	82.9	93.2	91.2	101.4	100.0	100.2
jack	21.8	14.7	9.5	20.6	13.1	8.6	20.6	13.1	8.6	91.3	88.0	89.7	98.4	99.3	99.7
avg	17.7	10.2	5.9	17.0	9.6	5.1	16.5	9.2	5.4	92.2	87.1	90.4	100.2	100.4	100.1

Table 14 also includes the average power results for the garbage collector and the mutator respectively, which are normalized to the ones obtained with the 200M heap. A normalized power value smaller than 100% means that the average power drops due to garbage collection. On average, the garbage collector’s power dissipation is between 87% and 92%, varying by heap sizes and programs, of the mutator’s. This is mainly because the poor data locality of the garbage collector impairs its execution time more than its energy consumption. On the other hand, garbage collection only slightly changes the mutator’s average power, with variations below 2% in all benchmarks and heap sizes. Garbage collection improves a program’s data locality [11], which may speed up the execution and increase the power of the mutator. Table 14 shows that for most programs, except for *jess* and *javac*, the power increase due to improved data locality is minimal. Figure 35 indicates that with garbage collection, a program’s average power dissipation

usually drops. The results in Table 14 denote that the garbage collector, instead of the mutator, is responsible for the drop of overall power dissipation.

7.1.3 Key insights

By reducing instruction and cycle counts, JIT optimization is effective on reducing a program's energy consumption. On the other hand, many of the JIT optimizations studied in this work reduce instruction counts more than execution time. As a result, JIT optimization decreases a program's average power dissipation. A detailed study implies that both the JIT optimizer and the JIT-optimized applications contribute to the drop of the average power.

Garbage collection incurs significant performance and energy overhead. However, poor data locality of the garbage collector affects execution time more than total energy consumption. Hence, garbage collection decreases the average power dissipated by a program. In contrast to JIT optimization, the drop of a program's average power is mainly contributed by the garbage collector, while the mutator's average power is rarely affected by varying garbage collection activities.

7.2 INTERFERENCE OF JIT OPTIMIZATION AND GARBAGE COLLECTION ON HARDWARE ADAPTATION

Both JIT optimizations and garbage collection can alter the runtime behavior of dynamically executed programs. Such changes of program behavior may interfere with the tuning of configurable units and affect their energy reduction. Hence, it is important to understand how JIT optimization and garbage collection interact with configurable hardware units to achieve overall energy reduction.

This section investigates the interference of JIT optimization and garbage collection on the adaptation of the five configurable units listed in Table 2. The hardware adaptation framework presented in Chapter 3 manages those configurable units. To allow

a fair comparison between the CUs, each CU has a performance degradation budget of 1%.

7.2.1 Interference of JIT optimization

7.2.1.1 Energy consumption of hardware units

To evaluate the interference of JIT optimization on the five configurable units, it is important to know the energy impact of JIT optimization on the corresponding fixed size units. For the five fixed size units, Table 15 presents the JIT1 energy results as fractions of the BASE ones. The smaller the result, the less energy a JIT optimized program consumes.

Table 15. Hardware units' energy consumption in JIT optimized system as fractions of those of un-optimized system.

JIT1/BASE	**compress**	**jess**	**db**	**javac**	**mpegaudio**	**mtrt**	**jack**	**avg**
IQ (%)	19.0	34.4	39.5	41.4	28.1	36.2	40.8	34.2
ROB (%)	18.6	35.6	39.3	42.5	27.3	35.1	41.5	34.3
L1D (%)	20.0	39.6	47.3	44.5	33.9	33.9	43.4	37.5
L1I (%)	18.0	36.7	40.0	43.0	30.9	34.0	41.5	34.9
L2 (%)	17.5	32.9	36.8	40.5	27.4	33.6	40.1	32.7

Due to JIT optimization, all five hardware units' energy consumption drops. With L2 miss rates reduced by JIT optimization (Table 11), L2 cache enjoys higher energy reduction than other units. In contrast, JIT optimization reduces less energy on the L1D cache than on others units, owing to the poor L1D performance under JIT optimization (Table 11). JIT optimization's energy impact on the other three units roughly equals to its energy impact on the microprocessor (Table 11). Note that higher IQ and ROB occupancies (Table 11) do not necessarily mean higher IQ and ROB energy consumption.

Despite their high occupancies, both issue queue and reorder buffer are rarely idle, i.e., not accessed, in a cycle. Hence, there are no significant changes in those two units' energy consumption.

7.2.1.2 Impact on overall energy reduction

To measure the impact of JIT optimization on hardware adaptation, we simulate each configurable hardware unit's energy consumption results in both BASE and JIT1 levels. To avoid the interference of one CU to another one's adaptation, each time only one CU is adapted, and all other units keep their largest sizes throughout program execution. Averaged over all the benchmarks, the results in Figure 36 represent the percentages of energy reduced by hardware adaptation in both JIT optimization levels. The differences between those two sets of results represent the impact of JIT optimization on the CUs' adaptation.

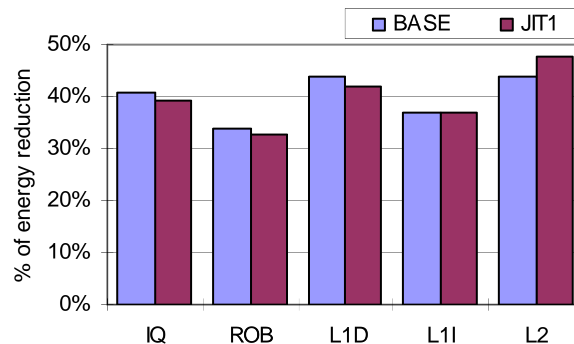

Figure 36. Impact of JIT optimization on configurable unit energy reduction.

According to Figure 36, JIT optimization impairs the adaptation of issue queue, reorder buffer, and L1D cache. As Table 11 shows, L1D miss rates increase as JIT optimizations are applied. In this case, the L1D cache may choose a larger size to prevent

high performance loss and incurs less energy reduction in JIT1 than in BASE. One interesting observation is that although higher IQ and ROB occupancies do not mean higher energy consumption in the fixed size IQ and ROB, they prevent the configurable IQ and ROB to use more aggressive configurations for higher energy saving. Similarly, since JIT optimization improves L2 cache's performance, the cache can use smaller sizes for higher energy reduction. Finally, the L1I cache performance is barely affected by JIT optimization. Consequently, L1I cache's adaptation efficiencies (i.e., percentage of energy reduction due to hardware adaptation) barely change as the JIT optimizations are applied.

7.2.1.3 JIT optimizer versus application

Figure 37 further investigates the CUs' adaptation efficiencies on the JIT optimizer (JIT) and the applications (App). For each CU, the energy results are obtained for the JIT optimizer and the applications respectively, and are then normalized to the corresponding ones using fixed size units to obtain the portions of energy reduced by adapting the CUs on the JIT optimizer and the applications. The results in Figure 37 are the arithmetic means of the seven benchmarks.

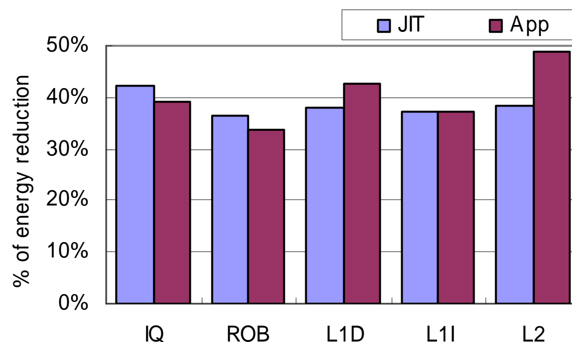

Figure 37. Comparison of JIT optimizer and application on configurable unit energy reduction.

Comparing with the applications, the JIT optimizer achieves higher energy reduction on issue queue and reorder buffer, but lower energy reduction on L1D and L2 caches. During JIT optimization, large intermediate data structures are traversed, resulting in a lot of data misses. Examining various conditions to find optimization opportunities causes a lot of branches. Consequently, the JIT optimizer usually has little ILP and prefers a simpler processor core, and issue queue and reorder buffer achieve more energy reduction on the JIT optimizer than on the applications. On the other hand, since the intermediate data structures are held in both L1D and L2 caches, it is hard to reduce the sizes of the two caches without impairing their performance. Hence, those two caches tend to choose more larger configurations on the JIT optimizer than on the applications. Finally, the L1I cache achieves similar energy reduction on both the JIT optimizer and the applications.

7.2.2 Interference of garbage collection

7.2.2.1 Energy consumption of hardware units

To evaluate the interference of garbage collection on the five configurable units, it is important to know the energy impact of garbage collection on the corresponding fixed size units. For each fixed size unit, we obtain its energy consumption under different heap sizes. Those results are then normalized to those using 200M heap. The results are presented in Table 16. Figure 38 presents each unit's energy averaged across all benchmarks, and the three bars for each hardware unit correspond to the three heap sizes. The higher the bar, the higher energy the specific unit consumes. For comparison, the energy results for the overall system (overall) are also presented in the figure, which are the arithmetic means of the one presented in Figure 35.

Table 16. Fixed size hardware units' energy consumption with small heaps as fractions of those with 200M heap.

	IQ (%)			ROB (%)			L1D (%)			L1I (%)			L2 (%)		
	25M	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**	**25M**	**42.5M**	**60M**
compress	110.9	105.4	103.1	110.9	105.4	103.1	108.1	101.3	99.1	107.9	102.2	99.5	111.6	107.3	103.8
jess	119.7	113.2	106.2	119.7	113.0	105.8	114.9	108.1	101.4	114.8	108.4	102.0	120.2	113.2	106.0
db	113.0	106.5	103.6	112.9	106.2	103.8	111.4	102.1	99.8	111.3	102.6	100.0	112.8	106.1	104.2
javac	166.2	146.7	122.9	166.2	145.7	122.6	163.1	137.7	116.7	163.1	141.7	118.5	166.5	144.9	122.4
mpegaudio	106.1	103.3	103.1	106.0	103.2	103.1	103.5	99.3	99.2	103.5	99.5	99.4	106.6	103.7	103.9
mtrt	117.3	102.4	102.3	117.2	102.6	102.6	114.8	97.7	98.1	114.8	99.4	99.2	117.5	105.5	104.8
jack	127.3	117.3	111.7	127.3	117.5	111.9	123.7	112.4	107.2	123.7	113.6	108.0	128.0	119.8	113.7
avg	122.9	113.5	107.6	122.9	113.4	107.6	119.7	108.4	103.1	119.9	109.6	103.8	123.3	114.4	108.4

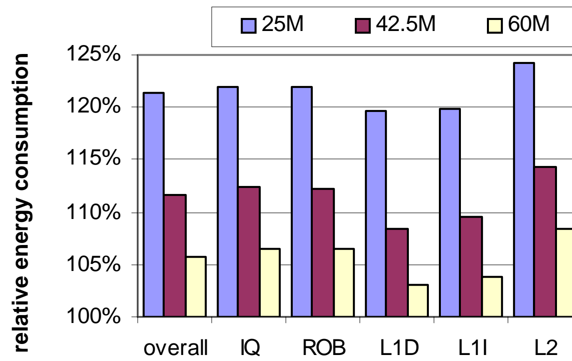

Figure 38. Fixed size hardware units' energy consumption with small heaps as fractions of those with 200M heap. (The results are the same as the ones in the last column of Table 16).

As heap size decreases, both level-one caches' energy increases slower than the other hardware units and the microprocessor, due to the two caches' good performance with garbage collection (Table 13). On the other hand, garbage collection significantly increases L2 cache misses (Table 13). Consequently, the L2 cache's energy consumption increases much faster than other units. The issue queue and the reorder buffer perform similarly to the whole system, implying that garbage collection rarely affect the power dissipation of those two units.

7.2.2.2 Impact on overall energy reduction

Both garbage collection and hardware adaptation affect a configurable unit's energy consumption. To examine the two factors, we first obtain each fixed size unit's energy consumption using the 200M heap, which is used as the baseline. Then hardware adaptation is enabled, and each unit's energy consumption is obtained using different heap sizes. To prevent the interference between CUs, each experiment tests only one configurable unit, and the other units keep constant with their largest configurations. The results are normalized to the baseline ones to show the relative change caused by hardware adaptation and garbage collection, which are then averaged all benchmarks and presented in Figure 39.

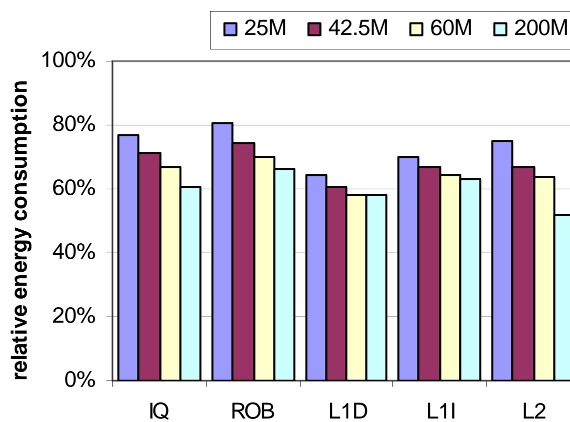

Figure 39. Impact of garbage collection and hardware adaptation on configurable unit energy consumption.

For all hardware units, a program's energy consumption is reduced by hardware, but increased by garbage collection. With all heap sizes, hardware adaptation reduces all CUs' energy consumption by 20% or more over the baseline. Hence, hardware adaptation effectively offsets the energy increase owing to garbage collection. Garbage collection's impact on CU energy consumption varies by CUs. With more frequent garbage

collections, issue queue, reorder buffer, and L2 cache's energy consumption increases faster than L1I and L1D cache, similar to the fixed size units (Figure 38).

Since garbage collection changes program behavior, it may affect hardware adaptation's performance. To study the impact of garbage collection on hardware adaptation, we obtain each configurable hardware unit's energy consumption results with different heap sizes. To avoid the interference of one CU to another one's adaptation, each time only one CU is adapted, and all other units keep their largest sizes throughout program execution. Then the configurable unit energy results are normalized to the corresponding fixed size unit results with the same heap sizes. The results, averaged across all benchmarks, are shown in Figure 40.

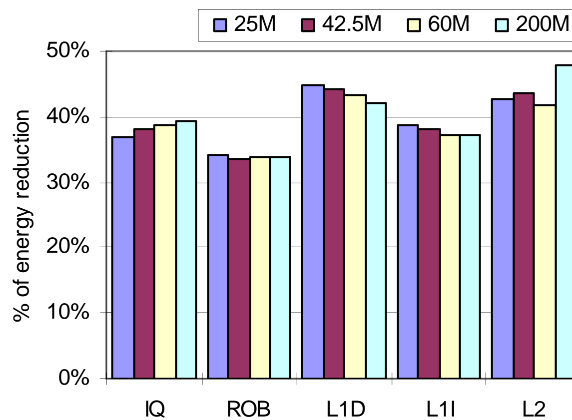

Figure 40. Impact of garbage collection on configurable unit energy reduction.

First, garbage collection's impact on the three caches differs. The adaptation efficiencies of both level-one caches increase as the heap size decreases. As shown in Table 13, garbage collection improves L1D and L1I caches' miss rates, allowing the two caches to use more aggressive configurations for higher energy reduction. On the other

hand, garbage collection hurts L2 cache's adaptation efficiency, owing to the increase of L2 cache misses caused by garbage collection (Table 13). In this case, reducing the L2 cache size will inevitably incur more misses and impair the performance. Hence, with more garbage collection activities, the L2 cache tends to uses more larger configurations and reduces less energy. Moreover, garbage collection adversely affects the adaptation of issue queue, while has minimal impact on the reorder buffer's adaptation.

7.2.2.3 Garbage collection versus mutator

Figure 41 separates the activities of the garbage collector from those of the mutator, and investigates the CUs' adaptation efficiencies on the garbage collector (GC) and the mutator (Mt) respectively.

First, the CUs tend to use smaller configurations on the garbage collector than on the mutator. Comparing with the mutator, the garbage collector possesses distinct runtime characteristics that prefer a simple processor core and small caches for high energy reduction. Being small enough [12], the GC code can normally be held in a small instruction cache. During garbage collection, pointer-chasings to find surviving heap objects result in many data accesses that have poor temporal locality. Hence, reducing L1D and L2 cache sizes rarely affects the garbage collector's performance. Being memory bound, the garbage collector usually has little ILP, and thus smaller issue queue and reorder buffer can be used with minimal performance impact. Hence, all the five CUs achieve higher energy reduction on the garbage collector than on the mutator. Furthermore, as heap size decreases, the variations of the CUs' energy reduction on the garbage collector are minimal, implying that they are rarely changed by the frequency of garbage collections.

The mutator results in Figure 41 represent the garbage collection's impact on the mutator, which vary by configurable units. With smaller heaps and more garbage

collections, IQ, ROB, and L2 cache's energy reduction on the mutator drops. On the other hand, L1D and L1I caches' adaptation efficiencies rarely changes as heap size decreases. The mutator is responsible for the overall variations of CUs' adaptation efficiencies (Figure 39) as heap size changes.

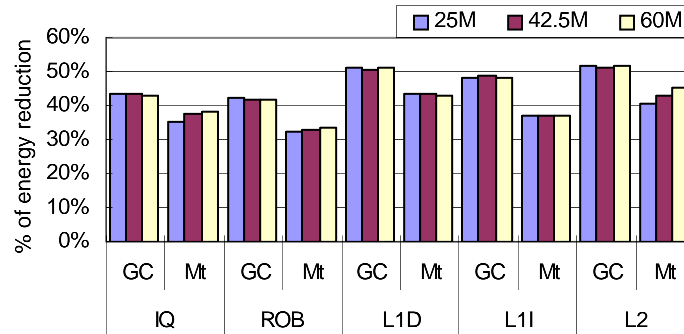

Figure 41. Comparison of garbage collector and mutator on configurable unit energy reduction.

7.3 DISCUSSION

The study in this section reveals that dynamic execution interferes with hardware adaptation. Both JIT optimization and garbage collection alter programs' behavior and runtime requirements. In an adaptive microarchitecture, such runtime requirements changes can considerably interfere with the adaptation of configurable units, and eventually affect the overall energy consumption.

Furthermore, the JIT optimizer and the garbage collector have runtime characteristics that differ considerably from application code. Consequently, the configurable units' adaptation preferences on the two DO services can differ substantially from the units' adaptation decisions on the application code. In a multi-threaded dynamic optimization system like Jikes RVM, the JIT compiler and the garbage collector use

separate threads to the application, which eases the separation of DO services from applications and makes hardware tuning for the dynamic optimization services feasible.

With the growing popularity of dynamic optimization systems in many software levels, it is important for a future system to provide high performance for such dynamic optimization system. For instance, the code of garbage collector is usually compact, and can be fixed in a very small instruction cache [12]. Hence, by providing a very small instruction cache configuration, instruction cache energy can be significantly reduced for the garbage collector.

Coprocessors for JIT compilation and garbage collection are proposed previously for improving the performance of dynamic optimization systems [57][66]. The study in this chapter implies that coprocessors may also benefit a system's energy efficiency. Differing from most applications, both the JIT compiler and the garbage collector prefer a simple and narrow execution engine. Such coprocessors should be much simpler and dissipate far less power than the main processor. The energy consumed by a coprocessor can be further reduced by carefully adjusted its parameters at design time to balance its performance and energy. Furthermore, using coprocessors avoids the hardware adaptation related overheads for the DO services. Hence, executing the dynamic optimization services on coprocessors may consume less energy than executing them on the main processor, even though the latter uses hardware adaptation for improved energy efficiency.

Chapter 8. Conclusions and Future Research

8.1 Conclusions

In adaptive microarchitectures, efficient management of the configurable resources is vital for maximizing energy reduction. To achieve high energy reduction, an adaptive microarchitecture usually has multiple configurable units. The tuning space explodes quickly as more hardware units become configurable. Since hardware units can hardly be adapted individually, the straightforward tuning strategy of testing all combinatorial configurations results in long tuning process and high tuning overhead, and impairs performance significantly.

This dissertation proposes a hardware adaptation framework that utilizes the inherent capabilities of dynamic optimization systems to manage multiple configurable units efficiently. Most dynamic optimizers typically detect hot spots for performance related optimizations. In the framework, hot spot boundaries are used for phase detection and hardware adaptation. Since hot spots are of variable sizes and are often nested, program phase behavior that is hierarchical in nature is automatically captured in this technique. Utilizing this capability, the framework decouples the reconfiguration of different CUs by adjusting the granularity of adaptation based on each CU's reconfiguration cost. This strategy significantly reduces the tuning process, and achieves better balance of benefit/overhead for each configurable unit.

The goal of a hardware adaptation framework is to reduce the overall microprocessor energy consumption. In this work, we implement five configurable units, and our results indicate that those configurable units consume 60% of a microprocessor's energy. Hence, reducing the energy consumed by those CUs can considerably reduce the overall microprocessor energy consumption. Figure 42 presents the percentages of energy

reduced for the microprocessor, achieved by adapting the five configurable units using the DO-based hardware adaptation framework that employs both CU decoupling and the two tuning reduction strategies described in Chapter 6. Using the framework, each CU's energy reduction results and the performance degradation results are presented in Section 6.3.2. As Figure 42 shows, with a 5% performance loss, the framework reduces a microprocessor's energy consumption by 23% on average. With its ability to manage multiple configurable units efficiently, the hardware adaptation framework can further improve the microprocessor energy efficiency if more hardware units are configurable.

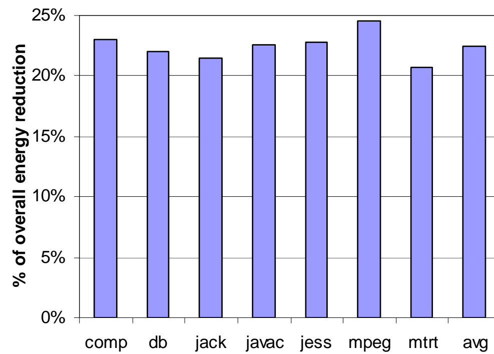

Figure 42. Percentage of a microprocessor's energy reduced by using the hardware adaptation framework to manage the five configurable units.

The following are the major findings and contributions to efficient adaptation of multiple configurable units.

- **Efficient adaptation of CUs with diverse adaptation costs via CU decoupling**

  This research demonstrates how inherent capabilities of a dynamic optimization system can be synergistically employed for efficient management of adaptive computing environments. Utilizing existing DO hot spot detection mechanisms, the proposed technique accurately detects program behavior at varying

granularities, providing the opportunity to significantly reduce the overhead associated with adaptation decisions. By matching each hot spot with a subset of available configuration units, the number of tested configurations is reduced while searching for the most energy-efficient one, thereby reducing the tuning process significantly.

CU decoupling requires that the phases are nested and of variable sizes. As demonstrated by the BBV scheme in this work, other hardware adaptation schemes can be extended to provide hierarchical phase detection. With this capability, they can employ the CU decoupling technique to improve energy efficiency.

- **Tuning space reduction via adaptation interference analysis**

During the tuning process of a multi-CU environment, two major factors affecting CUs' tuning decisions are the program runtime characteristics and the interference between CUs' adaptation. The following finds are gained during the investigation of those two factors.

1) One CU's size reduction may relieve/increase the performance pressure on the other one, and allows it to use smaller/larger sizes. Such positive interference exists between issue queue and reorder buffer.

2) The interference intensity between CUs varies. L1D and L1I caches have minimal mutual interference, and thus can be adapted independently.

3) For the CUs studied in this dissertation, it is inaccurate to predict a CU's best configuration based on the value of a microarchitecture-dependent runtime characteristic, such as IPC.

With the findings, two tuning reduction strategies are designed to further improve the hardware adaptation framework's efficiency on managing multiple

configurable units. The first strategy reduces the latency of tuning L1D and L1I caches by adapting them in parallel. Exploiting the positive property between issue queue and reorder buffer, the second one reduce the tuning latency of those two CUs by avoiding testing combinational configurations that conflict with the property. The tuning reduction algorithms are generalized for prune the tuning space of multiple independent CUs, or CUs with positive interference.

- **Impact of dynamic optimization services on energy consumption and hardware adaptation**

  As dynamic optimization systems have been increasingly popular, it is important to evaluate the impact of dynamic optimization systems on microprocessor energy consumption and hardware adaptation. This paper characterizes the energy and power impact of the two most important services of DO systems, Just-in-time (JIT) optimization and garbage collection. By reducing instruction counts, JIT optimization significantly reduces a program's energy consumption, while garbage collection incurs runtime overhead that consumes more energy. Interestingly, both JIT optimization and garbage collection decrease the average power dissipated by a program. Detailed analysis reveals that both JIT optimizer and JIT optimized code dissipate less power than un-optimized code. Since both JIT optimizer and JIT optimized code stress the level-one data cache, during a JIT optimized system' execution, hardware units in the execution engine are more likely to be idle and thus dissipate less power than an un-optimized system. On the other hand, being memory bound and with low ILP, the garbage collector dissipates less power than the mutator, but rarely affects the mutator's average power. This research also reveals that JIT optimization and garbage collection interfere with hardware adaptation. Both JIT optimization and garbage collection

alter programs' behavior and runtime requirements. In adaptive microarchitectures, such changes of runtime requirements can considerably affect the adaptation decisions of configurable hardware units, and eventually influence the overall energy consumption of the underlying adaptive microarchitecture. This research also studies the adaptation preferences of configurable units on the JIT optimizer and the garbage collector. Owing to their distinct runtime characteristics, such as both DO services' poor data cache performance, the two dynamic optimization services have adaptation preferences differing substantially from the applications.

8.2 DIRECTIONS FOR FUTURE RESEARCH

- **More configurable units for higher overall energy reduction**

  This research implements and investigates reconfiguration of five configurable units: issue queue, reorder buffer, level-one instruction and data caches, and level-two caches. As shown in Section 2.1, many other configurable units, such as branch predictors and the filter cache, are proposed. Adapting more configurable units efficiently can further improve the energy efficiency of an adaptive microarchitecture. Furthermore, those units may possess distinct adaptation requirements to the units studied in this research, and the interference between those CUs and the existing CUs may be more complex. Investigating those configurable units may lead to new tuning-reduction strategies that are specific for those configurable units.

- **Accurate prediction of CUs' best configurations**

  One of this research's findings is that it is inaccurate to predict a CU's configuration from a microarchitecture dependent characteristic. However, it is still possible to predict a CUs' optimization configurations via examining multiple

runtime characteristics. A CU's best configuration for a given phase is the smallest one that satisfies the phase's requirements. Statistical tools, such as Principle Component Analysis, can help identify the runtime characteristics that are closely correlated with a configurable unit's adaptation decisions. Another way is to analyze the program code to estimate its runtime requirements. In a dynamic optimization, the JIT optimizer analyzes the code to find optimization opportunities. The same code analysis mechanism can be augmented to help find the best configuration for the code. For instance, the code analysis can estimate the available instruction-level parallelism of the code, and guide the tuning of configurable units, such as pipeline width or functional units, by ruling out unpromising configurations, or even avoiding the whole tuning process of the CUs if the best configuration can be accurately predicted. With accurate prediction of CUs' best configurations, the lengthy tuning process can be avoided to achieve significant reduction of the CUs' energy consumption.

- **DO-service-aware adaptation**

  This research indicates that the configurable units' adaptation preferences on the JIT optimizer and the garbage collector differ considerably from their adaptation decisions on the application code. The phenomenon can be exploited by both adaptation microarchitectures and hardware adaptation schemes for high energy efficiency. A DO-service-aware adaptive microarchitecture can improve its energy efficiency by providing hardware configurations accurately matching the runtime requirements of the DO services. With a better understanding of the DO services' runtime requirements, a hardware adaptation scheme can also significantly reduce the tuning process for those DO services by avoiding testing

unpromising configurations, which further improves the energy efficiency of the underlying adaptive microarchitecture.

- **Integrated management of adaptive microarchitecture and dynamic voltage and frequency scaling**

  Currently, the DO-based hardware adaptation framework manages only configurable units. It can further improve a system's energy efficiency by incorporating other energy reduction techniques, such as dynamic voltage and frequency scaling (DVFS). Integrated management of adaptive microarchitecture and DVFS allows the DO-based framework to choose a better technique or a combination of both techniques for a given phase, and thus achieves higher energy efficiency.

Designing and developing techniques applicable to dynamic optimization systems to achieve high energy efficiency is extremely important. A dynamic optimization based hardware adaptation framework incorporating some of the techniques mentioned in this section will further improve the overall energy efficiency, and enhance its ease of use and commercial viability.

Bibliography

[1] D. Albonesi, Dynamic IPC/Clock Rate Optimization, Proceedings of the 25th International Symposium on Computer Architecture, pp. 282-292, 1998.

[2] D. Albonesi, Selective Cache Ways: On-Demond Cache Resource Allocation, Journal of Instruction-Level Parallelism, vol.2, 2000 (http://www.jilp.org/vol2/index.html).

[3] J. Abella and A. Gonzlez, Low-complexity distributed Issue Queue, Proceedings of the 10th International Symposium on High Performance Computer Architecture, pp. 73-84, 2004.

[4] A. W. Appel, Simple Generational Garbage Collection and Fast Allocation, Software: Practice and Experience, 19(2), pp. 171-183, 1989.

[5] B. Alpern, D. Attanasio, J. Barton, A. Cocchi, D. Lieber, S. Smith, and T. Ngo, Implementing Jalapeno in Java, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 314-324, 1999.

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney, A Survey of Adaptive Optimization in Virtual Machine, Proceedings of the IEEE, 93(2), pp. 449-466, Feb. 2005.

[7] I. Bahar and S. Manne, Power and Energy Reduction Via Pipeline Balancing, Proceedings of the 28th International Symposium on Computer Architecture, pp. 218-229, 2001.

[8] V. Bala, E. Duesterwald, and S. Banerjia, Dynamo: A Transparent Dynamic Optimization System, Proceedings of the ACM SIGPLAN 2000 Conference of Programming Language Design and Implementation, pp. 1-12, 2000.

[9] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, Memory hierarchy reconfiguration for energy and performance in general purpose architectures, Proceedings of the 33rd International Symposium on Microarchitecture, pp. 245-257, 2000.

[10] L. Baraz, T. Devor, O. Etzion, S. Gondenberg, A. Skaletsky, Y. Wang, and Y. Zemach, IA-32 Execution Layer: a Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems, Proceedings of the 36th International Symposium on Microarchitecture, pp. 191-201, 2003.

[11] S. Blackburn, P. Cheng, and K. McKinley, Myths and Realities: The Performance Impact of Garbage Collection. Proceedings of ACM SIGMETRICS Performance Evaluation Review, pp. 25-36, 2004.

[12] S. Blackburn, R. Jones, K. McKinley, and J. Moss. Beltway: Getting Around Garbage Collection Gridlock, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 153 - 164, 2002.

[13] D. Brooks, V. Tiwari and M. Martonosi, Wattch: A Framework for Architectural-level Power Analysis and Optimizations, Proceedings of the 27th International Symposium on Computer Architecture, pp. 83-94, 2000.

[14] D. Burger and T. M. Austin, Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report, Department of Computer Science, University of Wisconsin, Madison, 1997.

[15] R. Canal and A. Gonzlez, A Low-Complexity Issue Queue. Proceedings of the 14th International Conference on Supercomputing, pp. 327-335, 2000.

[16] L. Chakrapani, P. Korkmaz, V. Mooney III, V. Palem, K. Puttaswamy, and W. Wong. The Emerging Power Crisis in Embedded Processors: What Can a (Poor) Compiler Do? Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 176-180, 2001.

[17] S. Chheda, O. Unsal, I. Koren, C. Krishna, C. Moritz, Combining compiler and runtime IPC predictions to reduce energy in next generation architectures, Proceedings of the 1st conference on Computing frontiers, pp. 240-254, 2004.

[18] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, J. Mattson, The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges, Proceedings of the 1st International Symposium on Code Generation and Optimization, pp. 15-24, 2003.

[19] A. Dhodapkar and J. Smith, Managing Multi-Configuration Hardware via Dynamic Working Set Analysis, Proceedings of the 29th International Symposium on Computer Architecture, pp. 233-244, 2002.

[20] A. Dhodapkar and J. Smith, Comparing Program Phase Detection Techniques, Proceedings of the 36th International Symposium on Microarchitecture, pp. 217-227, 2003.

[21] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, 141-152, 2002.

[22] S. Dropsho, G. Semeraro, D. Albonesi, G. Magklis, and M. Scott. Dynamically Trading Frequency for Complexity in a GALS Microprocessor. In Proceedings of the 37th annual International Symposium on Microarchitecture, 157-168, 2004.

[23] E. Duesterwald, C. Cascaval, and S. Dwarkadas, Characterizing and Predicting Program Behavior and its Variability, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 220-231, 2003.

[24] K. Ebcioglu and E. Altman, DAISY: Dynamic Compilation for 100% Architectural Compatibility, Proceedings of the 24th International Symposium on Computer Architecture, pp. 26-37, 1997.

[25] D. Ernst, A. Hamel, and T. Austin, Cyclone: a Broadcast-free Dynamic Instruction Scheduler with Selective Replay, Proceedings of the 30th International Symposium on Computer Architecture, pp. 253-264, 2003.

[26] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, Drowsy Caches: Simple Techniques for Reducing Leakage Power, Proceedings of the International Symposium on Computer Architecture, pp. 147-157, 2002.

[27] D. Folegnani and A. Gonzalez, Energy-Effective Issue Logic, Proceedings of the 28th International Symposium on Computer Architecture, pp. 230-239, 2001.

[28] M. Franklin and M. Smotherman, A Fill-Unit Approach to Multiple Instruction Issue, Proceedings of the 27th International Symposium on Microarchitecture, pp. 162-171, 1994.

[29] A. Georges, D. Buytaert, L. Eeckhout, and K. Bosschere. Method-Level Phase Behavior in Java Workloads. In Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 270 – 287, 2004.

[30] M. Gowan, L. Biro, D. Jackson, Power Considerations in the Design of the Alpha 21264 Microprocessor, Proceedings of the 35th Annual Conference on Design Automation, pp. 726-731, 1998.

[31] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, SH3: High Code Density, Low Power, IEEE Micro, 15(6), pp. 11-19, 1995.

[32] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini, Application Transformations for Energy and Performance-Aware Device Management, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp.121-131, 2002.

[33] J. Hennessy, and D. Patterson, Computer Architecture: A Quantitative Approach, 3rd edition, Morgan Kaufmann Publishers, 2002.

[34] C.-H. Hsu, and U. Kremer, The Design, Implementation, and Evaluation of a Compiler Strategy for CPU Energy Reduction. Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 38-48, 2003.

[35] S. Hu, M. Valluri, and L. John, Effective Adaptive Computing Environment Management Using a Dynamic Optimization System, Proceedings of 2005 International Symposium on Code Generation and Optimization, pp. 63-73, 2005.

[36] M. Huang, J. Renau, and J. Torrellas, Positional Adaptation of Processors: Application to Energy Reduction, Proceedings of the 30th International Symposium on Computer Architecture, pp. 157-168, 2003.

[37] M. Huang, J. Renau, S. Yoo, and J. Torrellas, A Framework for Dynamic Energy Efficiency and Temperature Management, Proceedings of the 33rd International Symposium on Microarchitecture, pp. 202-213, 2000.

[38] X. Huang, J. Moss, K. McKinley, S. Blackburn, and D. Burger, Dynamic SimpleScalar: Simulating Java Virtual Machines. The First Workshop on Managed Run Time Environment Workloads, held in conjunction with CGO, 2003.

[39] Intel. Pentium III Processor Mobile Module: Mobile Module Connector 2 (MMC-2) Featuring Intel SpeedStep Technology, 2000.

[40] A. Joshi, S. Kumar, S. Sambamurthy, and L. John, Power Modeling of SDRAMs, Technical Report TR-040126-2, Dept. of Electrical and Computer Engineering, University of Texas at Austin, 2004.

[41] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, Influence of Compiler Optimizations on System Power, Proceedings of the 37th Conference on Design Automation, pp, 304-307, 2000.

[42] A. Iyer, and D. Marculescu, Microarchitecture-level Power Management. IEEE Transactions on VLSI Systems, 10(3), pp. 230-239, 2002.

[43] K. Itoh, Low Power Design Methodologies. Kluwer Academic Publisher, 1996.

[44] T. Jones, M. O'Boyle, J. Abella, and A. Gonzalez, Software Assisted Issue Queue Power Reduction, Proceedings of the 7th International Symposium on High Performance Computer Architecture, pp. 144-153, 2005.

[45] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power, Proceedings of the 28th Annual International Symposium on Computer Architecture, pp. 240-253, 2001.

[46] G. Kucuk, O. Ergin, D. Ponomarev, and K. Ghose, Distributed Reorder Buffer Schemes for Low Power, Proceedings of the 21st International Conference on Computer Design, pp. 364-370, 2003.

[47] G. Kucuk, D. Ponomarev, K. Ghose. Low Complexity Reorder Buffer Architecture. Proceedings of the International Conference on Supercomputing, pp. 24-36, 2002.

[48] J. Kin, M. Gupta, and W. Mangione-Smith, The Filter Cache: An Energy Efficient Memory Structure, Proceedings of the 30th International Symposium on Microarchitecture, pp. 184-193, 1997.

[49] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, Motivation for Variable Length Intervals and Hierarchical Phase Behavior, Proceedings of the International Symposium on Performance Analysis of Systems and Software, pp. 135-146, 2005.

[50] J. Lau, S. Schoenmackers, B. Calder, Transition Phase Classification and Prediction, Proceedings of the 11th International Symposium on High Performance Computer architecture, pp. 278-289, 2005.

[51] T. Li, and L. John. Run-time Modeling and Estimation of Operating System Power Consumption, In Proceedings of the International Conference on Measurement and Modeling of Computer Systems, pp 160-171, 2003.

[52] H. Lee, D. Dincklage, A. Diwan, and J. Moss. Understanding the behavior of compiler optimizations, University of Colorado at Boulder, Department of Computer Science Technical Report CU-CS-972-04, 2004.

[53] S. Manne, A. Klauser, and D. Grunwald, Pipeline Gating: Speculation Control for Energy Reduction, Proceedings of the 25th International Symposium on Computer Architecture, pp. 1-10, 1998.

[54] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In IEEE Micro 23(6). 62-68, 2003.

[55] E. Meijer and J. Gough, Technical Overview of the Common Language Runtime. (http://research.microsoft.com/~emeijer/Papers/CLR.pdf).

[56] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhall, and W. Hwu, An Architectural Framework for Runtime Optimization, IEEE Transactions on Computers, 50(6), pp. 567-589, 2001.

[57] M. Meyer. An On-Chip Garbage Collection Coprocessor for Embedded Real-Time Systems. In Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 517-524, 2005.

[58] P. Michaud and A. Seznec, Data-flow Prescheduling for Large Instruction Windows in Out-of-Order Processors, Proceedings of the 7th International Symposium on High Performance Computer Architecture, pp. 27-36, 2001.

[59] R. Min, W. Jone, Y. Hu. Phased Tag Cache: An Efficient Low Power Cache System, Proceedings of the International Symposium on Circuits and Systems, pp. 805-808, 2004.

[60] S. Palacharla, n. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In Proceedings of the 24th International Symposium on Computer Architecture, pp. 206-218, 1997.

[61] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power Issues Related to Branch Prediction. In Proceedings of the 8th International Symposium on High Performance Computer Architecture, pp, 233 - 244, 2002.

[62] F. Pollack, New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. Keynote of the 32nd annual ACM/IEEE international symposium on Microarchitecture, 2000.

[63] D. Ponomarev, G. Kucuk, K. Ghose, Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources, In Proceedings of the 34th International Symposium on Microarchitecture, pp. 90-101, 2001.

[64] M. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, Reducing Set-Associative Cache Energy via Way Prediction and Selective Direct-Mapping. In Proceedings of the 34th International Symposium on Microarchitecture, pp. 54-65, 2001.

[65] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chains. In Proceedings of the 29th International Symposium on Computer Architecture, pp. 318-329, 2002.

[66] R. Radhakrishnan, R. Bhargava, and L John. Improving Java Performance Using Hardware Translation. In Proceedings of the 15th ACM International Conference on Supercomputing, pp. 427-439, 2001.

[67] R. Radhakrishnan, N. Vijaykrishnan, L. John, A. Sivasubramaniam. Architectural Issues in Java Runtime Systems. In Proceedings of the 6th International Symposium on High-Performance Computer Architecture, pp. 387 – 398, 2000.

[68] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In Proceedings of the 27th International Symposium on Computer Architecture, pp. 214-224, 2000.

[69] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power Awareness through Selective Dynamically Optimized Traces. In Proceedings of the 31st International Symposium on Computer Architecture, pp. 162-172, 2004.

[70] J. Sharkey, D. Ponomarev, K. Ghose, and O. Ergin. Instruction Packing: Reducing Power and Delay of the Dynamic Scheduling Logic. In Proceedings of the International Symposium on Low Power Electronics and Design, pp. 30-35, 2005.

[71] J. S. Sen, E. S. Tune, and D. M. Tullsen. Reducing Power with Dynamic Critical Path Information. In Proceedings of the 34th International Symposium on Microarchitecture, pp. 114-123, 2001.

[72] X. Shen, Y. Zhong, C. Ding. Locality Phase Prediction. In Proceedings of the 11th International Conference on Architectural Support for Programming, Languages, and Operating Systems, pp. 165-176, 2004.

[73] R. Sasanka, C. Hughes, and S. Adve. Joint Local and Global Hardware Adaptations for Energy. In Proceedings of the 2002 International Conference on Architectural Support for Programming Language and Operating Systems. 144-155, 2002..

[74] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 45-57, 2002.

[75] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, Discovering and Exploiting Program Phases. In IEEE MICRO, 23(6), pp. 84-93, 2003.

[76] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In Proceedings of the 30th International Symposium on Computer Architecture, pp. 336-349, 2003.

[77] T. Simunic, L. Benini, and G. D. Micheli. Energy Efficient Design of Battery-Powered Embedded Systems, IEEE Transactions on Very Large Scale Integration Systems, 9(1), pp. 18-28, Feb. 2001.

[78] Y. Shuf, M. Serrano, M. Gupta, and J. Singh. Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations. In Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 194-205, 2001.

[79] E. Talpes and D. Marculescu. Power Reduction through Work Reuse. In Proceedings of the International Symposium on Low Power Electronics and Design, pp. 340-345, 2001.

[80] M. G. Valluri and L. John. Is Compiling for Performance = Compiling for Power? The 5th Annual Workshop on Interaction between Compilers and Computer Architectures, 2001.

[81] M. G. Valluri, L. John, and K. S. McKinley. Low-Power, Low-Complexity Instruction Issue Using Compiler Assistance. In Proceedings of the International Conference on Supercomputing, pp. 209-218, 2005.

[82] A. V. Veidenbaum. W. Tang, R. Gupta, A. Nocolau, X. Ji. Adapting Cache Line Size to Application Behavior. In Proceedings of the International Conference on Supercomputing, pp. 145-154, 1999.

[83] Q. Wu, V. Reddi, Y. Wu, D. Connors, D. Brooks, M. Martonosi, and D. Clark. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. To be appeared at the 38th IEEE/ACM International Symposium on Microarchitecture, 2005.

[84] Y. Wu, M. Breternitz, J. Quek, O. Etzion, J. Fang, The Accuracy of Initial Prediction in Two-Phase Dynamic Binary Translators, In Proceedings of the 2nd International Symposium on Code Generation and Optimization, pp. 227-238, 2004.

[85] H. Yang, G. Gao, and C. Leung. On Achieving Balanced Power Consumption in Software Pipelined Loops, Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 210-217, 2002.

[86] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A Way-Halting Cache for Low-Energy High-Performance Systems. In ACM Transactions on Architecture and Code Optimization, 2(1) pp. 34-54, 2005.

[87] IEEE Computer, Special issue on Adaptive Computing, Vol.37, No.7, July 2004.

[88] SPECjvm98 Benchmarks, http://www.spec.org/osg/jvm98

[89] SPECjbb 2000 Benchmark, http://www.spec.org/jbb2000/

[90] Java technology, http://java.sun.com

[91] Microsoft .NET technology, http://www.microsoft.com/net/

Vita

Shiwen Hu was born in Xi'an, China, on June 18, 1972, as the son of Pingxin Hu and Xiuzhen Yu. After completing his high school education at The High School of The Northwestern Fifth Cotton Mill, Xi'an, China, he entered the Department of Automation in Tsinghua University, Beijing, China, in September 1991. He received the degree of Bachelor of Engineering from Tsinghua University in July 1996. He joined the joint-graduate program for Applied Mathematics and Automation at Tsinghua University, Beijing, China in September 1996 and obtained the degree of Master of Science in July 1998. He entered the Ph.D. program in Computer Engineering at The University of Texas at Austin in September 2000. During the summer and fall of 2004, he interned at Freescale, Inc, working on a dynamic optimizer that translates ARM code to StarCore VLIW code. He is a student member of IEEE.

Permanent address: The Third Hezuo Building, Apt. 8
The Fifth Northwestern Cotton Mill
Xi'an, Shaanxi Province
China, 710038

This dissertation was typed by the author.