Effective Management of Multiple Configurable Units Using Dynamic Optimization

SHIWEN HU

Freescale Semiconductor

MADHAVI VALLURI

IBM, Austin

and

LIZY KURIAN JOHN

The University of Texas at Austin

As one of the promising efforts to minimize the surging microprocessor power consumption, adaptive computing environments (ACEs), where microarchitectural resources can be dynamically tuned to match a program's run-time requirement and characteristics, are becoming increasingly common. In an ACE, efficient management of the configurable units (CUs) is vital for maximizing the benefit of resource adaptation. ACEs usually have multiple configurable hardware units, necessitating exploration of a large number of combinatorial configurations in order to identify the most energy-efficient configuration. In this paper, we propose an ACE management framework for efficient management of multiple CUs, utilizing dynamic optimization systems' inherent capabilities of detecting and optimizing program hotspots, i.e., dominate code regions. We develop a scheme where hotspot boundaries are used for phase detection and adaptation. The framework achieves good energy reduction on managing multiple CUs with minimal hardware requirements and low implementation cost by leveraging the existing infrastructure of a dynamic optimization system. The proposed framework is evaluated by dynamically adapting five CUs with distinct reconfiguration latencies and overheads. Those CUs are issue queue, reorder buffer, level-one data and instruction caches, and level-two cache. Previous research indicates that those five components dominate the energy consumption of a microprocessor. Despite the growing complexity and overhead of adapting five CUs, our technique reduces the energy consumption of those CUs by as much as 45%, while one of the best techniques provided by prior literature achieves less than 15% energy reduction for all CUs.

A preliminary version of the paper entitled "Effective adaptive computing environment management via dynamic optimization," Shiwen Hu, Madhavi Valluri, and Lizy John, It appeared in *The 3rd Annual International Symposium on Code Generation and Optimization (CGO), March 2005* [Hu et al. 2005]. This research was supported in part by NSF grant 0429806, and by IBM, Intel, and AMD Corporations.

Authors' Addresses: Shiwen Hu, 7700 West Parmer Ln, Austin, Texas 78729; email: shiwen.hu@freescale.com; Madhavi Valluri, IBM, Austin, Texas 78712; email: mvalluri@us.ibm.com; Lizy Kurian John, 1 University Station C0803, Austin, TX 78712-0240; email: ljohn@ece.utexas.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2006 ACM 1544-3566/06/1200-0477 \$5.00

Categories and Subject Descriptors: D.3.4 [Software]: Processors—*Optimization*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Adaptive computing environment (ACE), power dissipation, dynamic optimization, hotspots

1. INTRODUCTION

In the past several years, 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 quickly growing. 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 few years.

The increasing power consumption in microprocessors raises concerns in both hardware and software communities. Among the efforts to reduce power consumption, one promising method is to dynamically adapt microarchitectural resources to match changing program requirements [Albonesi 1998, 2000; Bahar and Manne 2001; Albonesi 2000; Balasubramonian et al. 2000; Folegnani and Gonzalez 2001; Kin et al. 1997; Ponomarev et al. 2001; IEEE Computer]. With fixed-size microarchitectural structures, such as wide instruction window and large L1 caches, conventional microprocessors are designed to maximize performance, for a wide range of applications. However, a program rarely fully utilizes every microarchitectural resource to achieve high performance and reducing the sizes of those underutilized microarchitectural resources for energy reduction should have minimal performance impact. To achieve energy reduction, the hardware units must have multiple configurations that can be changed at run-time. Reconfiguration of many such configurable units (CUs), such as issue queue, reorder buffer, caches, and branch predictors, have been proposed [Albonesi 2000; Bahar and Manne 2001; Balasubramonian et al. 2000; Dhodapkar and Smith 2002; Folegnani and Gonzalez 2001; Kin et al. 1997; Ponomarev et al. 2001]. In this work, the term adaptive computing environment (ACE) is used to indicate a microprocessor design with one or more such configurable hardware units.

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. Using the definition of Sherwood et al. [2003], a phase is a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency. An application's execution usually passes through phases with varying run-time characteristics and hardware requirements. Phase boundaries are thus suitable points for resource reconfiguration. Previously proposed phase-detection schemes detect distinct phases by examining changes of various run-time characteristic.

Those phases associate with either successive program sampling intervals [Balasubramonian et al. 2000; Dhodapkar and Smith 2002; Huang et al. 2000; Iyer and Marculescu 2002; Lau et al. 2005; Ponomarev et al. 2001; Sherwood et al. 2003] or code positions [Hu et al. 2005; Huang et al. 2003]. The second issue of hardware adaptation, i.e., which configuration to adapt to, is addressed by the tuning strategy. In the most straightforward tuning strategy, all hardware configurations are tested on a phase change; the most energy-efficient one that satisfies the performance constraint is selected for the phase.

Meanwhile, dynamic optimization (DO) systems have grown in popularity. By dynamic optimization, we mean 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 [Dehnert et al. 2003], IBM DAISY [Ebcioglu and Altman 1997], HP Dynamo Microsoft. net, Intel IA32EL [Baraz et al. 2003], Java virtual machines [Appern et al. 1999; Sherwood et al. 2003], and Microsoft. NET's CLR [Bala et al. 2000]. To amortize the overhead of run-time translation and further improve performance, most DO systems apply high-cost high-quality optimizations only on frequently executed code sequences (*hotspots*), such as methods, loops, and basic block sequences.

This paper has two major contributions. First, we demonstrate that method-based hotspots detected by a dynamic optimization system, Jikes RVM [Appern et al. 1999], closely represent program phase behavior. We measure the per-phase and interphase IPC coefficient-of-variations (CoVs) of hotspots. The per-phase IPC CoVs are IPC variations among different invocations of the same hotspot, characterizing the homogeneity among different invocations of a hotspot. The interphase IPC CoVs are the variations of average IPCs of different hotspots and larger interphase IPC CoVs signify large characteristic differences among the hotspots. The results of the CoVs demonstrate that the differences among hotspots are more prominent than those among invocations of the same hotspot. Therefore, by the definition of phases [Sherwood et al. 2003], hotspots are good representation of phases, and can thus be used for ACE management.

The second contribution of the paper is that it demonstrates how inherent capabilities of a DO system can be synergistically employed for efficient management of ACE. We propose a scheme for efficient management of multiple CUs based on a generic DO system. Exploiting the existing hotspot detection mechanism of the DO system, the proposed ACE framework adapts microarchitectural resources at hotspot boundaries. Program hotspots are usually of variable sizes and invoked in a nested fashion. Smaller hotspots represent fine-grain phases nested within coarse-grain phases that appear as large hotspots. Thus, this framework automatically captures the hierarchical phase behavior by identifying nested hotspots. Utilizing this capability, the framework decouples the reconfiguration of different CUs in an ACE by adjusting the granularity of adaptation based on each CU's reconfiguration cost, and balances benefit/overhead for each configurable hardware resource. Hence, the framework achieves good energy reduction on managing multiple CUs with minimal hardware requirements and low implementation cost by leveraging the existing infrastructure of a DO system.

To demonstrate the usefulness and effectiveness of our framework, we implement and evaluate the proposed ACE management framework using Jikes Research Virtual Machine [Appern et al. 1999] and Dynamic SimpleScalar [Huang et al. 2003] simulator. Performance is evaluated on the SPECjvm98 benchmark suite. We implement five CUs: 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 [Folegnani and Gonzalez 2001]. Hence, improving those CUs' adaptation should considerably reduce the overall microprocessor energy consumption. Despite the growing complexity and overhead of adapting five CUs, our technique reduces the energy consumption of those CUs by as much as 45%, while one of the best techniques provided by prior literature achieves less than 15% energy reduction for all CUs.

The remainder of the paper is organized as follows. Section 2 presents the background on hardware resource adaptation. Section 3 introduces our proposed ACE management framework. The experimental methodology is discussed in Section 4. The evaluation results are presented in Sections 5 and 6 investigates CUs' adaptation efficiencies achieved by the two resource adaptation approaches. Finally, Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

The ultimate goal of a resource adaptation scheme is to achieve the highest reduction in microprocessor energy consumption within a predefined budget of overall performance loss. In a multi-CU environment, the tuning process is the adjustment of each CU's configuration to maximize overall energy saving within the budget of performance loss. Most resource adaptation schemes have two components: a phase-detection mechanism that identifies when to adapt hardware resources and a tuning algorithm to identify which units to configure and how to configure them.

In this section, we first introduce the terms reconfiguration overheads and intervals of CUs that are used throughout the paper. We then describe previously proposed resource adaptation schemes in terms of the above components and identify their key limitations on managing multi-CU ACEs efficiently.

2.1 Reconfiguration Overhead and Interval

Typical CUs include issue queue [Folegnani and Gonzalez 2001; Ponomarev et al. 2001], reorder buffer [Ponomarev et al. 2001], instruction and data caches [Albonesi 2000; Balasubramonian et al. 2000; Dhodapkar and Smith 2002], pipelines [Bahar and Manne 2001], filter cache [Kin et al. 1997], and function units [Bahar and Manne 2001]. Each CU may have multiple fixed settings (e.g., four different cache sizes).

Changing a hardware unit's setting at run-time 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 [Dhodapkar and Smith 2002]. Hence, a program's performance may be impaired by too frequent reconfigurations where reconfiguration overhead

overcomes the benefit gained by the reconfigurations. To amortize the CU reconfiguration overhead, a configuration should be utilized for a certain minimum time interval, called the CU's *reconfiguration interval*. Depending on its reconfiguration overhead, a CU's reconfiguration interval can vary from thousands (e.g., issue queue [Folegnani and Gonzalez 2001]) to millions (e.g., caches [Sherwood et al. 2003]) of instructions/cycles.

2.2 Program Phase Detection

Accurate and timely identification of program phases is essential for improving the effectiveness of adaptation. For instance, in systems that contain CUs with a large reconfiguration overhead, recurring phases may reuse configuration information to avoid repeated tunings, and improve performance.

The phase-detection schemes can be broadly divided into two categories: temporal and positional approaches [Huang et al. 2003]. In temporal approaches, the dynamic execution stream is divided into sampling intervals with fixed or varying sizes. At the end of each sampling interval, characteristics, such as IPC [Folegnani and Gonzalez 2001; Ponomarev et al. 2001], conditional branch counter [Balasubramonian et al. 2000], occupancy [Ponomarev et al. 2001], basic block vectors [Lau et al. 2005; Ponomarev et al. 2001], instruction working sets [Dhodapkar and Smith 2002], and hardware-detected hotspots [Merten et al. 2001], are gathered and compared with preceding ones. A phase change is detected when two consequent intervals behave differently. The phase-identification latency, defined as the number of sampling intervals required to recognize a new/recurring phase, is usually one sampling interval. Among those techniques, the basic block vector (BBV) method [Lau et al. 2005; Ponomarev et al. 2001] is shown to be one of the best [Dhodapkar and Smith 2003]. It gathers dynamic basic block distributions through an array of counters and then computes the Manhattan distances of BBVs to detect phase changes.

Differing from temporal approaches that detect phases at successive sampling intervals, the positional approach [Huang et al. 2003] 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 run-time, the positional approach simply adapts at boundaries of large procedures [Huang et al. 2003]. 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 because of their inability to adapt to changes within the procedures [Dhodapkar and Smith 2003]. Recently, Shen et al. [2004] propose to construct phases on their memory localities. The scheme predicts locality phases based on profiling information obtained via tracing a training input. The phase information is inserted into program code by a static compiler.

2.3 Resource Adaptation

The tuning algorithms used in several previous resource adaptation schemes [Balasubramonian et al. 2000; Huang et al. 2000, 2003; Iyer and Marculescu 2002] are similar in nature. In general, after a phase is found, the tuning

algorithm 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 best-performing configuration 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.

In resource adaptation, short tuning processes are always preferred over long ones. First, tuning latency represents a period of program execution where performance is impaired because of the application of mostly suboptimal configurations. Second, because of the CU reconfiguration overhead, a longer tuning process incurs higher overhead. Hence, with multiple CUs, the straightforward tuning strategy of testing all combinatorial configurations becomes inefficient.

Several prior efforts aim at efficient adaptation of multiple CUs. Dropsho et al. [2002] try to minimize the overhead of managing multiple CUs at the same time by using local information, such as a cache's LRU states and a buffer's occupancy statistics, to guide each CUs 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 to capture the CUs' performance in order to achieve high throughput by trading off clock frequencies with hardware structure sizes [Dropsho et al. 2004]. For multimedia applications, Sasanka et al. [2002] propose multiple local hardware adaptation schemes to exploit intraframe execution variability. Those local schemes are integrated with the global scheme, exploiting per-frame execution slack, to achieve better overall energy reduction.

3. EXPERIMENTAL METHODOLOGY

This research is conducted with the Dynamic SimpleScalar simulator [Huang et al. 2003] and the Jikes Research Virtual Machine (RVM) [Appern et al. 1999]. We evaluate the proposed ACE management framework with the SPECjvm98 benchmark suite.

3.1 Simulation Environment

The Dynamic SimpleScalar (DSS) [Huang et al. 2003] 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 as a result of 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 [Brooks et al. 2000]. We assume that the operating frequency and voltage of the target processor are 1 GHz and 2 V, respectively. The baseline processor configuration is presented in Table I.

Table I. 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	2 K-entry combined predictor, 3-cycle misprediction penalty
Issue/commit width	4 instructions per cycle
Memory Hierarchy	
L1 I-cache	64 KB, 64B blocks, 2-way, LRU, 1-cycle hit latency
L1 D-cache	64 KB, 64B blocks, 2-way, LRU, 1-cycle hit latency
L2 unified cache	1 MB, 128B blocks, 4-way, LRU, 10-cycles hit latency
DTLB/ITLB	128 entries, fully set associative

3.2 Basic Block Vector-Phase Detection

We also implement the basic block vector (BBV) approach [Lau et al. 2005; Sherwood et al. 2003] in DSS, and compare it with our framework. BBV is shown to be one of the best phase-detection techniques [Dhodapkar and Smith 2003]. It partitions the dynamic instruction stream into contiguous fixed-length intervals and gathers dynamic basic block distributions through an array of counters, called the BBV, for each interval. Hence, at the end of each interval, the BBV obtain the code signature for the interval. A large Manhattan distance between two consecutive intervals' code signatures of indicates a phase change [Sherwood et al. 2003]. 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 basic block vector has 32 entries, as in [Sherwood et al. 2003]. Each BBV entry is 20 bits, so it will never overflow with 1 million sampling intervals. For each branch instruction executed, the lower five bits (excluding two least significant bits) of the branch PC indexes a BBV entry and increments the BBV 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 work [Lau et al. 2005; Sherwood et al. 2003]. A 25% threshold yields results that are comparable with those using a 20% threshold. On the other hand, using 12.5% threshold considerably increases the number of detected phases and affects the code coverage (described in Section 4.2) of BBV phases.

To give an advantage to the BBV approach, this BBV implementation does not compress BBV values to form a phase's signature, and thus loses no information because of signature compression [Sherwood et al. 2003]. Furthermore, this BBV implementation allows unlimited number of BBV phase signatures. However, this BBV implementation does not contain a next-phase predictor, since both [Lau et al. 2005] and our preliminary results confirm that using costly next-phase predictors provide only marginal benefits over no prediction.

3.3 Dynamic Optimization System

Jikes RVM is a research Java virtual machine (JVM) developed in IBM T. J. Watson Center [Appern et al. 1999]. It is written in Java. This enables the

optimization techniques to be applied to both the application code and the JVM itself. We use the 2.0.2 version of Jikes since Dynamic SimpleScalar is not compatible with the latest version.

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. In this research, all experiments are conducted using the optimization level one.

The original Jikes RVM uses a sampling method to detect program hotspots. Approximately every 10 ms, 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. However, this sampling approach may detect different hotspots in different runs of the same program and incurs in-deterministic results in our research. Moreover, this sampling approach may overlook a frequently executed procedure if the sampling method is not activated on the procedure enough times.

To avoid those problems, the Jikes baseline compiler is modified to adopt the invocation-based hotspot detection scheme used by other dynamic optimization systems, such as Sun's Java HotSpot VM and HP Dynamo [Bala et al. 2000]. Initially, the baseline compiler inserts profiling code in the entrance and exits of every procedure when it compiles the procedure. A procedure has only one entrance, but may have multiple exits, and the exiting profiling code is inserted right before each possible procedure exit. The profiling code in a procedure's entrance increments the counter associated with the procedure each time the procedure is invoked, and a procedure executed more than 30 times are deemed as a hotspot. The profiling code in procedure exits counts the number of instructions executed by the current invocation of the procedure. When a new hotspot is detected, its size is the average length of its first 30 invocations. Finally, when the newly detected hotspot is optimized by the optimizing compiler, the profiling code in procedure entry and exits are removed.

Currently only the optimizing compiler with the first compilation level is used. This prevents the possible disruptions caused by the use of multiple versions of the hotspot. The simplification allows us to focus on the implementation and evaluation of the proposed ACE management framework. Furthermore, we intentionally choose a large heap (200 MB) to reduce garbage collection activities. By doing so, execution is dominated by the application rather than Jikes RVM.

3.4 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. We run the SPECjvm98 benchmarks with the largest s100 data sets. Table II provides a summary of these SPECjvm98 benchmarks.

Table II. Description of SPECjvm98 Benchmarks

Benchmark	Description
compress	A popular LZW compression program
db	Data management benchmarking software written by IBM
jack	A real parser-generator from Sun Microsystems
javac	The JDK 1.0.2 Java compiler
jess	A Java version of NASA's popular CLIPS rule-based expert systems
mpegaudio	The core algorithm for software that decodes an MPEG-3 audio stream
mrtt	A dual-threaded program that ray traces an image file

4. CAN HOTSPOTS REPRESENT PROGRAM PHASES?

In this section, we try to find out whether program hotspots are a good representation of phases. Intuitively, since invocations of the same hotspots execute instructions from the same code sequence, differences among their run-time characteristics should be much smaller than those among different hotspots. Hence, by the definition of phases [Sherwood et al. 2003], hotspots should be a good representation of phases. To answer the question, we measure per-hotspot and interhotspot CoVs, and observe that differences among hotspots are much larger than those among invocations of the same hotspot. We also calculate the CoVs for BBV phases with 10 K and 1 M-instruction sampling intervals and note that hotspots' CoVs are comparable with BBV phases'. The results clearly demonstrate that program hotspots closely represent phases. This section also provides other run-time characteristics for both hotspots and BBV phases.

4.1 Run-Time Characteristics of Hotspots

In this work, we detect frequently executed (executing more than 30 times) procedures as hotspots. Table III presents the run-time characteristics of hotspots in SPECjvm98 benchmarks. In this paper, only the hotspots whose average sizes are longer than 1000 instructions are considered, since we are interested in large hotspots that are suitable for adapting CUs at their boundaries. Each program usually has hundreds of such hotspots. The average sizes of those hotspots are more than 14,000 instructions, indicating that many hotspots are very long. The fourth row of the table shows the percentages of instructions belonging to at least one hotspot and indicates that hotspots dominate program execution for all the benchmarks.

On average, the resulting hotspots execute at least 823 times. The proportion of the hotspot identification latency over whole program execution can be estimated by dividing *hot.threshold* (i.e., 30) by the average invocations per hotspot. Since a hotspot's average number of invocations far exceeds *hot.threshold*, the hotspot identification latency (i.e., the time take to identify a hotspot) takes less than 3.65% of overall program execution.

To characterize how well hotspots represent program phase behavior, Table III also provides the per-phase and interphase IPC coefficient-of-variations (CoVs) of hotspots. CoV equals the percentage of the standard deviation divided by the average and provides a relative measure of data dispersion compared to the mean. The per-phase IPC CoVs are IPC variations among

Table III. Run-Time Hotspot Characteristics of SPECjvm 98 Benchmarks

	comp	db	jack	javac	jess	mpeg	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 hotspots	299	316	470	685	434	386	363
Average hotspot size	81,645	75,648	14,941	23,774	77,841	70,231	18,617
% of code in hotspots	98.75	89.11	87.57	90.04	93.38	97.21	85.62
Average invocations per hotspot	823	1,105	13,091	5,983	2490	4,747	3,284
Hotspot identification latency (as % of total execution time)	3.65	2.71	0.23	0.50	1.20	0.63	0.91
Per-hotspot IPC CoV(%)	9.17	9.97	6.74	9.33	7.79	5.37	8.09
Interhotspot IPC CoV(%)	43.78	42.99	49.38	46.47	52.49	49.05	46.69

different invocations of the same hotspot, characterizing the homogeneity among different invocations of a hotspot. The interphase IPC CoVs are the variations of average IPCs of different hotspots; larger interphase IPC CoVs signify large differences between the characteristics of detected hotspots. More than 34% disparities between hotspots' per-phase and interphase CoVs demonstrate that the differences between hotspots are more prominent than those between invocations of the same hotspot, further confirming that hotspots are closely related with program behavior changes.

4.2 Comparison with Stable BBV Phases

In this work, hardware adaptation happens at phase boundaries, and temporal approaches need one sampling interval to identify a phase change. Hence, if a phase lasts only one interval, it is difficult to be tuned since the next interval will have distinct characteristics to the phase. In this respect, a hardware adaptation scheme should avoid adapt hardware configurations for such *transitional phases*. In comparison, a *stable phase* lasts two or more continuous intervals. It must be recognized that only stable phases can also improve the phase-detection hardware utilization and considerably increase phase-detection accuracy [Dhodapkar and Smith 2002; Lau et al. 2005]. By default, BBV phases studied in this paper are all stable.

BBV phases with different sampling interval sizes may exhibit distinct run-time characteristics. Programs phases are usually hierarchical, i.e., nested 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 run-time characteristics of large BBV phases. Moreover, since transitional phases cannot be used for CU adaptation, we count only stable phases. Table IV presents the run-time characteristics of stable BBV phases with 10 K and 1 M-instruction sampling intervals, respectively.

Table IV. Run-Time Characteristics of Stable BBV Phases with 10 K and 1 M Sampling Intervals

		comp	db	jack	javac	jess	mpeg	mtrt
10 K	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
	Interphase IPC CoVs(%)	48.03	50.91	44.31	35.59	44.35	49.28	49.08
1 M	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
	Interphase IPC CoVs(%)	20.05	33.32	20.07	24.87	26.11	38.26	23.96

All benchmarks have hundreds of 10 K-instruction BBV phases, but only tens of 1 M-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 10 K-instruction sampling intervals.

Similar to the hotspot results in Table III, Table IV 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 much of dynamic program code can benefit from CU adaptation. Overall, hotspots have better coverage than BBV phases.

For the two BBV configurations, 10 K-instruction phases have worse code coverage than 1 M-instruction phases. Intuitively, in terms of BBV phase detection, a fluctuation of branch frequency affects small 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 transitional phase than a large interval, which attributes to the poor coverage of stable 10 K-instruction BBV phases. For the 10 K-instruction BBV phases, three out of seven benchmarks have code coverage less than 50%. On the other hand, except javac, all benchmarks have good code coverage using 1 M-instruction BBV phases. On average, the code coverages of 10 K and 1 M-instruction BBV phases are 57 and 73%, respectively. Lau et al. [2005] propose to filter out *transition phases* that contain fewer intervals than a given phase count threshold. However, a nontransition phase in [Lau et al. 2005] may still be a transitional one, as defined above, given that the intervals in this phase are not continuous. Hence, the coverage results in the two papers are not comparable.

Table IV also gives the CoVs for the BBV phases. The 10 K-instruction BBV phases have larger per-phase CoVs than 1 M-instruction 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, 10 K-instruction BBV phases also have larger interphase CoVs than 1 M-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 hotspots (Table III) with BBV phases, we notice that both the per-phase and interphase CoVs of hotspots are between the corresponding small and large BBV phases, indicating

Fig. 1. Flowchart of the proposed ACE management framework based on a DO system.

that hotspots, with their variable sizes, can detect both fine and coarse-grain program phase changes.

In summary, small BBV phases are more sensitive to program phase changes than large BBV phases and thus are more accurate on representing program phase behavior. On the other hand, with small sampling intervals, the BBV approach's tuning overhead increases dramatically and the portions of program codes belonging to stable phases diminish. Both factors impair the BBV approach's performance.

5. ACE MANAGEMENT BASED ON DYNAMIC OPTIMIZATION

We have seen in Section 5 that program hotspots detected by a DO system accurately represents program phase behavior and, thus, can be used for efficient hardware adaptation. In this section, we develop an ACE management framework, based on a generic DO system, that can efficiently manage multiple CUs. Although the idea of integrating hardware adaptation with a virtual machine is not new [Dhodapkar and Smith 2002], to the best of our knowledge, this research is the first one that utilizes the inherent capabilities of a general DO system for efficient management of multiple configurable hardware resources.

Figure 1 shows the flowchart of the proposed management framework. In the figure, thin lines indicate program control flows, while thick lines represent data flows. Three main tasks are performed. Initially, the DO system monitors program execution and detects hotspots. After a hotspot is detected and JIT optimized, the DO system inserts tuning code at hotspot boundaries to

identify the energy-efficient hardware configuration for the hotspot during its subsequent invocations. After the tuning finishes, the JIT compiler replaces the tuning code with the code that automatically adapts to the hotspot's most energy-efficient configuration whenever it is invoked. The details of the framework are explained in the following subsections.

5.1 Hotspot Detection

Program *hotspots* are frequently executed code sequences, such as procedures [Appern et al. 1999; Java Technology] or basic block groups [Bala et al. 2000; Ebcioglu and Altman 1997; Dehnert et al. 2003]. To amortize the overhead of run-time translation and further improve performance, most DO systems apply high-cost, high-quality optimizations only on hotspots. For instance, the Jikes research virtual machine [Appern et al. 1999] uses a low-overhead sampling method to detect execution frequencies of procedures, which are then used to determine the level of optimizations that are applied on the procedures.

A DO system usually includes the following steps to detect and optimize hotspots. Initially, a program code block is interpreted [Bala et al. 2000; Ebcioglu and Altman 1997; Dehnert et al. 2003; Java Technology] or quickly translated and instrumented [Appern et al. 1999]. The execution frequency information of the code block is then gathered by the interpreter or the profiling code instrumented at hotspot boundaries and saved in the code block's corresponding entry in the DO database that stores run-time profiling information for the DO system. The information is then examined to find frequently executed code blocks as hotspots and advanced optimizations are applied on them.

The hotspot detection mechanism in the DO system can be used directly for phase identification. Wu et al. [2004] indicate that the run-time characteristics of hotspots are usually stable throughout program execution; Huang et al. [2003] and Merten et al. [Merten et al. 2001] observe that program phase behavior is closely related to hotspot invocations. Hence, tuning and reconfiguring CUs at hotspot boundaries will accurately adapt to program changes.

5.2 CU Decoupling and Hotspot Tuning

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

5.2.1 CU Decoupling. Since hotspots can be of diverse sizes, we can adapt low-overhead CUs at the boundaries of small hotspots, while adapting high-overhead CUs at boundaries of large hotspots. To do this, we match CUs with the hotspots that have similar sizes to the CUs' reconfiguration interval sizes. This technique is called CU decoupling, since it decouples the reconfiguration of different CUs in a multiple-CU system and allows multigrain configuration of CUs.

The properties of hotspots enhance the effectiveness of CU decoupling. Hotspots are nested, i.e., a large hotspot usually contains many small hotspots. Hence, when the small hotspots tune low-overhead CUs, those CUs are automatically tuned for the outside large hotspot. Consequently, adapting different CUs at different hotspots 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.

5.2.2 Hotspot Tuning. After a hotspot is detected and JIT optimized, the subset of CUs is chosen for the hotspot, such that the CU's reconfiguration interval sizes match the hotspot size (an implementation will be given in Section 6.1). A list of configuration combinations of the selected CUs is then created and added to the hotspot'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 hotspot and the profiling code at all exit points of the hotspot. Immediately after the hotspot's invocation, the tuning code fetches the configuration pointed to by the list index and increments the index, adapting the hardware according to the fetched configuration. When leaving the hotspot, the hotspot's performance characteristics under the current configuration are gathered. The configurations applicable to the hotspot are thus tested one-by-one until all configurations are tested. The most energy-efficient configuration is then selected to complete the hotspot's tuning process.

Since hotspots are nested, it is possible that when one hotspot requests tuning, the other one is being tuned. Intuitively, the configuration change of one CU affects the adaptation of the other CU. To minimize such interference between CUs, this framework allows only one hotspot to be tuned at a time. To enforce this rule, a hardware bit stores the tuning status. The tuning code in hotspot entries checks the bit. If the bit is not set, indicating that no other hotspots are being tuned, then the tuning code sets the bit and begins to tune the hotspot. Otherwise, the tuning code does nothing.

5.3 Hardware Reconfiguration

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

In addition, the profiling code at a hotspot's exit is replaced by the sampling code that occasionally gathers performance statistics to detect the performance change between the hotspot's current and prior invocations. A large performance change indicates that the hotspot's behavior may have altered. Consequently, the hotspot is re-tuned. As observed by [Wu et al. 2004], run-time characteristics of hotspots are usually stable throughout program execution, and thus such retunings should be rare.

5.4 Hardware Support

The proposed scheme is mainly a software approach. It relies on the underlying DO system to detect and adapt program hotspots. However, some minimal

hardware support is needed. We assume 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. For safety, one possible approach is to request a system call for hardware configuration changes. The overhead of invoking the system call can be accounted to decide the proper CU reconfiguration interval sizes.

We also maintain one hardware counter for each CU to hold 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. With this hardware support, the proposed framework is freed from the burden of maintaining the minimal reconfiguration interval for each CU.

5.5 Comparison 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 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 [Lau et al. 2005; Sherwood et al. 2003], 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 considerably affecting performance. Hence, high prediction accuracy is imperative for such mechanisms.

In comparison, in a DO-based system, only new hotspots need to be detected and recurring hotspots are identified immediately. Hence, a hotspot's detection overhead is a one-time cost and thus can be diminished by long program execution. Moreover, since hotspots are often nested, detections of new hotspots are often overlapped, further reducing the overall hotspot identification cost. The run-time characteristics in Table III shows that although hotspots take more time than BBV phases to be recognized, it does not pose a big burden to the hotspot approach.

This hotspot-based framework is essentially a software positional approach. Differing from the original positional approach [Huang et al. 2003], the proposed framework detects hotspots instead of large procedures. The frequent-invocation nature of hotspots ensures that the most energy-efficient hardware configuration of a hotspot can be applied enough times for high benefit, while the positional approach can not enjoy this feature from the large procedures it uses. Furthermore, as with temporal approaches, the original positional approach also requires significant 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. [2003] 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 run-time profiling. In contrast, DO-based framework incurs run-time profiling overhead, although such overhead is minimal since the framework leverages the existing infrastructure of the underlying DO system for hardware adaptation. On the other hand, the reconfigurations chosen by the offline tools may not yield sufficient energy reduction on systems with different CUs or with CUs that have different sets of configurations to the target system.

5.6 Summary of Advantages

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

- Prompt recurring phase identification. By instrumenting hotspot headers, the framework can identify all previously seen hotspots with zero latency and, thus, needs no phase prediction at all.
- Reduced tuning latency. Since we configure only a subset of CUs in each hotspot, the tuning latency is greatly reduced.
- Adapting to hierarchical phase changes. By detecting nested and multigrain hotspots, tuning and reconfiguring CUs at those hotspots' boundaries accurately adapts to hierarchical phase changes.
- Differentiating low- 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 hotspots of any sizes, this approach works efficiently for workloads with diverse run-time characteristics and CUs with disparate reconfiguration overheads.

Finally, the proposed framework is a software-based approach, which is more flexible than hardware-based approaches. For instance, it can decide the performance loss budget and energy reduction on a per-program basis, based on the program's priority. A low-priority program can choose a high-performance loss budget, thus allowing CUs to adapt to aggressive configurations for high-energy reduction. Similarly, a time-critical program may disable hardware configuration to achieve high performance.

6. IMPLEMENTATION AND EVALUATION OF THE DO-BASED FRAMEWORK

In this section, we first implement the framework on Jikes RVM. To evaluate the proposed framework, we compare it with a system that uses the BBV

phase-detection technique [Sherwood et al. 2003] and the tuning algorithm prescribed in [Dhodapkar and Smith 2002]. Since both techniques are the best among their respective alternatives, this combination should be the best technique that prior literature can contribute. In this experiment, the two resource adaptation approaches adapt five CUs described in Section 6.1.

The BBV technique used in this paper does not use the phase-prediction mechanisms presented in Lau et al. [2005] and Sherwood et al. [2003]. 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 and both Lau et al. [2005] and our preliminary results confirm that using next-phase predictors provide only marginal benefits over no prediction. Hence, in this research, the next BBV phase prediction is not implemented. In contrast, the hotspot approach always identifies upcoming phases immediately and does not need next-phase prediction.

6.1 Implementation of Hardware Adaptation

6.1.1 DO-Based ACE Management Framework. It is fairly easy to implement the functionalities of hardware tuning and reconfiguration in Jikes RVM. As described in Section 5, hardware tuning and reconfiguration are performed after hotspots are detected and JIT optimized. The Jikes optimizing compiler is modified to insert the tuning and configuration code into hotspot entrances and Jikes' global data structure is also modified to store the necessary information for the hardware tuning and reconfiguration, such as a table containing the available hardware configurations or a table containing each tuned hotspot's optimal one.

Currently only the optimizing compiler with the first compilation level is used. This prevents the possible disruptions caused by the use of multiple versions of the hotspot. However, there is virtually no problem for the framework to work with an adaptive optimization system. A hotspot may behave differently when it is optimized at different levels. Such behavior changes can be easily captured by retuning a hotspot whenever it is recompiled. This scheme may be further optimized if we have a better understanding of the impact of JIT optimization on hardware adaptation. We feel this is an important direction for future research.

6.1.2 Configurable 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 V). Each CU has four different sizes. The power model is augmented to reflect the run-time 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 run-time 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,

Table V. CUs and Their Adaptation Parameters

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

called the unit's reconfiguration interval (presented in the third column of Table V).

The hardware adaptation framework proposed in this work adapts CUs at program hotspot boundaries. To avoid testing all CUs at the same time, each hotspot tests only a subset of CUs. The fourth column of Table V lists the size ranges of hotspots at whose boundaries corresponding CUs are adapted. As described in Section 5.2, during hotspot detection, the number of instructions executed by a procedure's each invocation is counted. A hotspot's size is the average length of its first 30 invocations. Hotspots that configure the issue queue and the reorder buffer (*IQ-ROB hotspots*) are between 1 and 100 K instructions long; hotspots adapting the L1I and L1D caches (*L1 hotspots*) are between 100 K and 1 M instructions long, while *L2 hotspots* are at least 1 M instructions long.

6.1.3 BBV-Based Hardware Adaptation Scheme. 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 3.2. In the DO-based hardware adaptation scheme, CUs with diverse reconfiguration overheads are adapted at different hotspot boundaries (Section 6.1.1), which essentially reduce the tuning latency of each CU. To allow a fair comparison of the two hardware adaptation approaches, the BBV approach should also be able to decouple the tuning of different CUs. This can be implemented in multiple ways. This work uses multiple BBV, each corresponding to a sampling interval size, which detects a different set of BBV phases. Similar to the DO-based framework, a CU's adaptation occurs 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 V shows the sampling interval sizes that correspond to the CUs.

Note that in Table V, issue queue and reorder buffer are adapted at boundaries of 10 K-instruction BBV phases, instead of 1 K-instruction BBV phases, although the CUs have a reconfiguration interval size of 1 K instruction. This is mainly because we found that for the programs evaluated in this work, 1 K-instruction stable BBV phases usually have much poor code coverage. As explained in Section 4.2, a fluctuation of branch behavior has a larger impact on a small interval's phase classification than on a large interval, since the latter contains much more branches to minimize the impact of individual branches. Hence, 10 K-instruction phases have a better code coverage and thus higher energy reduction on IQ and ROB, than 1 K-instruction phases.

Table VI. Run-Time Characteristics of the Three Types of Hotspots on SPECjvm 98 Benchmarks

	comp	db	jack	javac	jess	mpeg	mtrt
Number of IQ-ROB hotspots	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 hotspots	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 hotspots	22	29	31	33	30	23	21
L2 coverage	73.80	87.90	56.90	80.00	84.30	95.70	82.60

6.2 Energy Reduction via Adapting Five Configurable Units

To demonstrate the scalability and efficiency of the hotspot-based approach on adapting multiple CUs, the two resource adaptation schemes manage all the five CUs for energy reduction and their run-time characteristics and energy reduction results are compared. For the hotspot approach, the five CUs are adapted at the boundaries of IQ-ROB, L1, and L2 hotspots, respectively. To find the most energy-efficient configuration, each L2 hotspot corresponds to only one CU, the L2 cache, and needs to test four L2 configurations to find the optimal one. On the other hand, since an IQ-ROB hotspot or a L1 hotspot adapts two CUs, for each hotspot, it has to test sixteen combinatorial configurations of the two CUs to find the optimal one. Similarly, a 10 or a 100 K BBV phase needs to test 16 combinatorial configurations, while a 1 M BBV phase tests only 4 configurations. 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%.

6.2.1 Run-time Characteristics. Table VI counts the number of IQ-ROB, L1, and L2 hotspots. As shown in the table, there are much more small hotspots than large ones. Since low-overhead CUs are adapted at the boundaries of small hotspots, this property, as well as the fact that small hotspots are invoked more frequently than large hotspots, guarantees that low-overhead CUs are adapted more frequently than high-overhead CUs.

Table VI also gives the *coverage* (i.e., the portion of dynamically executed instructions within the hotspots) results for the three types of hotspots. As shown in the table, all the IQ-ROB, L1D, and L2 hotspots have good coverage across benchmarks. Good coverage and numerous hotspot invocations indicate that CU decoupling does not sacrifice each CU's reconfiguration opportunity. For 10 K- and 1 M-instruction BBV phases, their phase counts and code coverage results are listed in Table IV.

Table VII presents the number of tuning attempts made (tunings) and the number of times the most energy-efficient configuration is applied (reconfigurations) for both the hotspot and the BBV algorithms. Because of the use of the decoupling strategy to adapt the CUs, both approaches have more reconfigurations than tunings, indicating that both approaches have enough opportunities to apply the optimal configurations for energy reduction. Furthermore, since both approaches adapt low-overhead CUs at boundaries of fine-grain hotspots or BBV phases, low-overhead CUs are reconfigured more frequently than

Table VII. Run-Time Characteristics of Adapting the Five CUs

		comp	db	jack	javac	jess	mpeg	mtrt
	IQ-ROB tunings	3285	3664	5606	8704	5149	4784	4244
	IQ-ROB reconfigurations	477705	1100160	6823045	4295271	1265328	234213	1849379
	L1 tunings	992	898	1296	1663	992	1024	1149
	L1 reconfigurations	1789	2138	29461	45359	9461	42397	47269
	L2 tunings	82	114.1	124	132	117	87	84
	L2 reconfigurations	838	1254	4349	2897	1283	8452	863
BBV	10 K tunings	6516	5259	8669	18264	9549	12513	9042
	10 K reconfigurations	111686	22158	137826	119579	40823	108146	31917
	100 K tunings	1782	1359	2463	3594	2718	1413	1785
	100 K reconfigurations	7615	2659	5385	10557	7091	1563	6537
	1 M tunings	170	109	151	275	175	104	163
	1M reconfigurations	636	242	329	880	299	346	150

high-overhead CUs. Hence, low-overhead CUs can exploit fine-grain phase changes for energy reduction. Comparing each approach's tuning counts, for a given CU (i.e., L2 cache) or CU pair (i.e., IQ-ROB or L1 caches), the hotspot approach usually conducts relative fewer tunings than the BBV approach, although the differences are not large enough to affect the energy reduction of the BBV approach. This is mainly because in the same granularity, there are more BBV phases than hotspots (Table IV and Table VI). The impact of the long tuning process on long-running applications will not be as prominent as on short-running applications.

6.2.2 Energy Reduction. The energy reduction results are given in Figure 2. For both the BBV and the hotspot algorithms, 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.

In terms of energy reduction, the BBV approach is slightly better than the DO-based approach on the L1I and L2 caches, slightly worse on the L1D cache, but performs much worse on the issue queue, reorder buffer, and the L1D cache. The poor performance of the BBV approach on issue queue and reorder buffer is mainly because of the poor code coverage of the BBV approach with 10 K-instruction sampling intervals (Table V). Furthermore, by adapting issue queue and reorder buffers at a coarse granularity (10 K instructions) than their optimal reconfiguration interval size (1 K instructions), the BBV approach loses tuning opportunities for the two CUs. However, we find that using 1 K-instruction sampling intervals yields poorer performance than using 10 K-instruction sampling intervals, mainly because the code coverage of 1 K-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 hotspot approach reduces the CUs' energy consumption by 33, 28, 37, 31, and 45%.

6.2.3 Performance impact. In this experiment, the IPC degradation threshold is 5% for both approaches, and the performance degradation results

Fig. 2. Percentage of energy consumption reduced by adapting the five CUs (BBV sampling interval sizes: 10 K for issue queue and reorder buffer, 100 K for L 1 caches, and 1 M for L2 cache).

Fig. 3. Performance impact of the resource adaptation schemes on the 5-CU system.

are presented in Figure 3. For the BBV approach, most benchmarks yield performance losses well below the 5% budget, mainly because during most of program execution, the CUs are in their less aggressive configurations. On the other hand, the hotspot approach's IPC degradation rates are around 5% for all the benchmarks. On average, the BBV and the hotspot approaches yield IPC degradation rates of 5.0 and 4.9%, respectively. The results in Sections 5.2.2 and 5.2.3 confirm that the DO-based approach achieve comparable energy reduction with one of the best prior approaches.

In summary, this section demonstrates that in terms of energy reduction, the DO-based adaptive computing environment manage framework achieves comparable results to one of the best prior approaches, which, in turn, further confirms that program hotspots accurately capture program phase behavior. More importantly, being able to detect hotspots of multiple granularities, the framework can efficiently manage multiple CUs with diverse reconfiguration costs and achieve high overall energy reduction.

Compared with hardware-based or static hardware adaptation schemes, the DO-based framework has its advantages as well as limitations. By leveraging the existing hotspot detection and optimization infrastructure of a dynamic optimization, the DO-based framework requires much simpler hardware supports 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.

7. CONCLUSION

In ACE, efficient management of the configurable resources is vital for maximizing the benefit of resource adaptation. First, we demonstrate that method-based hotspots detected by a dynamic optimization system closely represent program phase behavior. The other contribution of this paper is that we demonstrate how inherent capabilities of a dynamic optimization system can be synergistically employed for efficient management of adaptive computing environments. Utilizing existing DO hotspot detection and optimization mechanisms, the proposed technique accurately detects program behavior at varying granularities, providing us the opportunity to significantly reduce the overhead associated with adaptation decisions. By matching each hotspot with a subset of available configuration units, we reduce the number of tested configurations while searching for the most energy-efficient one, thereby reducing the tuning process significantly.

Dynamic optimization systems become increasingly popular. For instance, in the next-generation Windows operating system, Windows Vista, most applications and OS services will be managed by the .NET framework, essentially a DO system similar to a Java virtual machine. Those existing DO systems can utilize our framework for better hardware/software integration and optimizations. On the other hand, the benefits of using the proposed framework in systems without such infrastructure may be affected by the extra time and energy spent on hotspot detection and binary rewriting.

We implement the proposed scheme in a state-of-the-art JVM and evaluate for the SPECjvm98 benchmark suite with the ACE having five CUs (issue queue, reorder buffer, L1D and L1I caches, and L2 cache). Our technique reduces the energy consumption of the L2 cache by as much as 52%, whereas the average energy reductions are between 28 and 45% for all the CUs.

The proposed framework also demonstrates the benefit of integrating software adaptability with hardware adaptability. We envision several new optimization opportunities being enabled by the integration. For example, one could use the JIT compiler in the DO system to provide a good estimate for the resource configuration required for this hotspot through appropriate code analysis. Such a feature could potentially completely eliminate the tuning latency and overhead seen in all existing ACE schemes. In the future, we plan to investigate this and other such avenues for improving the performance of DO-based ACE.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Brad Calder and the anonymous reviewers for their valuable comments.

REFERENCES

- ALBONESI, D. 1998. Dynamic IPC/clock rate optimization. In *Proceedings of the 25th International Symposium on Computer Architecture*, 282–292.
- ALBONESI, D. 2000. Selective cache ways. On-demand cache resource allocation. In *Journal of Instruction-Level Parallelism* 2. (<http://www.jilp.org/vol2/index.html>).

- APPERN, B., ATTANASIO, D., BARTON, J., COCCHI, A., LIEBER, D., SMITH, S., AND NGO, T. 1999. Implementing Jalapeno in Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 314–324.
- BAHAR, I. AND MANNE, S. 2001. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th International Symposium on Computer Architecture*. 218–229.
- BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference of Programming Language Design and Implementation*. 1–12.
- BALASUBRAMONIAN, R., ALBONESI, D., BUYUKTOSUNOGLU, A., AND DWARKADAS, S. 2000. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *Proceedings of the 33rd International Symposium on Microarchitecture*, 245–257.
- BARAZ, L., DEVOR, T., ETZION, O., GONDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. 2003. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on itanium-based systems. In *Proceedings of the 36th International Symposium on Microarchitecture*. 191–201.
- BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*. 83–94.
- DAS A., LU, J., AND HSU, W. 2006. Region monitoring for local phase detection in dynamic optimization systems. In *Proceedings of the 2006 International Symposium on Code Generation and Optimization (CGO)*. 124–134.
- DEHNERT, J., GRANT, B., BANNING, J., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSO, J. 2003. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*. 15–24.
- DHODAPKAR, A. AND SMITH, J. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th International Symposium on Computer Architecture*. 233–244.
- DHODAPKAR, A. AND SMITH, J. 2003. Comparing program phase detection techniques. In *Proceedings of the 36th International Symposium on Microarchitecture*. 217–227.
- DROPSHO, S., BUYUKTOSUNOGLU, A., BALASUBRAMONIAN, R., ALBONESI, D., DWARKADAS, S., SEMERARO, G., MAGKLIS, G., AND SCOTT, M. 2002. 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.
- DROPSHO, S., SEMERARO, G., ALBONESI, D., MAGKLIS, G., AND SCOTT, M. 2004. Dynamically trading frequency for complexity in a GALS microprocessor. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, 157–168.
- EBCIOGLU, K. AND ALTMAN, E. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture*. 26–37.
- FOLEGNANI, D. AND GONZALEZ, A. 2001. Energy-effective issue Logic. In *Proceedings of the 28th International Symposium on Computer Architecture*. 230–239.
- HU, S., VALLURI, M., AND JOHN, L. 2005. Effective adaptive computing environment management using a dynamic optimization system. In *Proceedings of 2005 International Symposium on Code Generation and Optimization*. 63–73.
- HUANG, M., RENA, J., AND TORRELLAS, J. 2003. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the 30th International Symposium on Computer Architecture*. 157–168.
- HUANG, M., RENA, J., YOO, S., AND TORRELLAS, J. 2000. A framework for dynamic energy efficiency and temperature management. In *Proceedings of the 33rd International Symposium on Microarchitecture*. 202–213.
- HUANG, X., MOSS, J., MCKINLEY, K., BLACKBURN, S., AND BURGER, D. 2003. Dynamic SimpleScalar: Simulating Java virtual machines. *The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-03-03*.
- IEEE Computer. July 2004. Special issue on Adaptive Computing 37, 7.
- JAVA Technology, <http://java.sun.com>.
- JAVA HotSpot Technology, <http://java.sun.com/products/hotspot/>

- IYER, A. AND MARCULESCU, D. 2002. Microarchitecture-level power management. In *IEEE Transactions on VLSI Systems* 10, 3, 230–239.
- KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. 1997. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th International Symposium on Microarchitecture*. 184–193.
- LAU, J., SCHOENMACKERS, S., AND CALDER, B. 2005. Transition phase classification and prediction. In *Proceedings of the 11th International Symposium on High Performance Computer architecture*. 278–289.
- LAU, J., PERELMAN, E., AND CALDER, B. 2006. Selecting software phase markers with code structure analysis. In *Proceedings of the 2006 International Symposium on Code Generation and Optimization (CGO)*. 135–146.
- MAGKLIS, G., SCOTT, M., SEMERARO, G., ALBONESI, D., AND DROPSHO, S. 2003. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *IEEE Micro* 23, 6, 62–68.
- MERTEN, M., TRICK, A., BARNES, R., NYSTROM, E., GEORGE, C., GYLLENHALL, J., AND HWU, W. 2001. An architectural framework for runtime optimization. In *IEEE Transactions on Computers* 50, 6, 567–589.
- Microsoft. NET technology, <http://www.microsoft.com/net/>
- PEREIRA, C., LAU, J., CALDER, B., AND GUPTA, R. 2005. Dynamic phase analysis for cycle-close trace generation.
- PONOMAREV, D., KUCUK, G., AND GHOSE, K. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th International Symposium on Microarchitecture*. 90–101.
- SASANKA, R., HUGHES, C., ADVE, S. 2002. 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.
- SHEN, X., ZHONG, Y., AND DING, C. 2004. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming, Languages, and Operating Systems*. 165–176.
- SHERWOOD, T., SAIR, S., AND CALDER, B. 2003. Phase tracking and prediction. In *Proceedings of the 30th International Symposium on Computer Architecture*. 336–349.
- SHERWOOD, T., PERELMAN, E., HAMERLY, G., SAIR, S., AND CALDER, B. 2003. Discovering and exploiting program phases. In *IEEE MICRO*, 23, 6, 84–93.
- SHUF, Y., SERRANO, M., GUPTA, M., AND SINGH, J. 2001. 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*. 194–205.
- SPECjvm98 Benchmarks, <http://www.spec.org/osg/jvm98>.
- WU, Y., BRETERNITZ, M., QUEK, J., ETZION, O., AND FANG, J. 2004. The accuracy of initial prediction in two-phase dynamic binary translators. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*. 227–238.

Received August 2005; revised March 2006; accepted July 2006