Effective Adaptive Computing Environment Management via Dynamic Optimization

Shiwen Hu Madhavi Valluri Lizy Kurian John
Department of Electrical and Computer Engineering
The University of Texas at Austin
{hushiwen, valluri, ljohn}@ece.utexas.edu

Abstract

To minimize the surging power consumption of microprocessors, adaptive computing environments (ACEs) where microarchitectural resources can be dynamically tuned to match a program's runtime requirement and characteristics are becoming increasingly common. Adaptive computing environments 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 a scheme for efficient management of multiple configurable units, utilizing the inherent capabilities of dynamic optimization systems. Most dynamic optimizers typically detect dominant code regions (hotspots). We develop an ACE management scheme where hotpot boundaries are used for phase detection and adaptation. Since hotspots are of variable sizes and are often nested, program phase behavior which is hierarchical in nature is automatically captured in this technique.

To demonstrate the usefulness and effectiveness of our framework, we use the proposed framework to dynamically adapt the sizes of L1 data and L2 caches that have different reconfiguration latencies and overheads. Our technique reduces L1D and L2 cache energy consumption by 47% and 58%, while a popular previously proposed technique only achieves reduction of 32% and 52% respectively.

1. Introduction

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 [1][2][4][6][12][18][22][27]. An application's execution usually passes through phases with varying runtime characteristics and hardware requirements. Phase boundaries are thus suitable points for resource reconfigurations. 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 [6][9][15][17][20][22][24] or code positions [14]. Usually, upon a phase change, the schemes test all hardware configurations and select the most energy-efficient one.

To achieve more energy reduction, an adaptive computing environment (ACE) usually has multiple configurable units (CUs). Typical configurable units include issue queue [12][22], reorder buffer [22], instruction and data caches [2][6][9], pipelines [4], filter cache [18], and function units [4], and each CU may have multiple fixed settings (e.g. four different cache sizes). Hardware units can hardly be adapted individually. With more configurable units, the total number of combinatorial configurations increases dramatically. Hence, the straightforward tuning strategy of testing all combinatorial configurations results in long tuning process and higher tuning overhead, and impairs performance.

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 [19], IBM DAISY [11], HP Dynamo [5], Intel IA32EL [7], Java virtual machines [3][29], and Microsoft .NET's CLR [30]. To amortize the overheads of runtime translation and further improve performance, most DO systems apply high-cost, high-quality optimizations only on frequently executed code sequences (hotspots). It has been shown that hotspots usually have stable runtime characteristics throughout program execution [26], and closely represent program behavior changes [14][21]. Therefore, DO systems are good platforms for adaptive computing environment management.

In this paper, we propose a scheme for efficient management of multiple configurable units based on a generic dynamic optimization 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. Intuitively, they closely represent hierarchical phase behavior. 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 adaptive computing environment by adjusting the granularity of adaptation based on each CU's reconfiguration cost. This CU decoupling strategy significantly reduces the tuning process, and achieves better balance of benefit/overhead for each configurable hardware resource.

To demonstrate the usefulness and effectiveness of our framework, we implement and evaluate the proposed ACE management framework using Jikes Research Virtual Machine [3] and Dynamic Simplescalar [16]. Performance is evaluated on the SPECjvm98 benchmark suite [28]. Our target configurable units are the L1 data cache and the L2 cache. Our technique reduces L1D and L2 cache energy consumption by 47% and 58%, while a popular previously proposed technique only achieves reduction of 32% and 52% respectively.

The main contribution of this paper is that it demonstrates how inherent capabilities of a dynamic optimization system can be synergistically employed for efficient management of adaptive computing environments. The other contributions are the following:

- The proposed scheme automatically detects hierarchical phase behavior of programs, and does not complicate hardware design.
- It shows how multiple hardware resources with varying reconfiguration overheads can be managed simultaneously in an efficient fashion.
- Significant power saving over one of the best performing phase adaptation schemes is observed.

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 Section 5, and Section 6 concludes the paper.

2. Hardware resource adaptation

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 the units. In this section, we first introduce the terms reconfiguration overheads and intervals of CUs that are used throughout the paper. Then, we 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

Changing a hardware unit's setting on the 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 [9]. 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 configurable unit's reconfiguration interval can vary from thousands (e.g. reorder buffer [22]) to millions (e.g. caches [24]) of instructions/cycles.

2.2. 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 large reconfiguration overheads, 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 [14]. In temporal approaches, dynamic execution stream is divided into sampling intervals with fixed or varying sizes. At the end of each sampling interval, characteristics, such as IPC [12][22], conditional branch counter [6], occupancy [22], basic block vectors [20][24], instruction working sets [9], and hardware-detected hotspots [21], 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 [20][24] is shown to be one of the best [10]. It gathers dynamic basic block distributions through an array of counters, and then computes the Manhattan distances of BBVs to detect phase changes.

Phases can be classified into stable and transitional ones [9]. Transitional phases have short lifetimes (e.g. last only one sampling interval), rarely recur, and are thus difficult to be tuned. Recognizing only stable phases that last two or more continuous intervals can improve the phase detection hardware utilization and increase phase detection accuracy considerably [9]. Figure 1 shows the distributions of stable and transitional phases of SPECjvm 98 benchmarks. The phases are identified by the BBV method with parameters

described in Section 4.1. As demonstrated by *javac*, ignoring transitional phases may considerably reduce the coverage of resource adaptation. In [20], Lau et al propose to filter out transition phases also.

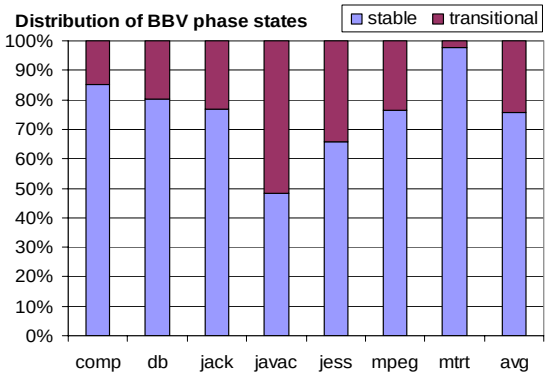


Figure 1. Distribution of stable/transitional phases of SPECjvm 98 benchmarks (A phase is stable if it has two or more successive sampling intervals; otherwise it is an unstable phase)

Differing from temporal approaches that detect phases at successive sampling intervals, the positional approach [14] 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 [14]. 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 [10]. Recently, Shen et al [23] 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. Hardware tuning

The tuning algorithms used in previous resource adaptation schemes [6][14][15][17] are similar in nature. In general, after a phase is found, the tuning algorithm tests different configurations of CUs in successive sampling intervals (the temporal approaches), or successive invocations of the same code (the positional approach). The tuning process completes when all the configurations have been tested or a performance threshold is reached. 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 due to application of mostly sub-optimal configurations. Second, because of the CU reconfiguration overhead, longer tuning process incurs higher overhead.

With multiple configurable units, the straightforward tuning strategy of testing all combinatorial configurations becomes inefficient. The length of the tuning process is proportional to the number of combinational configurations, which increases exponentially with more CUs. Moreover, in temporal approaches, the sampling interval size must be chosen to accommodate the largest CU reconfiguration interval. Thus, all CUs are adapted at the same pace, although some low-overhead CUs can be adapted more frequently, leading to lost reconfiguration opportunities.

3. ACE management using a dynamic optimization system

We have seen in Section 2 that 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, we develop an adaptive computing environment management framework based on a generic dynamic optimization system, and present it in this section. Although the idea of integrating hardware adaptation with a virtual machine is not new [9], to the best of our knowledge, this paper is the first one that utilizes the inherent capabilities of a general DO system for efficient management of multiple configurable hardware resources.

Figure 2 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.

3.1. Hotspot detection

Program hotspots are frequently executed code sequences, such as procedures [3][29] or basic block groups [5][11][19]. To amortize the overheads of runtime translation and further improve performance, most DO systems apply high-cost, high-quality optimizations only

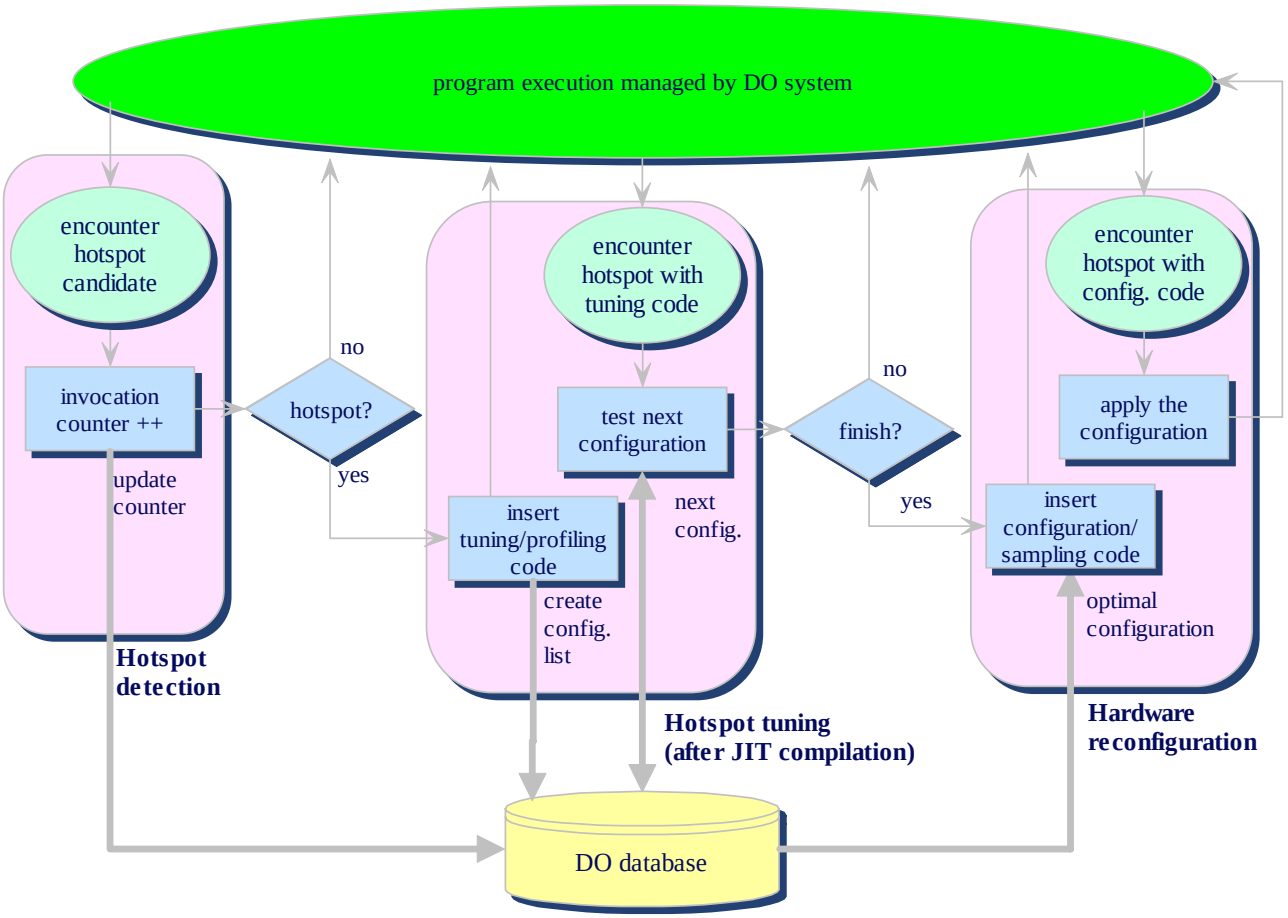


Figure 2. Flowchart of the proposed ACE management framework based on a DO system

on hotspots. For instance, the Jikes research virtual machine [3] 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 [5][11][19][29] or quickly translated and instrumented [3]. 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 runtime 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 [26] indicate that the runtime characteristics of hotspots are usually stable throughout program execution, and Huang et al [14] and Merten et al [21] observe that program phase behavior is closely related with hotspot invocations. Hence, tuning and reconfiguring CUs at hotspot boundaries will accurately adapt to program changes.

3.2. CU decoupling and hotspot tuning

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

3.2.1. CU decoupling. In temporal approaches, the sampling interval size must be equal to or larger than the largest reconfiguration interval of all CUs (Section 2.3). This limits the minimum granularity at which hardware can be adapted, leading to significant loss of adaptation opportunities.

In contrast, hotspots can be of diverse sizes. Hence, we can adapt low-overhead CUs at boundaries of small hotspots, while adapting high-overhead CUs at boundaries of large hotspots. This technique is called *CU decoupling* since it decouples the reconfigurations of different CUs in a multiple-CU system. In this paper, we examine configurable L1D/L2 caches with reconfiguration intervals of 100K/1M instructions respectively. Hence, the hotspots that can adapt the L1D cache (called *L1D hotspots*) have

Table 1. Comparing DO-Based ACE management scheme with temporal approaches

Type	Temporal approaches	DO-based approach
New phase identification latency	***At least one sampling interval***	Hotspot invoked hot_threshold times
Recurring phase identification latency	At least one sampling interval	***None***
Tuning latency	All configurations are tested	***A subset of all configurations are tested***

average sizes between 50K to 500K instructions, while the L2D hotspots are longer than 500K instructions.

The effectiveness of CU decoupling relies on the properties of hotspots. Hotspots are typically nested, i.e., a large hotspot usually contains many small hotspots. Hence, when those 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.

3.2.2. Hotspot tuning. After a hotspot is detected and JIT optimized, the subset of CUs is chosen for the hotspot, with the reconfiguration intervals of those CUs being in the same range as the hotspot size. Then, a list of configuration combinations of the selected CUs is 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, and then adapts 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 or the performance is worse than *performance_threshold* (e.g. 2% IPC degradation). The most energy-efficient configuration is then selected to complete the hotspot's tuning process.

Contrary with previous tuning algorithms that test all interdependent CUs, our tuning algorithm adapts only a subset of all CUs for each hotspot. Consequently, this technique significantly reduces the tuning process, and allows multi-grain adaptation which further improves performance.

3.3. 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. There will be no further tuning latency or phase identification latency incurred by the hotspot.

Additionally, the profiling code at a hotspot's exit is replaced by the sampling code which 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 tuned again. As observed by [26], runtime characteristics of hotspots are usually stable throughout program execution, and thus such re-tunings are 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 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. 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.

We 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.

3.5. Comparison with prior approaches

Table 1 qualitatively compares the temporal approaches and the proposed framework with their respective latencies. The better performing approach for each metric is emphasized in bold italics. The DO-based approach exceeds the temporal approaches for two out of three.

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,

Table 2. Baseline configuration of the simulated system (The L1D and L2 cache values in the parenthesis are the sizes and reconfiguration intervals of the configurable units)

CPU (1000M Hz with 2V)		Memory Hierarchy	
Instruction window	64-IFQ, 64-RUU, 32-LSQ	L1 I-cache	64KB, 64B blocks, 2-way, LRU, 1 cycle hit latency
functional units	4 intALU, 2 IntMult/Div,4 FPALU, 2 FPMult/Div	L1 D-cache	64KB (64KB/32KB/16KB/8KB, 100K-instruction reconfiguration interval), 64B blocks, 2-way, LRU, 1 cycle hit latency,
Branch predictor	2K-entry combined predictor, 3-cycle misprediction penalty	L2 unified cache	1MB (1MB/512KB/256KB/128KB, 1M-instruction reconfiguration interval), 128B blocks, 4-way, LRU, 10 cycles hit latency
Issue/Commit width	4 instructions per cycle	DTLB/ITLB	128 entries, fully set associative

regardless the length of program execution. Recurring phase identification latencies can be reduced by next phase detection mechanisms [20][24], which predict what the next phase will be and when it will occur. However, incorrect predictions cause unnecessary or wrong adaptations and subsequent rollbacks 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 hotspots need to be detected, and recurring hotspots are identified immediately and needs no prediction. Hence, a hotspot's detection overhead is a one-time cost, and 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.

More importantly, since the DO-based approach can have phases of any length, it enables CU decoupling, which significantly reduces the tuning process. These benefits are difficult to achieve in temporal approaches that use fixed-size sampling intervals.

This hotspot-based framework is essentially a software positional approach. Differing from the original positional approach [14], 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.

3.6. Summary of advantages

Utilizing existing DO services, 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. In return, the most energy-efficient configuration can be applied more times for improved performance.
- **Hierarchical phase change detection.** Hotspots are often nested and of diverse sizes. By detecting those hotspots, hierarchical phase changes are automatically captured.
- **Multi-grain adaptation.** With CU decoupling, the granularity of each CU's reconfiguration can be adjusted based on its reconfiguration overhead, resulting in better balance of reconfiguration benefit/cost.
- **Versatility and scalability.** By detecting hotspots of any sizes, this approach works efficiently for workloads with diverse runtime characteristics and CUs with different rent reconfiguration overheads.

4. Experimental methodology

In this research, we implement the proposed framework with the Dynamic Simplescalar simulator [16] and the Jikes Research Virtual Machine (RVM) [3]. We evaluate the proposed ACE management framework with the SPECjvm98 benchmark suite [28].

4.1. Simulation environment

The Dynamic Simplescalar (DSS) [16] 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 [8]. We augmented the power model to obtain our energy reduction results. We assume that the operating frequency and voltage of the target processor are

1GHz and 2V respectively. The baseline processor configuration is presented in Table 2.

In this paper, we use size-adaptable L1 data cache and L2 cache to demonstrate the framework's capability to manage CUs with varying reconfiguration intervals. Each cache has four different sizes (Table 2). Owing to the significant differences among their sizes and speeds, L1D and L2 caches' reconfiguration intervals differ considerably. In this paper, we assume the caches' reconfiguration intervals are 100K [9] and 1M instructions respectively. The modified power model also takes into account the power consumed for reconfiguring the hardware (i.e. power consumed for writing dirty cache lines into the lower level of memory hierarchy). We are implementing several more CUs, such as the issue window and the reorder buffer.

We also implement the basic block vector approach [24] in DSS, and compare it with our framework. To give advantages to the BBV approach, the BBV implementation allows unlimited number of uncompressed BBV signatures, each with 32 24-bit uncompressed buckets. The accumulator table is indexed by the lower 6 bits (excluding 2 least significant bits) of branch PCs. Furthermore, a phase's basic block vector information and tuning results are stored. Hence, a recurring phase can use its chosen configuration if available, or resume its tuning from the last tested configuration. However, this BBV implementation does not contain a next phase predictor.

4.2. Dynamic optimization system

Jikes RVM is a research Java virtual machine (JVM) developed in IBM T. J. Watson Center [3]. 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 the original Dynamic Simplescalar is not fully 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.

The Jikes RVM uses a low-overhead sampling method to detect program hotspots. 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.

Currently only the optimizing compiler with the highest 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 (200MB) to reduce garbage collection activities. By doing so, execution is dominated by the application rather than Jikes RVM.

As described in Section 3, hardware tunings and reconfigurations are performed after hotspots are detected and JIT optimized. The functionalities are implemented in the Jikes optimizing compiler, and Jikes' global data structure is also modified to store the necessary information for the hardware tunings and reconfigurations.

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. We run the SPECjvm98 benchmarks with the largest s100 data sets. Table 3 provides a summary of these SPECjvm98 benchmarks.

Table 3. 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.
mtrt	A dual-threaded program that ray traces an image file.

5. Evaluation results

Hotspot detection and the associated CU tunings on hotspot boundaries are the two major components of the proposed ACE management framework. This section evaluates the two components and compares the results with the BBV approach [24], one of the best existing approaches.

5.1. Runtime characteristics of hotspots

On average, the resulting hotspots execute at least 823 times. As shown in Table 1, the only disadvantage of the proposed DO-based framework over temporal approaches is the former's long initial hotspot identification latency, which may potentially override all other benefits. The proportion of the hotspot identification latency over whole program execution can be estimated by dividing *hot_threshold* by the average invocations per hotspot. Since a hotspot's average number of invocations far exceeds *hot_threshold*, the hotspot identification latency

Table 4. Runtime 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	99.03%	99.41%	99.96%	99.92%	99.83%	99.87%	99.87%
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%

Table 5. Runtime characteristics of the hotspot and BBV approaches

		comp	db	jack	javac	jess	mpeg	mtrt
Hotspot	**number of L1D hotspots**	64	58	81	108	68	64	73
	number of L2 hotspots	22	29	31	33	30	23	21
	total number of hotspots	85	87	112	141	98	87	94
	number of tuned hotspots	69	77	101	132	86	79	78
	% of tuned hotspots	81.18%	88.51%	90.18%	93.62%	87.76%	90.80%	82.98%
	per-hotspot IPC CoV	9.17%	9.97%	6.74%	9.33%	7.79%	5.37%	8.09%
	inter-hotspot IPC CoV	43.78%	42.99%	49.38%	46.47%	52.49%	49.05%	46.69%
BBV	**number of phases**	70	50	70	84	80	58	75
	number of tuned phases	35	16	14	22	24	13	17
	% of dynamic sampling intervals in tuned phases	81.40%	75.35%	71.44%	40.40%	56.97%	73.34%	93.37%
	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%

takes less than 3.65% of overall program execution. The data clearly shows that although hotspots take more time than BBV phases to be recognized, it does not pose a big burden to the hotspot approach.

5.2. Evaluation of the framework

To evaluate the proposed framework, we compare it with a system that uses the BBV phase detection technique [24] and the tuning algorithm prescribed in [9]. Since both techniques are the best among their respective alternatives, this combination should be the best technique that prior literature can contribute. The only difference between the tuning algorithms used in [9] and our framework is that our algorithm uses CU decoupling to reduce the tuning process. Since CU decoupling requires that phases are of variable sizes and nested, this technique is hard to be combined with the BBV technique.

The BBV technique used in this paper is not as aggressive as it can be, since it could use the phase prediction mechanisms presented in [20] and [24]. Accurate phase prediction tells what the next phase will be and when it will occur, and thus can improve the coverage of resource adaptations. In contrast, the hotspot approach does not need next phase prediction since it always identifies upcoming phases immediately.

Complying with the 1M reconfiguration interval of the L2 cache, we set the sampling interval size of the BBV scheme to 1M instructions. As for the DO-based framework, hotspots that configure the L1D cache (L1D hotspots) are between 50K and 500K instructions long, while L2 hotspots are at least 500K instructions long. With such multi-grain adaptation, each cache reaches better balance of reconfiguration benefit/overhead. As described in Section 3.4, hardware counters prevent too frequent reconfigurations of the caches.

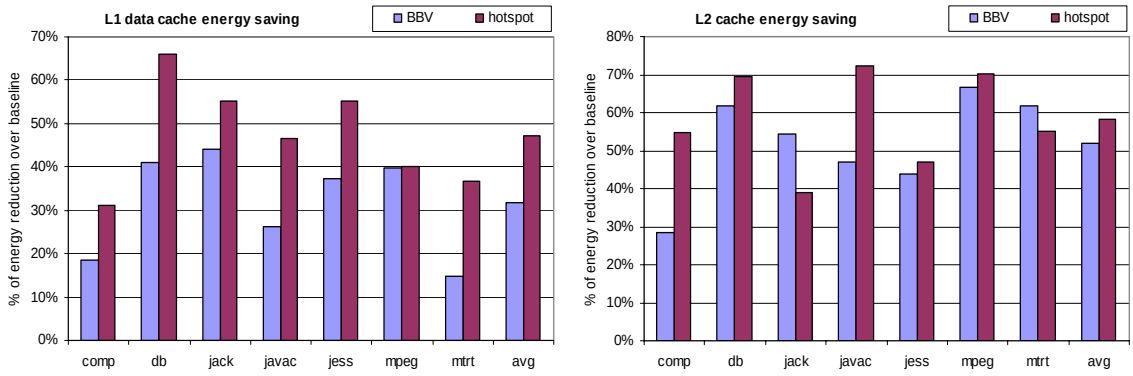
5.2.1. Runtime characteristics. Table 5 shows the total number of L1D/L2 hotspots (BBV phases) as well the number of hotspots (BBV phases) that complete the tuning process. Since hotspots are inherently invoked frequently and need to test 4 instead of 16 configurations required for BBV phases, on average 88% of hotspots finish tuning.

Table 5 also presents the percentages of dynamic sampling intervals in tuned phases. Although tuned BBV phases consist of only 29% of all BBV phases, they constitute on average 70% of the dynamic program execution. The rest of the program execution consists of either transitional phases (24%), or short-running phases (6%) that cannot finish their tunings.

Since tuned BBV phases dominate the overall program execution, the performance impact of unfinished tunings of short phases is minimal. However, as the number of CUs in the system grows, and more phases fail to finish their tuning because of the use of the straightforward tuning algorithm of testing all combinatorial configurations, the resulting adverse performance impact will increase dramatically. Note that it is possible to reduce the tuning process of both the BBV approach by ruling out some unpromising configurations found via offline profiling. Nevertheless, CU decoupling offers us a natural and elegant way to accomplish this in a DO-based system.

Table 6. Tunings, reconfigurations and coverage of hotspots and BBV phases

	hotspot						BBV		
	L1D tunings	L1D reconfigs	L1D coverage	L2 tunings	L2 reconfigs	L2 coverage	tunings	reconfigs	coverage
comp	247	2640	71.7%	85	835	73.8%	693	331	85.0%
db	218	3060	87.9%	130	1253	87.9%	419	832	80.1%
jack	338	30574	85.0%	109	4509	56.9%	443	464	77.0%
javac	506	46754	81.2%	58	3047	80.0%	711	1305	48.4%
jess	281	10321	92.7%	108	1333	84.3%	635	526	65.6%
mpeg	249	43753	91.0%	99	8514	95.7%	368	2018	76.5%
mtrt	355	48493	81.4%	21	396	82.6%	474	192	97.6%



(a) L1D cache energy reduction **(b) L2 cache energy reduction**

Figure 3. Cache energy consumption reduced by the resource adaptation schemes

Table 5 also the per-phase and inter-phase IPC coefficient-of-variations (CoVs) of both approaches. CoV equals the percentage of the standard deviation divided by the average. The *per-phase IPC CoVs* are IPC variations among different invocations of the same hotspot, which characterizes the homogeneity among different invocations of a hotspot. The *inter-phase IPC CoVs* are the variations of average IPCs of different hotspots, which quantify the heterogeneity among different hotspots. Larger inter-phase IPC CoV signifies larger differences between the characteristics of detected hotspots. More than 34% difference between hotspots' per-phase and inter-phase CoVs further confirms that hotspots are closely related with program behavior changes.

BBV phases have smaller per-phase CoVs than hotspots, which indicates that BBV phases are more stable than hotspots, i.e. there are fewer variations among different invocations of the same phase. On the other hand, BBV phases have smaller inter-phase CoVs than hotspots, which may be because BBV phases are insensitive to small-grain phase changes within sampling intervals; such small phase changes can be detected better by the hotspot approach.

Table 6 presents the number of tuning attempts made (*tunings*) and the number of times the most energy-efficient configuration is applied (*reconfigs)* for both the hotspot and the BBV algorithms. Due to CU decoupling, the hotspot algorithm conducts fewer tunings and is able to apply the most energy-efficient configurations more times than the BBV approach, which clearly demonstrates the advantage of tuning L1D and L2 caches separately on different hotspots. Note that using the hotspot tuning algorithm, the L1D cache is reconfigured more frequently than the L2 cache. This demonstrates the flexibility of the hotspot-based algorithm to finely tune the CUs with lower reconfiguration overheads for better performance.

In long-running applications, the impact of long tuning process will diminish. However, with the ability of multi-grain adaptation, our framework reaches better balance of adaptation benefit/overhead for each CU. This advantage does not diminish with longer execution.

Table 6 also gives the *coverage* (i.e. the portion of dynamically executed instructions under tuned/ reconfiguration configurations) results for L1D/L2 hotspots and BBV phases. In the BBV approach, hardware resources are adapted only at stable phases. Hence, the coverage results are the same as the stable phase distributions shown in Figure 1. As shown in Table 6, both L1D and L2 hotspots have good coverage across benchmarks. Good coverage and numerous reconfigurations indicate that CU decoupling does not sacrifice each CU's reconfiguration opportunity.

5.2.2. Energy reduction. Figure 3 shows the cache energy reduction achieved by the resource adaptation schemes. Both the BBV and the hotspot algorithms are examined and compared with the baseline configuration that uses the maximum sizes of the L1D and L2 caches. As for the L1D

cache, the hotspot-based algorithm is superior to the BBV-based algorithm on all workloads. The hotspot-based algorithm also performs better than the BBV algorithm for L2 on most benchmarks, except *jack* and *mtrt*. The hotspot approach performs especially well on *db* with 66% L1D cache energy reduction. In *db*, less than 10 procedures are responsible for more than 95% of data cache misses [25]. Consequently, the average cache sizes can be dramatically reduced for the hotspots that have very few data misses. On average, the hotspot approach achieves 47% energy reduction on the L1D cache and 58% energy reduction on the L2 cache over the baseline configuration. In comparison, the BBV approach only achieves cache energy reduction of 32% and 52% over the baseline configuration, respectively.

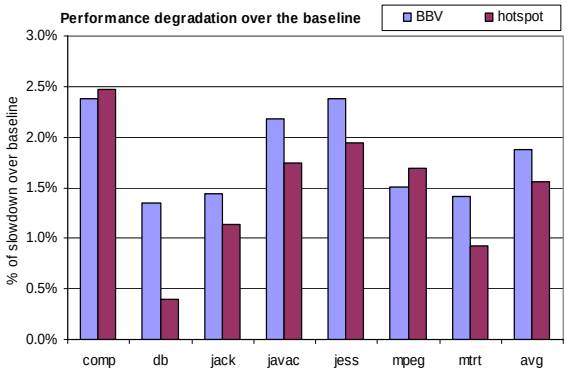


Figure 4. Performance impact of the resource adaptation schemes

5.2.3. Performance impact. The performance impact of using the resource adaptation schemes is illustrated in Figure 4. The performance degradation seen by the BBV technique ranges from 1.34% to 2.38%. For the hotspot technique the penalty ranges from 0.4% to 2.47%. On average the performance penalty for the hotspot technique is 1.56% and for the BBV scheme it is marginally worse at 1.87%. For the similar performance penalty, the DO-based ACE management framework achieves more energy reductions in L1D and L2 caches than the BBV method. These results indicate that the DO-based scheme is more efficiently than the BBV approach on managing multiple CUs with varying reconfiguration overheads/intervals, mainly due to CU decoupling that significantly reduces the tuning process, and multi-grain adaptation that achieves better balance of benefit/overhead for each configurable hardware unit.

6. Conclusion

In an adaptive computing environment, efficient management of the configurable resources is vital for maximizing the benefit of resource adaptation. The main 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 mechanisms, the proposed technique accurately detects program behavior at varying granularities, providing us the opportunity to significantly reduce the overheads 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, Longhorn, 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 adaptive computing environment having two configurable units (L1D cache and L2 cache). Our technique reduces L1D and L2 cache energy consumption by 47% and 58%, while a popular previously proposed technique only achieves reduction of 32% and 52% respectively.

In contrast to previously proposed techniques, the DO-based ACE management framework is inherently scalable to handle large number of configurable hardware resources. 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 adaptive computing environments.

Acknowledgments

We want to thank Xianglong Huang and Dr. Kathryn McKinley for providing us the Dynamic Simplescalar simulator. We would also like to thank Dr. Brad Calder and Dr. Michael Hind and the anonymous reviewers for their helpful suggestions on this paper. This research is

supported in part by NSF grants 0113105, 0429806, and Intel and IBM Corporations.

References

[1] D. Albonesi, "Dynamic IPC/Clock Rate Optimization", *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[2] D. Albonesi, "Selective Cache Ways: On-Demond Cache Resource Allocation", *Proceedings of the International Symposium on Microarchitecture*, November 1999.

[3] B. Appern, 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*, 1999.

[4] I. Bahar and S. Manne, "Power and Energy Reduction Via Pipeline Balancing", *Proceedings of the International Symposium on Computer Architecture*, June 2001.

[5] V.Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System", *Proceedings of Programming Language Design and Implementation*, June 2000.

[6] 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*, Dec. 2000.

[7] L. Baraz, T. Devor, O. Etzion, S. Gondenberg, A. Skaletsky, Y. Wang, 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*, Dec. 2003.

[8] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-level Power Analysis and Optimizations", *Proceedings of the International Symposium on Computer Architecture*, June 2000.

[9] A. S. Dhodapkar and J. E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis", *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[10] A. Dhodapkar and J. Smith, "Comparing Program Phase Detection Techniques", *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.

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

[12] D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic", *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[13] 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*, 1998

[14] M. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction", *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[15] 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*, Dec. 2000.

[16] X Huang, J. Moss, K. McKinley, "Dynamic SimpleScalar: simulating Java Virtual Machines". *The 1st Workshop on Managed Run Time Environment Workloads*, March 2003.

[17] A. Iyer, D. Marculescu, "Microarchitecture-level Power Management," *IEEE Transactions on VLSI Systems*, Vol.10, No.3, June 2002.

[18] J. Kin, M. Gupta, and W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure", *Proc. of the 30th International Symposium on Microarchitecture*, Dec. 1997.

[19] 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*, March, 2003.

[20] J. Lau, S. Schoenmackers, B. Calder, "Transition Phase Classification and Prediction", *Proceedings of the 11th International Symposium on High Performance Computer architecture*, Feb. 2005.

[21] 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*, Vol. 50, No. 6, pp. 567-589, June 2001.

[22] D. Ponomarev, G. Kucuk, K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources", *Proceedings of the 34th International Symposium on Microarchitecture*, Dec. 2001.

[23] X. Shen, Y. Zhong, C. Ding, "Locality Phase Prediction", *Proceedings of the 11th International Conference on Architectural Support for Programming, Languages, and Operating Systems*, October 2004.

[24] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction", *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[25] Y. Shuf, M. Serrano, M. Gupta, and J. Singh, "Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations", *ACM SIGMETRICS*, 2001.

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

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

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

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

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