Impact of Virtual Execution Environments on Processor Energy Consumption and Hardware Adaptation

Shiwen Hu
Networking and Computing Systems Group
Freescale Semiconductor, Inc.
7700 W. Parmer Lane, Austin, TX 78729
shiwen.hu@freescale.com

Lizy K. John
Laboratory for Computer Architecture
The University of Texas at Austin
1 University Station C0803, Austin, TX 78712
ljohn@ece.utexas.edu

Abstract

During recent years, microprocessor energy consumption has been surging and efforts to reduce power and energy have received a lot of attention. At the same time, virtual execution environments (VEEs), such as Java virtual machines, have grown in popularity. Hence, it is important to evaluate the impact of virtual execution environments on microprocessor energy consumption. This paper characterizes the energy and power impact of two important components of VEEs, Just-in-time (JIT) optimization and garbage collection. We find that by reducing instruction counts, JIT optimization significantly reduces energy consumption, while garbage collection incurs runtime overhead that consumes more energy. Importantly, both JIT optimization and garbage collection decrease the average power dissipated by a program. Detailed analysis reveals that both JIT optimizer and JIT optimized code dissipate less power than un-optimized code. On the other hand, being memory bound and with low ILP, the garbage collector dissipates less power than the application code, but rarely affects the average power of the latter.

Adaptive microarchitectures are another recent trend for energy reduction where microarchitectural resources can be dynamically tuned to match program runtime requirements. This research reveals that both JIT optimization and garbage collection alter a program's behavior and runtime requirements, which considerably affects the adaptation of configurable hardware units, and influences the overall energy consumption. This work also demonstrates that the adaptation preferences of the two VEE services differ substantially from those of the application code. Both VEE services prefer a simple core for high energy reduction. On the other hand, the JIT optimizer usually requires larger data caches, while the garbage collector rarely benefits from large data caches. The insights gained in this paper point to novel techniques that can further reduce microprocessor energy consumption.

Categories and Subject Descriptors B.8.2 [**Hardware**]: Performance and Reliability - *Performance Analysis and Design Aids*.

General Terms Experimentation, Management, Measurement.

Keywords Energy Efficiency, Hardware Adaptation, Power Dissipation,.

VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.

1. Introduction

The surging power dissipation and energy consumption in microprocessors raise concerns in both hardware and software communities. Power dissipation affects circuit reliability and package costs, while energy consumption determines battery life of embedded systems. Previous work shows that a better understanding of program runtime behavior can help alleviate the problem [10][18][22][23].

During recent years, Virtual execution environments (VEEs), such as Java virtual machines [26] and Microsoft .NET common language runtime [27], have grown in popularity. Compared with platform-specific binaries, these systems provide several software engineering advantages, including portability, automatic memory and thread management, and dynamic class loading. All those features are implemented as runtime VEE services that support the dynamic execution of applications. Among various VEE services, Just-in-Time (JIT) optimization and garbage collection (GC) are two important ones. A VEE usually spends more time on JIT optimization and garbage collection than on other services. More importantly, these two VEE services can alter the runtime behavior, and thus energy consumption, of applications. Consequently, the runtime characteristics of a program's dynamic execution differ significantly from its statically compiled counterpart, necessitating a study of virtual execution environments' impact on energy consumption and power dissipation.

Meanwhile, adaptive microarchitectures are a recent trend for energy reduction where microarchitectural resources can be dynamically tuned to match program runtime requirements [1] [9][16]. The execution of an application usually passes through phases with varying runtime characteristics and hardware requirements [19]. Hardware configurations can thus be adjusted dynamically to reduce energy consumption with minimal performance impact. Since both JIT optimization and garbage collection alter programs runtime requirements, it is important to understand how they interact with hardware adaptation to achieve overall energy reduction.

This research first characterizes the energy and power impact of JIT optimization and garbage collection on SPECjvm 98 benchmarks [25]. Then, we evaluate the two VEE services' influence on the adaptation of five hardware units: issue queue, reorder buffer, level-one instruction and data caches, and level-two unified cache. The research is conducted with the hardware adaptation framework presented in [12], which executes the Jikes research virtual machine (RVM) [3] using Dynamic Simplescalar [13].

To our best knowledge, this is one of the first efforts to examine the impact of VEEs on microprocessor energy consumption and

hardware adaptation. We find that by reducing instruction counts, JIT optimization significantly reduces energy consumption, while garbage collection incurs runtime overhead that consumes more energy. Interestingly, both JIT optimization and garbage collection decrease the average power dissipated by a program. Detailed analysis reveals that both JIT optimizer and JIT optimized code dissipate less power than un-optimized code. Since both JIT optimizer and JIT optimized code stress the level-one data cache, during a JIT optimized system' execution, hardware units in the execution engine are more likely to be idle and thus dissipate less power than an un-optimized system. On the other hand, being memory bound and with low ILP, the garbage collector dissipates less power than the application code, but rarely affects the average power of the latter.

This research also reveals that VEE services interfere with hardware adaptation by altering program behavior and runtime requirements. In adaptive microarchitectures, such changes of runtime requirements can considerably affect the adaptation of configurable hardware units, and eventually influence the overall energy consumption. We also study the adaptation preferences of configurable units on the JIT optimizer and the garbage collector, demonstrating the adaptation preferences of the two VEE services differ substantially from those of the application code. For instance, both JIT compiler and garbage collector prefer a simple core for energy reduction. On the other hand, the JIT optimizer usually requires larger data caches to sustain its performance, while the garbage collector can use smaller caches with minimal performance loss. The insights gained in this paper point to novel techniques that can further reduce microprocessor energy consumption.

This paper is organized as follows. Section 2 describes prior research on compiler assisted energy reduction. Section 3 presents the simulation methodology. The energy impact of JIT optimization and garbage collection is examined in Section 4. And Section 5 studies the interference of dynamic execution on hardware adaptation. Section 6 concludes the paper.

2. Related work

In this paper we concentrate on the influence of JIT optimization and garbage collection on microprocessor power dissipation, energy consumption, and hardware adaptation. In comparison, most previous efforts focus on the impact of static compiler optimizations on power and energy.

With the importance of energy reduction, previous work examines the impact of static compiler optimizations on energy consumption and power dissipation. Kandemir et al. [15] investigate the influence of several compiler optimizations, such as linear loop transformations, blocking, loop unrolling and loop fusion, on energy consumption. They find that those optimizations usually reduce the memory energy consumption at the cost of rising core energy consumption. Valluri et al. [21] note that compiler optimizations for locality and instruction counts usually optimize energy consumption, while optimizations improving ILP increase average power dissipation. In their study of Pentium 4 power consumption, Seng et al. [17] note that compiler optimizations affect energy consumption mainly by reducing their execution time. Chakrapani et al. [8] examine the interaction between compiler optimizations and processor components, and draw similar observations as Valluri [21].

Power-aware compiler optimizations improve energy consumption and power dissipation. Simunic et al. [18] perform several compilation optimizations, including loop merging, unrolling, software pipelining, loop invariant extraction to optimize the performance and energy consumption of a MPEGAUDIO video decoder in an embedded system, with the assistance of profiling information. Software pipelining is used by Yang et al. [24] in a power-aware instruction scheduling scheme to reduce power variation throughout program execution.

Compiler optimizations can also assist hardware energy reduction proposals. Valluri et al. [22] use the compiler to assist instruction scheduling, thus achieving low power, low-complexity instruction issue. Compared with run-time scheduling, compile-time scheduling features fast and simple hardware, but at the expense of conservative schedules [11]. The paper implements a compile-time analyzer to identify basic blocks free of memory misses, false dependences, or unresolved alias edges, and schedules them statically. This issue queue design includes two types of queues, a FIFO queue for statically scheduled instructions and a fully associative queue for instructions requiring dynamically scheduling. The scheme consumes less energy than the conventional issue queue since the fully associative queue is much smaller, and has a smaller issue width than the conventional one.

A compiler-directed strategy [11] identifies non-critical program code regions, and dynamic voltage and frequency can be scaled down on those regions for energy reduction. The compiler can also detect program regions that do not use certain hardware components, e.g., hard disk, for certain long enough periods so that the hardware components can be in hibernation for energy reduction [10].

Recent research shows that virtual execution environments can help reduce microprocessor energy consumption. Hu et al. [12] propose a hardware adaptation framework based on a virtual execution environment for efficient management of multiple configurable hardware units for energy reduction. Wu et al. [23] use a lightweight dynamic compilation framework for dynamic voltage and frequency scaling (DVFS). Both schemes utilize the runtime profiling and compilation capabilities of virtual execution environments to identify energy reduction opportunities, and do not investigate the impact of virtual execution environments on microprocessor energy consumption.

Prior works demonstrate that static software optimizations affect energy consumption, which motivates us to study the impact of runtime optimizations and GC on energy consumption. On the other hand, unlike offline optimizations, VEE services are executed on-the-fly. They affect program runtime behavior in two ways: 1) they change a program's runtime behavior; 2) their own runtime execution contributes to the overall energy consumption. Hence, besides examining the overall program behavior, we need to investigate those two impacts individually. This is one major difference between our work and prior works.

3. Experimental methodology

The research is conducted with the hardware adaptation framework introduced in [12]. It is implemented with the Dynamic Simplescalar simulator [13] and the Jikes RVM [3]. We obtain the results using the SPECjvm98 benchmark suite [25].

Table 2. Characteristics of SPECjvm 98 benchmarks executed by Jikes RVM with JIT1 and no GC.

	comp	jess	db	javac	mpeg	mtrt	Jack
Instruction count (billions)	9.83	5.73	8.78	8.90	10.93	4.17	8.18
Cycle count (billions)	5.30	4.93	14.15	7.66	6.98	2.76	5.88
IPC	1.85	1.16	0.62	1.16	1.56	1.51	1.39
Energy consumption (J)	141.70	98.08	219.96	153.39	174.55	63.46	127.33
Number of methods	140	254	145	750	260	287	491

Table 3. The configurable hardware units and their adaptation parameters.

Configurable units	Configurations (units)	Reconfiguration interval (instructions)	Hotspot size (instructions)
Issue queue/Reorder buffer	128/96/64/32 (entries)	1000	1K – 100K
L1I/L1D caches	64K/48K/32K/16K (bytes)	100K	100K – 1M
L2 cache	1M/768K/512K/256K (bytes)	1M	> 1M

3.1 Simulation environment

The Dynamic Simplescalar (DSS) simulator [13] used in our work adds a series of major extensions to Simplescalar/PowerPC 3.0, and permits simulation of a full Java run-time environment on a simulated hardware platform. The original Simplescalar 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 newest version (1.01) of DSS incorporates a power model that is based on Wattch [7]. Leakage power is one of the dominate factors in power consumption. We do not measure leakage power since compared to dynamic power, leakage power is relative constant in that it is less likely to be affected by programs.

Wattch has three different options for clock gating to disable unused resources in the processor. We chose power and energy results corresponding to the third scheme since it is the most realistic of all schemes. In this scheme, power is scaled linearly with port or unit usage, but unused units dissipate 10% of their maximum power. The percentage was chosen as it represents a typical turnoff figure for industrial clock-gated circuits. The operating frequency and voltage of the target processor are 1GHz and 2V respectively. The baseline processor configuration is presented in Table 1.

Table 1. Baseline configuration of the simulated system.

CPU (1GHz with 2V)	
Instruction window	64-IFQ, 128-IQ, 128-ROB, 64-LSQ
Functional units	4 int. ALU, 2 int. Mult/Div, 4 FP ALU, 2 FP Mult/Div
Branch predictor	2K-entry, combined, 3-cycle misprediction penalty
Issue/Commit width	4 instructions per cycle
Memory Hierarchy	
L1I cache	64KB, 64B blocks, 4-way, LRU, 1 cycle hit latency
L1D cache	64KB, 64B blocks, 4-way, LRU, 1 cycle hit latency
L2 unified cache	1MB, 128B blocks, 4-way, LRU, 10 cycles hit latency, 100 cycles miss penalty
DTLB/ITLB	128 entries, fully set associative

3.2 Virtual execution environment

Jikes RVM is a research Java virtual machine developed at 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 java virtual machine itself. The 2.0.2 version of Jikes RVM is used since Dynamic Simplescalar is not compatible with the latest version of Jikes RVM.

Jikes RVM employs a compile-only strategy (i.e., no interpreter mode). It includes a baseline and an optimizing compiler. Jikes RVM's baseline compiler is a fast non-optimizing compiler that converts Java bytecodes to machine code. The optimizing compiler of Jikes RVM has three levels of optimizations (JIT0, JIT1, and JIT2), each one consisting of its own group of optimizations as well as the optimizations that belong to lower levels. The lower two levels (JIT0 and JIT1) perform optimizations that are fast and high-payoff. JIT0 performs inlining and register allocation. JIT1 contains optimizations, such as common sub-expression elimination, copy and constant propagation, and dead-code elimination. JIT2 contains more expensive optimizations, such as those based on static single assignment (SSA) form.

The Jikes RVM uses a low-overhead sampling method to detect program hot spots. Approximately every 10 milliseconds, Jikes RVM increments a counter associated with the currently active procedure. For all methods that have been sampled, Jikes RVM uses a cost/benefit model to determine whether it is profitable to recompile the method, and if so, what level of optimization to use. In this research, we do not use Jikes RVM's sampling based adaptive optimization system. All methods are compiled only once, either by the baseline or by the optimizing compiler, and thereafter executed with no further compilation.

The garbage collector implemented in the toolchain we use is Appel's generational collector [2] from the GCTk toolkit [6]. The Appel collector uses a size-adaptable nursery, and initially it is the whole heap. Each collection reduces the nursery size by the survivors. A full heap triggers a full heap collection. To test various garbage collection activities, we use three heap sizes, 25M, 42.5M, and 60M in our simulations, and compare the results with those obtained with no GC activities (using a 200M heap). For several benchmarks, 25M is the minimal size in which the Appel collector works [13].

3.3 Benchmarks

The industry standard SPECjvm98 benchmarks [25] 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 java virtual machine. We run the SPECjvm98 benchmarks with the largest s100 data sets. Table 2 provides a summary of the runtime characteristics of these SPECjvm98 benchmarks.

Table 4. Runtime characteristics of JIT optimized system as fractions of the corresponding results of non-optimized system.

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

Table 5. Performance, energy and power of JIT optimized system as fractions of the corresponding results of non-optimized system.

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

3.4 Hardware adaptation framework

In this research, configurable hardware units (*CUs*) are managed by the hardware adaptation framework [12] that is implemented with the Dynamic Simplescalar simulator [13] and the Jikes RVM [3]. It utilizes the virtual execution environment to detect program hotspots and adapts hardware units at program hotspot boundaries. By adapting CUs with varying reconfiguration costs at different hotspot boundaries, the framework is effective on managing multiple configurable hardware units.

Five CUs are implemented in this work: issue queue (IQ), reorder buffer (ROB), level-one data (L1D) and instruction (L1I) caches and level-two (L2) cache (Table 3). Each configuration unit has four different sizes that can be adjusted on the runtime. The DSS power model is modified to reflect the size reduction of the CUs and the power consumed for reconfiguring the hardware. Owing to the significant differences among their sizes and speeds, the reconfiguration intervals of the CUs differ considerably [9][16]. Table 3 also lists the size ranges of the hotspots at whose boundaries the CUs are adapted. We refer the readers to [12] for the details about the framework.

4. Energy and power impact of VEE services

The runtime characteristics of a program's dynamic execution (i.e., execution in a virtual execution environment) usually vary significantly from its statically compiled counterpart, owing to the following reasons. First, optimizations used in offline and JIT compilers may differ substantially. Although a JIT optimizer usually contains some commonly used offline compiler optimizations, it rarely adopts other offline optimizations, such as loop transformations, due to their high costs. Meanwhile, novel adaptive optimizations utilizing feedback information are designed specifically for VEEs [4]. Hence, dynamically optimized programs usually exhibit characteristics that differ to their statically optimized counterparts. Second, as an important part of many VEEs, the garbage collector may also change a program's runtime behavior by compacting and rearranging heap objects. Finally, a program's dynamic execution comprises the application and numerous assisting VEE services, such as JIT optimization and garbage collection. Being integral parts of a program's dynamic execution, JIT optimization and garbage collection have distinct characteristics to applications, and thus affect the overall runtime characteristics.

With the popularity of virtual execution environments and the urgency to reduce surging microprocessor energy consumption, it is important to understand the impact of VEEs on energy consumption and power dissipation. This section characterizes the energy and power impact of two important VEE services, JIT optimization and garbage collection.

4.1 Impact of JIT optimization

In this section, JIT1 optimizations are compared with the baseline compiler (BASE). We do not show results for JIT optimization level zero since they are similar to JIT1 results. Unfortunately, JIT2 optimizations cannot be tested in this work since Dynamic Simplescalar fails when high level optimizations generate instructions or call system calls that are not implemented in Dynamic Simplescalar. If DSS does not fail, we expect that using high level optimizations will yield incremental energy reduction over the first level. The main source of energy reduction has been performance improvement. In a compiler with multiple levels of optimizations, lower level optimizations usually achieve the largest performance improvements over no optimization, while high level optimizations yield diminishing improvements. Hence, the energy reduction achieved by high level optimization should also diminish. The experiment uses a 200M heap to minimize garbage collection activities in SPECjvm 98 benchmarks.

4.1.1 Impact on instruction window and caches

To examine the impact of JIT optimization on program execution, the characteristics for the instruction window and the cache hierarchy are obtained. Those metrics include issue queue and reorder buffer occupancies, and L1D/L1I/L2 cache miss rates. The results presented in Table 4 are the fractions of the JIT1 results over the corresponding BASE ones.

JIT optimization increases IQ and ROB occupancies, as well as L1D and L1I miss rates. During the JIT optimizations, large data structures are used to store intermediate representations of methods and other information [3]. Since the L1D cache can rarely hold all the data structures, the JIT optimizer's traversing of the data structures incurs many L1D misses and increases overall L1D miss rates. As the pipeline is slowed down to wait for the missed data accesses, many dependent instructions stall in the issue queue and the reorder buffer and result in higher IQ and ROB occupancies. JIT optimization's impact on the L1I cache is two-fold. First, JIT optimizations generate more compact code and reduce L1I capacity misses. On the other hand, after the optimized code is generated, the instruction cache must be flushed

Table 6. Comparison of JIT optimizer and application's energy consumption and power dissipation.

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

Table 7. Number of garbage collections and changes in cache misses due to garbage collection.

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

to bring in the newly generated code, incurring more cold misses. Determined by those two conflicting factors, L1I miss rates increase slightly.

JIT optimization slightly reduces a program's L2 cache miss rate. JIT optimization greatly reduces the size of a program' instruction working set, improving its L2 cache miss rate. In contrast to the L1D cache, the L2 cache is much larger to hold most of the intermediate data structures, and thus the JIT optimizer's accesses to those data structures incurs few L2 misses.

Note that we intentionally choose characteristics that are fractions by themselves. For instance, the cache miss rate equals total cache misses divided by total cache accesses. As shown in Table 5, the biggest impact of JIT optimization is the reduction of instruction and cycle counts, which affects other non-fractional characteristics as well. Comparing those characteristics allows us to identify other program behavior changes hidden by the reduction of instruction and cycle counts.

4.1.2 Impact on performance, energy, and power

For SPECjvm 98 benchmarks, Table 5 presents their instruction counts, cycle counts, energy consumption, and average power using JIT1 optimizations as fractions of the corresponding BASE results. JIT optimization significantly reduces a program's instruction count, execution time, and energy consumption. On average, JIT1 optimizations reduce a program's instruction count, cycle count, and energy consumption to 32%, 36%, and 33% of those without JIT optimization.

JIT1 optimizations are more effective on reducing instruction counts than execution time. Compiler optimizations can be broadly classified into two groups: those that reduce instruction counts, and those that improve ILP without reducing instruction counts, such as instruction scheduling. Most JIT1 optimizations, such as common sub-expression elimination, copy and constant propagation, and dead code elimination, belong to the first group. Hence, for all programs studied, the reduction of execution time is not as much as the reduction of instruction counts. The increases of L1D miss rates (Table 4) also contribute to the growing disparities between instruction and cycle counts.

A program's energy consumption is related to both the total amount of work done by the program (instruction count) and the time taken to finish the work (cycle count). Apparently there is more idling when the optimized code is executed. Consequently, SPECjvm 98 benchmarks' average power dissipation decreases due to the JIT optimizations. On average, a program's JIT-optimized execution dissipates 93% of the power as its non-JIT-optimized execution.

Among the seven benchmarks, *compress* benefits the most from JIT optimization. The benchmark *compress* spends most of its execution on two loops. The first loop compresses input files, which are then decompressed in the second loop. Hence, *compress* has a few hot methods within the loops that dominate its execution. Optimizing those hot methods can significantly improve the performance and energy consumption while incurring minimal overhead. In contrast, *javac* and *jack* achieve the least performance improvement and energy reduction among all the workloads. The benchmark *javac* is a Java compiler that translates Java source code into binary code, while *jack* is a Java parser generator with lexical analysis. Compared to other benchmarks, they contain more methods that require more time to optimize (Table 2).

4.1.3 JIT optimizer versus application

Both JIT optimized code and the JIT optimizer normally possess distinct characteristics to non-optimized system, and thus affect a program's dynamic execution. This subsection evaluates the performance and energy impact of each of the above two factors. To do so, the Jikes RVM is instrumented to identify code regions belonging to the JIT optimizer.

Table 6 gives the portions of overall execution time (% of JIT cycle count), instruction count (% of JIT instr. count), and energy consumption (% of JIT energy) corresponding to the JIT optimizer. The differences between 100% and the JIT percentages are the portions for applications. On average, the JIT optimizer accounts for 9.8% of total instructions, 11% of total cycles, and 10.7% of total energy consumed in a program's JIT-optimized execution.

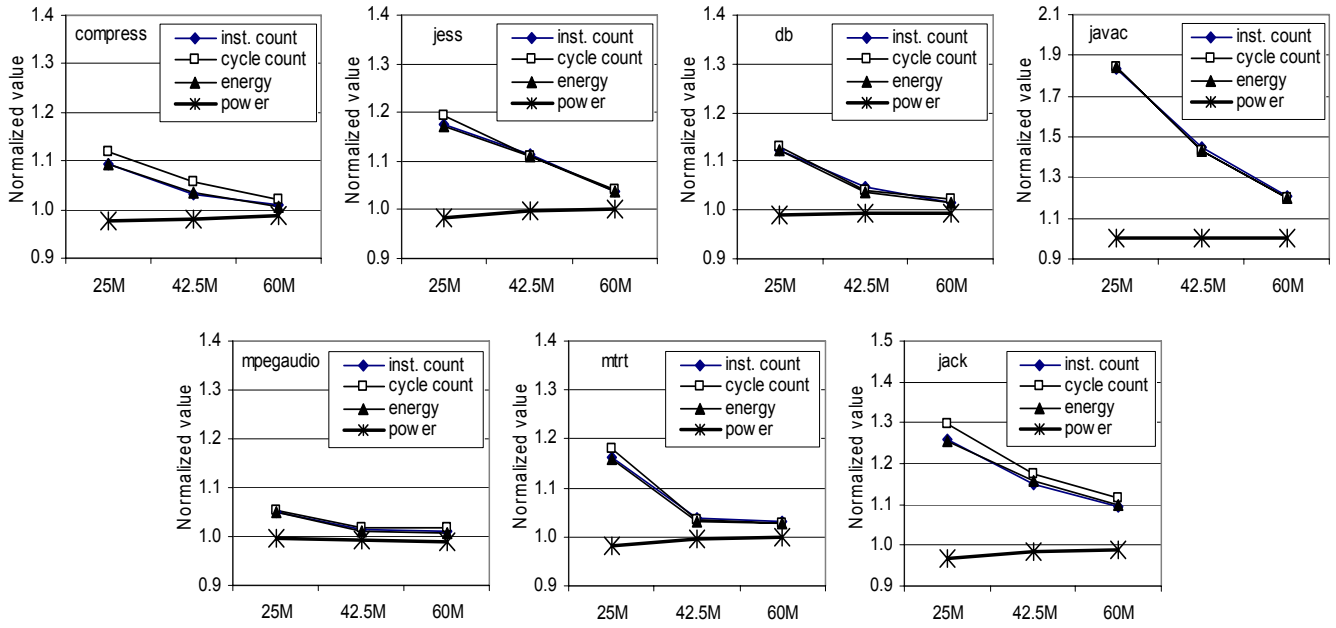

Figure 1. Changes in performance, energy, and power due to garbage collection.

The last two rows of Table 6 represent the normalized power results for the JIT compiler and the applications respectively. Those power results are normalized to the ones corresponding to the non-optimized execution of the programs. Table 6 shows that both the JIT optimizer and the JIT-optimized applications dissipate less power than the non-optimized applications, and contribute to the overall power reduction of JIT-optimized systems (Table 5).

4.2 Impact of garbage collection

In this subsection, the performance, energy, and power results of programs executed by Jikes RVM with different heap sizes are obtained. Those results are then compared with the ones obtained with a 200M heap that is large enough to have minimal garbage collection in all benchmarks. The energy impact of the garbage collector is also examined and compared with that of the mutator (i.e., a program's execution other than the garbage collections). The experiment uses JIT1 optimizations.

4.2.1 Impact on caches

Table 7 lists the number of garbage collections (Number of GCs) conducted throughout program execution with heap sizes varying from 25MB to 60MB. A program's number of GCs is highly dependent on the program's memory requirements and lifetimes of the objects allocated. Both *compress* and *mpegaudio* have fewer garbage collections than other benchmarks, confirming that those two benchmarks allocate very little heap space [20]. In contrast, *javac* performs many more garbage collections than the other benchmarks.

Since the task of garbage collection is to compact and reallocate heap objects, its major performance impact is on the memory subsystem. For each benchmark, Table 7 provides its L1D/L1I/L2 misses using the 25M/42.5M/60M heaps as fractions of the ones obtained using the 200M heap. The difference between a value and 100% represents the percentage of misses increased due to garbage collection. For comparison, the average instruction counts increase by 21% (25M heap), 12% (42.5M) and 6% (60M) respectively over no GC, and each benchmark's normalized instruction counts are drawn in Figure 1.

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

4.2.2 Impact on performance, energy, and power

For each benchmark, Figure 1 shows the normalized instruction count, execution time, and energy consumption by dividing the results for the small heaps by the corresponding ones with the 200M heap.

The energy consumed is directly proportional to the number of instructions. *javac*'s performance and energy consumption are significantly affected by garbage collection. With a 25M heap, *javac*'s instruction count, cycle count, and energy all increase by 65% over no GC, which is in accordance with the fact that *javac* has more garbage collections than other benchmarks (Table 7). In contrast, with 42.5M and 60M heaps, *mpegaudio*'s performance and energy consumption almost equal the corresponding ones with no garbage collection, reflecting the fact that *mpegaudio* allocates very little heap space [20]. All the other benchmarks'

Table 8. Comparison of garbage collector and mutator's energy consumption and power dissipation.

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

performance degradation and energy increases are between those of *javac* and *mpegaudio*.

Garbage collection slightly reduces a program's average power dissipation. As Table 7 shows, many benchmarks' L2 misses increase very fast as heap size decreases. Such cache misses can often stall the pipeline, and considerably affect the execution time. On the other hand, since idle units consume only 10% of their peak power, energy is less sensitive to pipeline stalls than execution time, resulting in the lower power dissipation during garbage collection. Among the evaluated workloads, *compress* and *javac*'s average power results drop most, varying between 2% and 4%. According to Table 7, both benchmarks' L2 cache performance and *jack*'s L1 cache performance are impaired by garbage collection.

Note that the energy results in Figure 1 account for only the microprocessor, not the main memory. In our experiment, different heap sizes are used. Heaps reside in the virtual memory, which is usually larger than the physical memory. Several factors determine an application's memory energy consumption. First, accessing a small physical memory consumes less power than accessing a large one. One study on SDRAM energy consumption shows that as the memory size doubles, its energy consumption increases by up to 10% [14]. On the other hand, if the application's data working set is larger than the physical memory, page swaps may occur frequently between the physical memory and storage devices, hurting both performance and energy consumption. Third, the heap size can also affect an application's memory energy consumption. A small heap has to be collected more frequently, and thus has more memory accesses than a larger heap. As shown in Table 7, reducing heap size dramatically increases the amount of L2 misses, i.e., memory accesses. The overall memory energy consumption is determined by those factors as well as the memory access patterns. Unfortunately, a detailed study on memory energy consumption is out of the scope of this research.

4.2.3 Garbage collection versus mutator

Similar to JIT optimization, garbage collection's impact on a program's dynamic execution represents both GC-incurred mutator behavior changes and the increase of garbage collection activities. This subsection evaluates the performance and energy impact of each of the above two factors.

Table 8 presents the portions of overall execution time (% of GC cycle count), instruction count (% of GC inst. count), and energy consumption (% of GC energy) spent on the garbage collections. The differences between 100% and the GC percentages are the portions taken by the mutator. The results confirm that with smaller heaps, the garbage collector is responsible for more of the overall work done and energy consumed by a program.

Table 8 also includes the average power results for the garbage collector and the mutator respectively, which are normalized to the ones obtained with the 200M heap. A normalized power value smaller than 100% means that the average power drops due to garbage collection. On average, the garbage collector's power dissipation is between 87% and 92%, varying by heap sizes and programs, of the mutator's. This is mainly because the poor data locality of the garbage collector impairs its execution time more than its energy consumption. On the other hand, garbage collection only slightly changes the mutator's average power, with variations below 2% in all benchmarks and heap sizes. Garbage collection improves a program's data locality [5], which may speed up the execution and increase the power of the mutator. Table 8 shows that for most programs, except for *jess* and *javac*, the power increase due to improved data locality is minimal. Figure 1 indicates that with garbage collection, a program's average power dissipation usually drops. The results in Table 8 denote that the garbage collector, instead of the mutator, is responsible for the drop of overall power dissipation.

4.3 Key insights

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

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

Although obtained using Jikes RVM, the observation should be applicable to other virtual execution environments that employ JIT optimization and garbage collection. For all those VEEs, JIT optimization improves performance while garbage collection incurs runtime overhead. Since energy is proportional to instruction and cycle counts, JIT optimization should always reduce energy, while garbage collection consumes more energy. Similarly, most observations on the power impact of the VEE services should also be applicable to other VEEs. One exception is the overall power reduced by JIT compiler. A JIT optimizer,

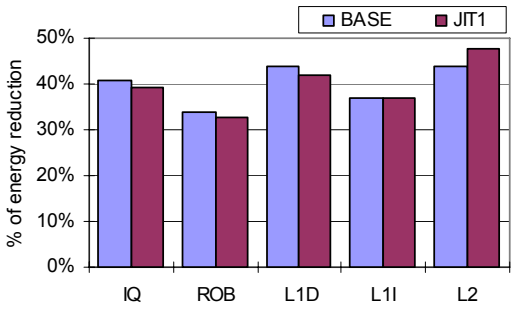

Figure 2. Impact of JIT optimization on configurable unit energy reduction

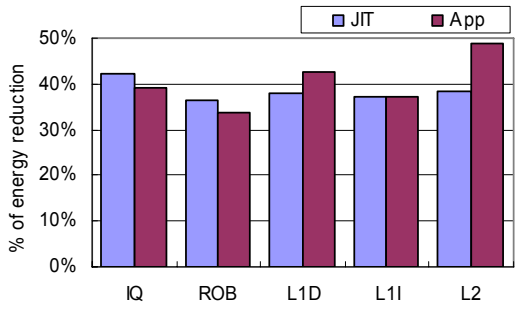

Figure 3. Comparison of JIT optimizer and application on configurable unit energy reduction.

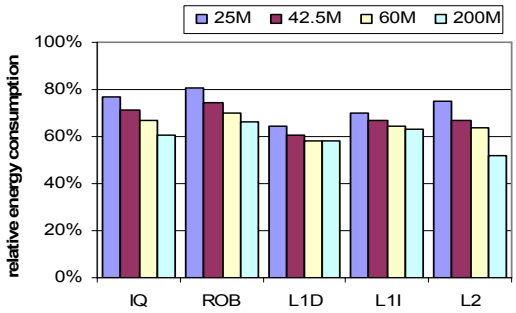

Figure 4. Impact of garbage collection and hardware adaptation on configurable unit energy consumption.

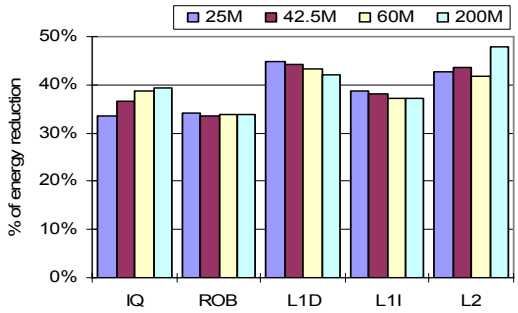

Figure 5. Impact of garbage collection on configurable unit energy reduction.

containing different JIT optimizations, may be more effective on reducing cycle counts than instruction counts, and it may increase the average power.

5. Interference of VEE Services on hardware adaptation

Adaptive microarchitectures are another recent trend for energy reduction where microarchitectural resources can be dynamically tuned to match a program's runtime requirement [1][9][16]. Since both JIT optimizations and garbage collection can alter the runtime behavior of dynamically executed programs, such changes of program behavior may interfere with the tuning of CUs and affect their energy reduction. Hence, it is important to understand how JIT optimization and garbage collection interact with configurable hardware units to achieve overall energy reduction.

This section investigates the interference of JIT optimization and garbage collection on the adaptation of the five CUs listed in Table 3. The hardware adaptation framework presented in [12] manages those CUs. Each configuration unit has a performance degradation budget of 1%.

5.1 Interference of JIT optimization

5.1.1 Impact on overall energy reduction

To measure the impact of JIT optimization on hardware adaptation, we simulate each configurable hardware unit's energy consumption results in both BASE and JIT1 levels. To avoid the interference of one CU to another's adaptation, each time only one CU is adapted, and all other units keep their largest sizes throughout program execution. Averaged over all the benchmarks, the results in Figure 2 represent the percentages of energy reduced by hardware adaptation in both JIT optimization levels. The differences between those two sets of results represent the impact of JIT optimization on the adaptation of the CUs.

According to Figure 2, JIT optimization impairs the adaptation of issue queue, reorder buffer, and L1D cache. As Table 4 shows, L1D miss rates increase as JIT optimizations are applied. In this case, the L1D cache may choose a larger size to prevent high performance loss and incurs less energy reduction in JIT1 than in BASE. One interesting observation is that although higher IQ and ROB occupancies do not mean higher energy consumption in the fixed size IQ and ROB, they prevent the configurable IQ and ROB to use more aggressive configurations for higher energy saving. Similarly, since JIT optimization improves L2 cache's performance, the cache can use smaller sizes for higher energy reduction. Finally, the L1I cache performance is barely affected by JIT optimization. Consequently, L1I cache's adaptation efficiencies (i.e., percentage of energy reduction due to hardware adaptation) barely change as the JIT optimizations are applied.

5.1.2 JIT optimizer versus application

Figure 3 further investigates the adaptation efficiencies on the JIT optimizer (JIT) and the applications (App). For each CU, the energy results are obtained for the JIT optimizer and the applications respectively, and are then normalized to the corresponding ones using fixed size units to obtain the portions of energy reduced by adapting the CUs on the JIT optimizer and the applications. The results in Figure 3 are the arithmetic means of the seven benchmarks.

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

5.2 Interference of garbage collection

5.2.1 Impact on overall energy reduction

Both garbage collection and hardware adaptation affect a CU's energy consumption. To examine the two factors, we first obtain each fixed size unit's energy consumption using the 200M heap, which is used as the baseline. Then hardware adaptation is enabled, and each unit's energy consumption is obtained using different heap sizes. To prevent the interference between CUs, each experiment tests only one CU, and the other units keep constant with their largest configurations. The results are normalized to the baseline ones to show the relative change caused by hardware adaptation and garbage collection, which are then averaged all benchmarks and presented in Figure 4.

For all hardware units, energy consumption is reduced by hardware adaptation, but increased by garbage collection. With all heap sizes, hardware adaptation reduces a CU's energy by 20% or more over the baseline. Hence, hardware adaptation effectively offsets the energy increase owing to garbage collection. Garbage collection's impact on CU energy consumption varies by CUs. With more frequent garbage collections, issue queue, reorder buffer, and L2 cache's energy consumption increases faster than L1I and L1D cache.

Since garbage collection changes program behavior, it may affect hardware adaptation's performance. To study the impact of garbage collection on hardware adaptation, we obtain each configurable hardware unit's energy consumption results with different heap sizes. To avoid the interference of one CU to another one's adaptation, each time only one CU is adapted, and all other units keep their largest sizes throughout program execution. Then the CU energy results are normalized to the corresponding fixed size unit results with the same heap sizes. The results, averaged across all benchmarks, are shown in Figure 5.

First, garbage collection's impact on the three caches differs. The adaptation efficiencies of both level-one caches increase as the heap size decreases. As shown in Table 7, garbage collection improves L1D and L1I caches' miss rates, allowing the two caches to use more aggressive configurations for higher energy reduction. On the other hand, garbage collection hurts L2 cache's adaptation efficiency, owing to the increase of L2 cache misses caused by garbage collection (Table 7). In this case, reducing the L2 cache size will inevitably incur more misses and impair the performance. Hence, with more garbage collection activities, the L2 cache tends to uses larger configurations and reduces less energy. Moreover, garbage collection adversely affects the adaptation of issue queue, while has minimal impact on the reorder buffer's adaptation.

5.2.2 Garbage collection versus mutator

Figure 6 separates the activities of the garbage collector from those of the mutator, and investigates the adaptation efficiencies on the garbage collector (GC) and the mutator (Mt) respectively.

First, the CUs tend to use smaller configurations on the garbage collector than on the mutator. Compared to the mutator, the garbage collector possesses distinct runtime characteristics that prefer a simple processor core and small caches for high energy reduction. Being small enough [6], the GC code can normally be held in a small instruction cache. During garbage collection, pointer-chasings to find surviving heap objects result in many data accesses that have poor temporal locality. Hence, reducing L1D and L2 cache sizes rarely affects the garbage collector's performance. Being memory bound, the garbage collector usually has little ILP, and thus smaller issue queue and reorder buffer can be used with minimal performance impact. Hence, all the five CUs achieve higher energy reduction on the garbage collector than on the mutator. Furthermore, a CU's energy is rarely affected by the frequency of garbage collections.

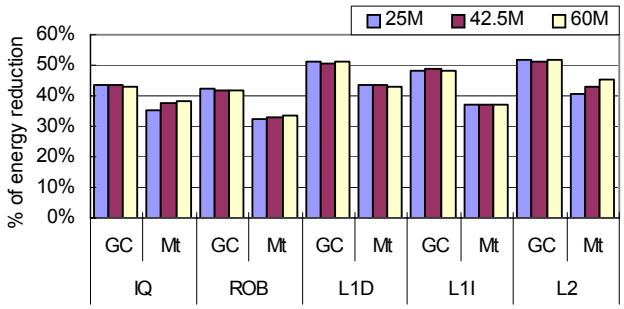

Figure 6. Comparison of garbage collector and mutator on configurable unit energy reduction.

The mutator results in Figure 6 represent the garbage collection's impact on the mutator, which vary by CUs. With smaller heaps and more garbage collections, IQ, ROB, and L2 cache's energy reduction on the mutator drops. On the other hand, L1D and L1I caches' adaptation efficiencies rarely changes as heap size decreases. The mutator is responsible for the overall variations of a CU's adaptation efficiency (Figure 5) as heap size changes.

5.3 Key insights

This research reveals that JIT optimization and garbage collection interfere with hardware adaptation. Both JIT optimization and garbage collection alter program behavior and runtime requirements. In adaptive microarchitectures, such changes of runtime requirements can considerably affect the adaptation decisions of configurable hardware units, and eventually influence the overall energy consumption of the underlying adaptive microarchitecture.

This research also studies the adaptation preferences of CUs on the JIT optimizer and the garbage collector. Owing to their distinct runtime characteristics, such as both VEE services' poor

data cache performance, the two dynamic optimization services have adaptation preferences differing substantially from the applications. For instance, both JIT compiler and garbage collector prefer a simple core for energy reduction. On the other hand, the JIT optimizer usually requires larger data caches to sustain its performance, while the garbage collector chooses smaller caches with minimal performance loss.

The phenomenon can be exploited by both adaptation microarchitectures and hardware adaptation schemes for high energy efficiency. A VEE-service-aware adaptive microarchitecture can improve its energy efficiency by providing hardware configurations accurately matching the runtime requirements of the VEE services. With a better understanding of the VEE services' runtime requirements, a hardware adaptation scheme can also significantly reduce the tuning process for those VEE services by avoiding testing unpromising configurations, which further improves the energy efficiency of the underlying adaptive microarchitecture. In a multi-threaded execution environment like Jikes RVM, the JIT compiler and the garbage collector use separate threads to the application, which eases the separation of VEE services from applications and makes hardware tuning for the VEE services feasible.

6. Conclusions

In this paper, we explore the impact of dynamic execution on a program's power dissipation and energy consumption, as well as the interference of dynamic execution on hardware adaptation. Due to the use of JIT optimization and garbage collection, programs executed by a virtual execution environment usually possess differing runtime characteristics than their statically compiled counterparts. We find that by reducing instruction counts, JIT optimization significantly reduces energy consumption, while garbage collection incurs runtime overhead that consumes more energy. Interestingly, both JIT optimization and garbage collection decrease the average power dissipated by a program.

The study also reveals that dynamic execution does interfere with hardware adaptation, and may considerably affect configurable units' energy consumption. We also study the adaptation preferences of CUs on the JIT optimizer and the garbage collector, demonstrating that they can differ substantially from the units' adaptation decisions on the application code.

The insights gained in this paper point to novel techniques that can further reduce microprocessor energy consumption. For instance, the tuning latency of adapting multiple CUs increases exponentially as more units become configurable. This research indicates that the adaptation preferences of the JIT optimizer and the garbage collector differ substantially from those of the application code. By identifying the code regions belonging to the JIT optimizer and the garbage collector, we can avoid the prolonged tuning latency for the JIT optimizer and the garbage collector, and greatly improve a CU's adaptation efficiency on those code regions. In the future, we plan to investigate this and other such avenues to improve microprocessor energy consumption.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. This research was supported in part by NSF grant 0429806, and by IBM, Intel and AMD Corporations.

References

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

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

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

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

[5] S. Blackburn, P. Cheng, K. McKinley, Myths and Realities: The Performance Impact of Garbage Collection. In Proceedings of ACM SIGMETRICS/Performance, 2004.

[6] S. Blackburn, R. Jones, K. McKinley, and J. Moss. Beltway: Getting Around Garbage Collection Gridlock, In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2002.

[7] 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.

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

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

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

[11] C.-H. Hsu, and U. Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. the Conference on Programming Languages, Design, and Implementation, 2003.

[12] S. Hu, M. Valluri, and L. John, Effective Adaptive Computing Environment Management Using a Dynamic Optimization System, 2005 International Symposium on Code Generation and Optimization, March 2005.

[13] X Huang, J. Moss, K. McKinley, S. Blackburn, and D. Burger, Dynamic SimpleScalar: simulating Java Virtual Machines. The First Workshop on Managed Run Time Environment Workloads, 2003.

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

[15] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of Compiler Optimizations on System Power Proceedings of the 2000 Design Automation Conference, 2000.

[16] 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.

[17] J. Seng, and D. Tullsen. The Effect of Compiler Optimizations on Pentium 4 Power Consumption, Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures, 2003.

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

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

[20] 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.

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

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

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

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

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

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

[27] .NET technology, http://www.microsoft.com/net/default.mspx