Adaptive Garbage Collection for Battery-Operated Environments *

G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin
Dept. of Computer Science and Engg.
The Pennsylvania State University
University Park, PA 16802
{ gchen, kandemir, vijay, mji }@cse.psu.edu

M. Wolczko
Sun Microsystems
Palo Alto, CA
mario@eng.sun.com

Abstract

Energy is an important constraint for battery-operated embedded Java environments. In this work, we show how the garbage collector (GC) can be tuned to reduce the energy consumption of Java applications. In particular, we show the importance of tuning the frequency of invoking GC based on object allocation and garbage creation rates to optimize leakage energy consumption. We reduce the leakage energy by exploiting the supply-gated leakage power optimization that is controlled by the GC. In this mechanism, power supply to memory banks that do not hold any useful data can be shut down. We implement a new adaptive GC mechanism within Sun's KVM that optimizes the ability to shut down more banks. An evaluation of our approach using various embedded applications shows that the adaptive garbage collection scheme is effective in reducing the system energy consumption across different hardware configurations.

1 Introduction

Java-based language environments and Java-enabled devices are fast becoming popular in the embedded computing world. Some of the primary features of Java that make it attractive for embedded and energy-sensitive environments are platform independence (write-once, run-anywhere), on-demand loading and compilation, remote update/ execution, the advanced software development features of object oriented programming, automatic garbage collection, and pointer and type safety. As more and more battery-operated embedded environments employ Java, we believe that system designers should focus on individual components of the JVM and try to make them as energy-efficient as possible [15, 7].

The amount of energy consumed by a device determines the duration of battery life between recharges. The main sources of energy consumption in current CMOS-technology based processors are dynamic and leakage energy [1]. Dynamic energy is consumed whenever a component is utilized. In contrast, leakage energy is consumed as long as power supply is maintained and is independent of component activity. While leakage energy used to be an insignificant part of the overall energy consumption, due to the continued shrinking of transistors and associated technology changes, leakage power has become an important portion of overall energy consumption. In fact, it is shown to be the dominant part of the chip power budget for 0.10 micron technology and below [2]. Supply gating is an effective mechanism for reducing the leakage energy that shuts down the power supply to components that are idle [2, 12]. However, supply gating has two repercussions. First, when the unit that is supply-gated is a memory element, the state of the memory is lost. Second, reactivating the power supply to the unit after a period of idleness requires a few hundreds of processor cycles. In this work, our focus is on applying supply gating mechanism to memory elements under the control of a garbage collector (GC) [9].

One of the important components of the Java virtual machine is the GC. The GC automatically determines what objects a program is no longer using, and recycles them

*This research is supported in part by NSF Awards 0073419, 0103583 & 0097998; NSF CAREER Awards 0093082 & 0093085; GSRC.

for other use. The GC is a suitable component to power off memory banks that do not hold any useful data as it has access to accurate object usage [1]. In particular, after performing a collection, the GC can turn off a bank if it does not contain any live objects. In a traditional GC implementation, garbage collection is generally invoked when there is not enough memory to allocate the new object requested. While this strategy is acceptable for a pure performance-oriented JVM implementation, in an energy-sensitive environment with a banked memory architecture, delaying collecting garbage until available free memory has run out may lead to wasted leakage energy consumption. This is because the leakage energy expended on a bank during the time it holds only garbage (dead objects) is effectively wasted. One way of reducing this wasted energy is to perform garbage collections more frequently than necessary. That is, instead of waiting until heap space is exhausted, we can invoke the GC more frequently (e.g., at regular intervals), thereby detecting the dead objects earlier and turning off power supply to the memory banks aggressively.

In this paper, we focus on an embedded Java environment and demonstrate that an adaptive garbage collection strategy, that is, a strategy that tunes the frequency of GC invocations based on object creation and garbage generation patterns, can lead to large energy savings in memory. Using KVM, Sun Microsystem's JVM designed for embedded and battery-operated environments, and a set of nine applications that are typical for hand-held devices, we also show in this paper that the energy behavior resulting from our adaptive garbage collection strategy is competitive with the best fixed allocation garbage collection. Our results also indicate that, for most of our benchmarks, as compared to the best fixed allocation garbage collection, the adaptive GC strategy incurs much less performance penalty. Based on these results, we encourage future GC implementors for embedded Java systems to explore adaptive garbage collection strategies.

The remainder of this paper is organized as follows. Section 2 presents our adaptive GC strategy. Section 3 introduces our benchmarks, presents the GC strategies compared, and discusses our simulation environment. Section 4 gives experimental results showing the effectiveness of our strategy in reducing energy consumption. We briefly discuss some related work in Section 5. Finally, Section 6 concludes the paper by summarizing our major contributions and by giving a brief outline of the planned future work on this topic.

[1] Memory structures are typically organized as smaller partitions called banks or sub-banks for power and performance optimizations

2 Our Approach

The garbage collector in KVM is invoked when, during objection allocation, the available free heap space is insufficient to accommodate the object to be allocated. During the time between an object becomes garbage and the time it is detected to be so and collected, the object occupies space in heap memory unnecessarily. While in a pure performance-oriented environment this may not be much of a problem, in an energy-conscious environment it can cause unnecessary leakage energy consumption. For example, suppose that a bank holds only a single object that becomes garbage at time t_1. Assuming that this object is collected only at time t_2, one can see that the bank consumes leakage energy during the time period [t_1,t_2] unnecessarily. Obviously, the larger the difference between these two times, the higher this wasted energy consumption. It is thus vital from the energy perspective to detect and collect garbage as soon as possible. However, previous GC algorithms that are known to collect some objects as soon as they become garbage (e.g., reference counting [9]) have other problems such as reclaiming circular structures and efficiency problems in updating counts. While many variants have been proposed to address such problems, these collectors have not been adopted widely. Then, an alternative option would be to increase the invocation frequency of widely-used garbage collectors such as the mark and sweep collectors employed the KVM. However, the potential savings obtained through more frequent garbage collection should be balanced with the additional overhead required to collect the dead objects earlier (i.e., the energy cost of garbage collection). Accesses to the heap and execution of instructions in the processor when the GC executes incur additional dynamic energy. Further, additional leakage energy is consumed due to the increased time expended due to more frequent invocation of the GC.

An important issue then is to come up with a suitable garbage collection frequency that balances the potential leakage energy savings with the additional overheads. It is not difficult to see that if we fix the GC frequency at a specific value over all applications, this value may be suitable for only some of these applications. This is because each application has a different object allocation and garbage generation pattern. Our experimental results detailed in Section 4 shows that this is really the case. Another option would be tuning the frequency of garbage collection dynamically (i.e., during execution).

In this section, we describe an adaptive garbage collection strategy that tunes the frequency of the garbage col-

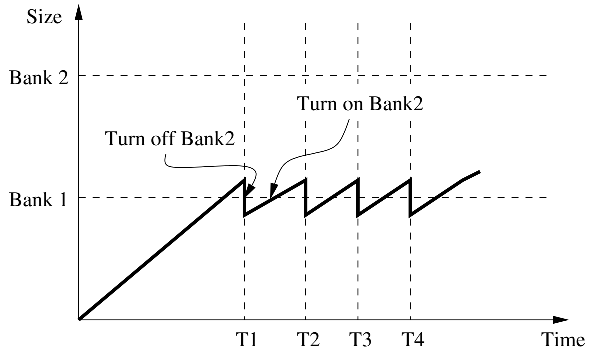

Figure 1: Garbage collector is invoked at T1, T2, T3 and T4.

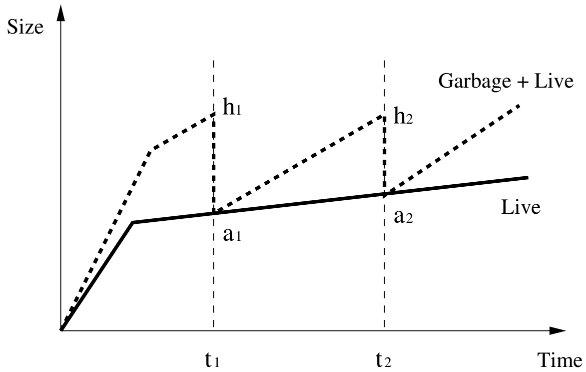

Figure 2: Garbage collection c_1 and c_2 are invoked at t_1 and t_2 respectively.

lection according to the dynamic behavior of the application at runtime. A basic principle is that *whenever there is an opportunity to turn off a bank that holds only dead objects, the garbage collector should be invoked* . Simple application of this principle, however, may cause frequent turn on and turn off of the same bank (thrashing) if too few objects are collected at each collection (Figure 1). To avoid thrashing, we need an additional principle: *garbage collector should not be invoked unless a certain number of objects have become garbage* .

It should be noted, however, without actually invoking GC, it is not possible to tell exactly how many (and what size) objects have become garbage (since the last collection phase). However, due to the fact that many object allocation/deallocation patterns exhibit some regularity in one form or another [9], we can use past (history) information to estimate the size of the dead (unreachable) objects.

Let k_1 and k_2 be the object creation rate and the garbage generation rate, respectively. Assume two successive garbage collections, c_1 and c_2, that are invoked at times t_1 and t_2, respectively (Figure 2). Assume further that after c_1, the total size of the (live) objects in the heap is a_1 and that, at t_2, just before c_2, the total size (including the unreachable objects) in the heap is h_2. Finally, let us assume that after c_2 the total size of the objects in the heap (excluding the unreachable ones) is a_2. Based on this, the object creation rate during the time period $[t_1, t_2]$ is:

$$k_1 = (h_2 - a_1)/(t_2 - t_1).$$

Similarly, the garbage generation rate during $[t_1, t_2]$ can be expressed as:

$$k_2 = (h_2 - a_2)/(t_2 - t_1).$$

In our strategy, we use k_1 and k_2 to estimate the object creation and garbage generation rates after t_2 until the next collection, say c_3, is invoked. After c_3, the values of k_1 and k_2 will be updated to adapt their values to (potentially) changing heap behavior.

Following each allocation i that occurs at time t_i, we use the following strategy to decide whether to invoke GC or not:

- If after GC, at least two banks can be supply gated, the garbage collector should be invoked:

  $$b(h_i + s_i) - b(h_i + s_i - k_2(t_i - t_2)) \geq 2,$$

  where $b(s)$ is a function that returns the number of banks, given the total object size s; h_i is the total object size in the heap right before the allocation; and s_i is the size of the object to be allocated.

- We consider the ability to supply-gate only one bank differently, as a fast object allocation rate may require the re-activation of the supply gated unit within a short duration of time. Due to the time penalty for re-activation, it may be more beneficial to keep the power supply to the bank on if it would result in re-activation within a few allocations.

  If after the collection, one bank could be supply gated, then the space GC creates must allow the application to run for an interval no shorter than L:

  $$b(h_i + s_i) - b(h_i + s_i - k_2(t_i - t_2)) = 1 \quad \text{and}$$

  $$\begin{aligned} &b(h_i + s_i - k_2(t_i - t_2))B \\ &-(h_i + s_i - k_2(t_i - t_2)) \geq k_1 L. \end{aligned}$$

  In our implementation, L is measured in terms of the number of allocations. In the last formulation above, B is the size of a memory bank.

- As our estimation is not perfect, to limit the penalty due to misestimation, the GC should be invoked after every B byte allocation if no collection happens in between.

The overhead of our adaptive algorithm is not too much. This is because our decision making requires simple calculations and h_i and a_i can be easily obtained by adding only a few lines to garbage collector codes. It should also be stressed that both the information collection and the decision making activities consume energy. It is true that more sophisticated heuristic algorithms may give more accurate predictions. However, such algorithms can also be expected to be more complex and to require more detailed past history information, thereby potentially incurring a more significant performance (execution time) overhead.

3 Experimental Setup and Benchmarks

3.1 KVM and its Garbage Collector

Sun's K Virtual Machine (KVM) [13] is a virtual machine designed with the constraints of inexpensive embedded/mobile devices in mind. It is suitable for devices with 16/32-bit RISC/CISC microprocessors/ controllers, and with as little as 160 KB of total memory available, 128 KB of which is for the storage of the actual virtual machine and libraries themselves. The KVM technology targets smart wireless phones, mainstream personal digital assistants, pagers, and small retail payment terminals.

KVM uses two different garbage collectors, both based on mark-and-sweep [9]. These collectors differ from each other in that one of them supports compaction, whereas the other does not. A mark-and-sweep collector makes two passes over the heap. In the first pass (called the mark pass), a bit is marked for each object indicating whether the object is reachable (live). After this step, a sweep pass returns unreachable objects (garbage) to the pool of free objects. Our adaptive garbage collection strategy can be made to work with other popular collectors (e.g., a generational collector [9]) as well. In this work, we use only the compacting collector; the results with the non-compacting collector are similar, and omitted due to lack of space.

In the compacting mark-and-sweep collector used in KVM, permanent objects are distinguished from dynamic objects. A certain amount of space from the end of the heap is allocated for permanent objects and is called the permanent space. This is useful because the permanent space is not marked, swept, or compacted (since it contains permanent objects which will be referenced until the end of execution of application). The mark and sweep part of this collector is the same as the non-compacting collector. Compaction takes place on two occasions: (i) after the mark and sweep phase if the size of the object to be allocated is still larger than the largest free chunk of memory obtained after sweeping; and (ii) when the first permanent object is allocated, and, as needed, when future permanent objects are allocated. During compaction, all live objects are moved to one end of the heap. While allocating a new dynamic object, the free list is checked to see whether there is a chunk of free memory with enough space to allocate the object. If there is not, then the garbage collector is called. During garbage collection (after the sweep phase), it is checked whether the largest free chunk of memory (obtained after the sweep phase) satisfies the size to be allocated. If not, then the collector enters the compaction phase. After compaction, object allocation is attempted again. If there is still not any space, an out-of-memory exception is signaled. In the rest of this paper, this compacting mark-and-sweep collector is referred to as the *base collector* or *base* for short.

3.2 Different Versions

To see how our adaptive garbage collector performs, we compared it to a class of collectors called *k-allocation collectors* that call the GC (that is, the mark-and-sweep algorithm) once after every k object allocations. We conducted experiments with six different values of k: 10, 40, 75, 100, 250, and 500. In cases where the collection frequency of the base collector is lower than the k value used, we can expect that k-allocation collector will generate a better energy behavior. In fact, we can expect that for each application there is a k value (among the k values used in our experiments) that generates the best energy result. Consequently, for the best results, the GC frequency should be tuned to application behavior. However, this, in general, may not be possible as we may not have any information about the application's heap behavior until the runtime. Consequently, our adaptive scheme tries to detect the optimal frequency at runtime.

For each application from a given set of Java applications, a successful adaptive collector should generate at least the same energy behavior as the k-allocation collector that performs best for that application. For some applications, we can even expect the adaptive collector to generate better results than even the best k-allocation collector. This may occur, for example, when there are different phases during the course of the application's execution and each phase works best with a different garbage collection frequency. Thus, we expect the adaptive collector to be competitive with the best k-allocation collector from the energy perspective.

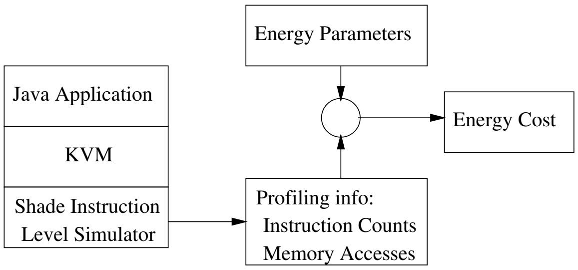

Figure 3: Simulation environment.

3.3 Architecture and Simulation Environment

In this work, we focus on a system-on-a-chip (SoC) based architecture that executes KVM applications. An SoC is an integrated circuit that contains an entire electronic system in a single sliver of silicon. Considering the fact that SoCs keep getting more and more complex, their energy demand is expected to increase significantly. This is particularly true for their memory systems as many of the applications run on SoCs are data intensive. Our SoC architecture has a CPU core and two main memory modules (in addition to some peripheral custom circuitry which is not relevant in this discussion). The processor in our SoC is a microSPARC-IIep embedded core. This core is a 100MHz, 32-bit five-stage pipelined RISC architecture that implements the SPARC architecture v8 specification. It is primarily targeted for low-cost uniprocessor applications.

In our SoC architecture, the main memory is composed of three parts. The first part contains the KVM code and associated class libraries; the second part is the heap that contains objects and method areas; and the third part holds the non-heap data that contain the runtime stack and KVM variables. Typically, the KVM code and the class libraries reside in a ROM. The ROM size we use is 128KB for the storage of the actual virtual machine and libraries themselves [4]. Since, not all libraries are used by every application, banked ROMs can provide energy savings. We activate the ROM partitions only on the first reference to the partition. A ROM partition is never disabled once it has been turned on. This helps to reduce the leakage energy consumption in memory banks not used throughout the application execution. While it may be possible to optimize the energy consumed in the ROM further using techniques such as clustering of libraries, in this study, we mainly focus only on the RAM portion of memory (SRAMs are commonly used in embedded environments as memory modules) which holds the heap. The heap (a default size of 128KB) holds both application bytecodes and application data, and is the target of our energy management strategies. An additional 32KB of SRAM is used for storing the non-heap data. We assume that the memory space is partitioned into banks and depending on whether a heap bank holds a live object or not, it can be shutdown (for saving leakage energy). In this work, each bank is assumed to be 16KB. Our objective here is to shutdown as many memory banks as possible in order to reduce leakage and dynamic energy consumption. Note that the operating system is assumed to reside in a different set of ROM banks for which no optimizations are considered here. Further, we assume a system without virtual memory support, as this is uncommon in many embedded environments [8].

To perform energy calculations, we built a custom simulation environment on top of the Shade [5] (SPARC instruction set simulator) tool-set (Figure 3). Shade is an instruction-set simulator and custom trace generator that simulates the SPARC V8 instruction set. Our simulator takes a KVM system with an application as input and keeps track of the energy consumption in the processor core (datapath), on-chip caches, and the on-chip SRAM and ROM memories. The datapath energy includes the energy spent during application execution and that expended during garbage collection. The energy consumed in the processor core is estimated by counting (dynamically) the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The energy consumption of the different instruction types is obtained using a customized version of a validated cycle accurate energy simulator [14]. The simulator is configured to model a five-stage pipeline similar to that of the microSPARC-IIep architecture. An analytical energy model (validated to be within 3% of actual values) similar to that proposed in [11] is used for evaluating the dynamic energy consumption of the memory elements, and a supply voltage of 1V and a threshold voltage of 0.2V. Further, we assume that the leakage energy per cycle of the entire memory is equal to the dynamic energy consumed per access. This assumption tries to capture the anticipated importance of leakage energy in 0.10 micron (and below) technologies [2]. Similar abstractions for leakage energy modeling have been employed in recent architectural studies due to the lack of validated leakage energy models [10, 3].

The memory system energy is divided into three portions: energy spent in accessing KVM code and libraries, energy spent in accessing heap data, and energy spent in accessing the runtime stack and KVM variables. Our simulator also allows the user to adjust the various parameters for these components. Energies spent in on-chip interconnects are included in the corresponding memory components. In this study, we assume that each memory bank can be in one of three states: R/W (Read/Write), Active, and Inactive. In the R/W state, the memory bank is being read or written. It consumes full dynamic energy as well as full leakage energy. In the active state, the memory bank is active (powered on) but not being read or written. It consumes no dynamic energy but full leakage energy. Finally, in the inactive state, the bank does not contain any useful data and is supply-gated. We conservatively assume that when in the inactive state a memory bank consumes 5% of its original leakage energy (as a result of supply-gating). While placing unused memory banks in inactive state may save significant leakage energy, it may also cause some performance degradation. Specifically, when a bank in the inactive state is accessed (to service a memory request), it takes 350 cycles to transition to the active state [3]. We term this time as the resynchronization time (or resynchronization penalty). It should be mentioned that both inactive state leakage energy consumption and resynchronization time are highly implementation dependent and are affected by the sizing of the driving transistors. The values that we used in this study are reasonable and conservative [3]. We also assume that the per cycle leakage energy consumed during resynchronization is the average of per cycle leakage energy consumed when the system is in the active state and that consumed when it is in the inactive state.

3.4 Benchmark Codes

To test the effectiveness of our energy saving strategy, we collected nine applications shown in Table 1. These applications represent a group of codes that are executed in energy-sensitive devices such as hand-held computers and electronic game boxes, and range from utilities such as calculator and scheduler, embedded web browser to game programs.[2] We believe that these applications represent a good mix of codes that one would expect to run under KVM-based, battery-sensitive environments.

[2]Our applications and GC executables are publicly available from www.cse.psu.edu/∼gchen/kvmgc/

4 Experimental Results

Unless otherwise stated, in all the results reported in this section we use an L value of 20 allocations (see Section 2) and a bank size of 16KB.

4.1 Heap Energy

Figure 4 shows the heap energy consumption for our different versions. All values shown in this graph are *normalized* with respect to the base collector. In addition to base and our adaptive collector (denoted `adpt`), we report the results for a set of k-allocation collectors with k values of 10, 40, 75, 100, 250, and 500. Above each bar is the normalized energy consumption in heap. From these results, we can make the following observations. First, as far as the k-allocation collector is concerned, we clearly see that different applications work best with different garbage collection frequencies. Second, our adaptive garbage collection strategy reduces the heap energy of the base collector in KVM significantly; the average (over all applications) heap energy saving is 28.4%. Third, for a given benchmark, the adaptive strategy is always competitive with the best k-allocation collector. For example, in `Dragon`, the adaptive strategy comes close to the 10-allocation collector (the best k-allocation collector for this benchmark). Similarly, in `Kvideo` (resp. `Kshape`), our adaptive collector generates competitive results with 75-allocation (resp. 250-allocation) collector. These results clearly show that the adaptive garbage collection is very successful in optimizing heap energy.

4.2 Overall Energy

While our objective is optimizing the energy consumption in heap, changing the frequency of garbage collections might also change the energy profile in other parts of the system such as ROM, the memory banks that hold runtime stack, and the CPU datapath. For example, a more frequent collection is expected to increase the energy spent in CPU due to garbage collection. To analyze this behavior, we give in Figure 5 the energy consumption in all components that we are interested in. We observe from these results that, for each code in our experimental suite, the adaptive strategy comes very close to the best k-allocation collector, even when one focuses on the overall energy consumption.

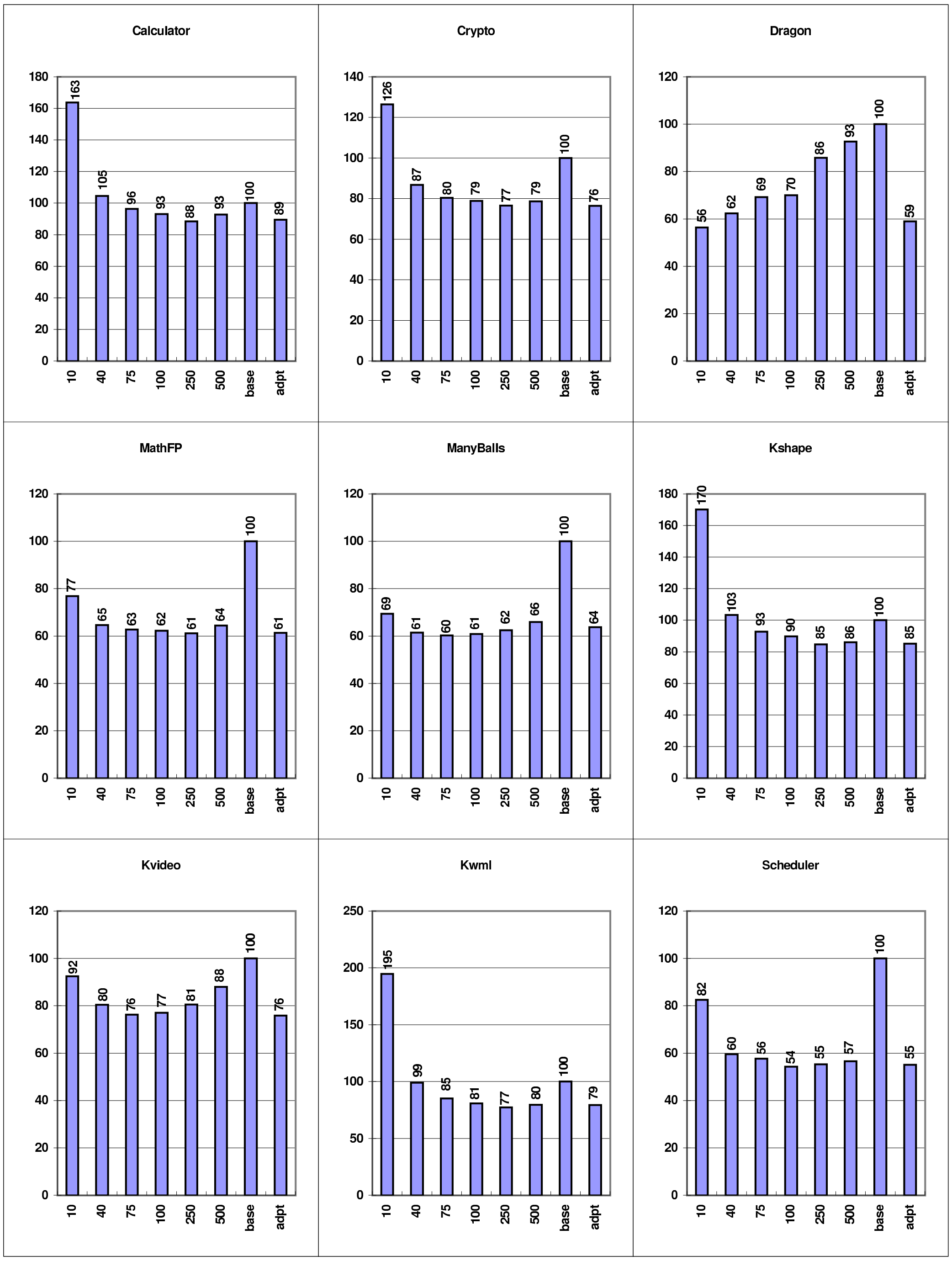

Figure 4: Normalized energy consumption (static+dynamic) in the heap memory.

Table 1: The benchmarks used in our experiments.

Application	Brief Description	Source
`Calculator`	Arithmetic calculator	www.cse.psu.edu/∼gchen/kvmgc/
`Crypto`	Cryptography	www.bouncycastle.org
`Dragon`	Game program	comes with Sun's KVM
`Kshape`	Electronic map on KVM	www.jshape.com
`Kvideo`	KPG decoder	www.jshape.com
`Kwml`	WML browser	www.jshape.com
`Manyballs`	Game program	comes with Sun's KVM
`MathFP`	Integer math lib	home.rochester.rr.com/ohommes/MathFP/
`Scheduler`	Weekly/daily scheduler	www.cse.psu.edu/∼gchen/kvmgc/

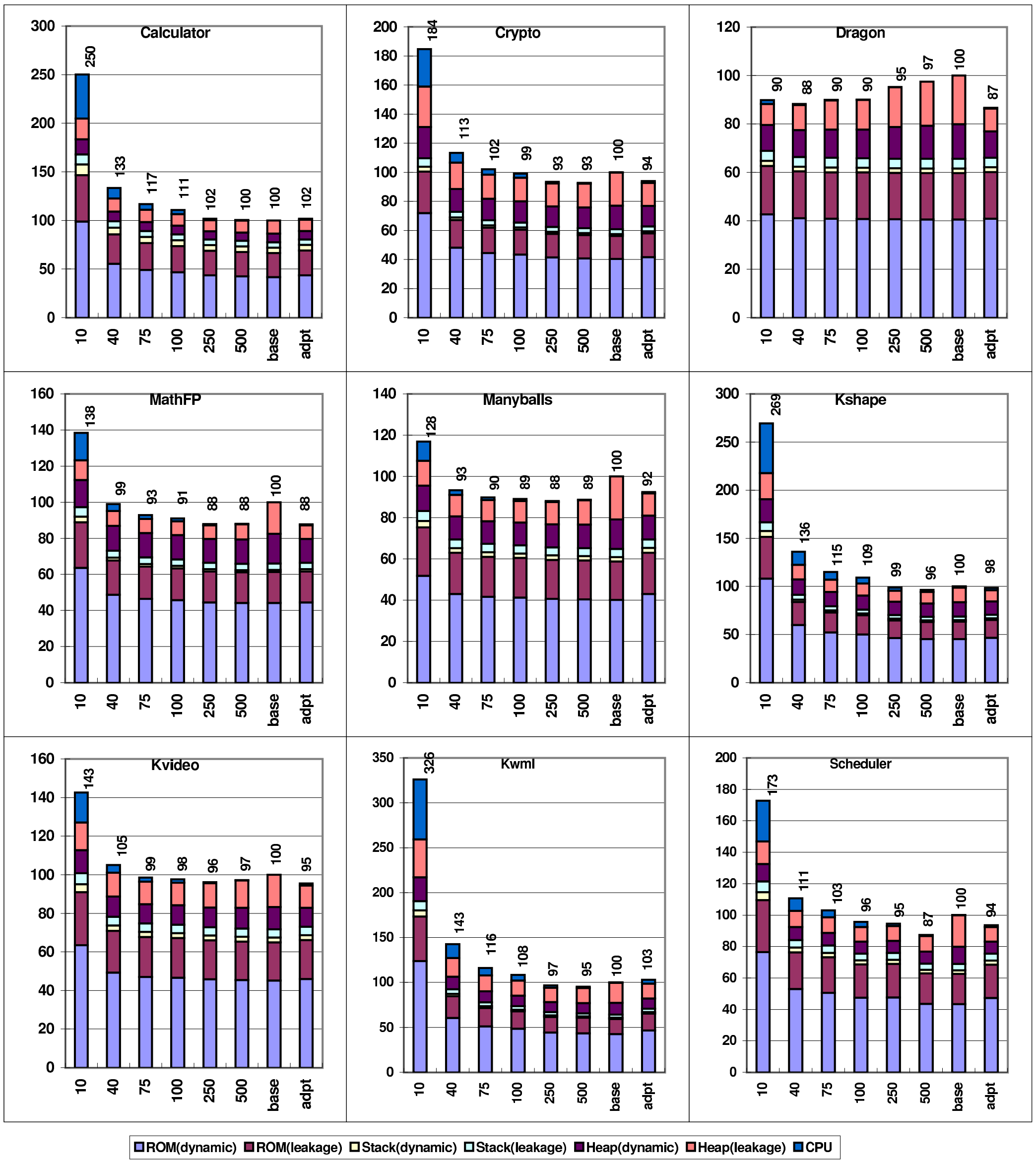

Figure 5: Normalized energy consumption breakdown.

4.3 Execution Profiles

To better understand the energy savings due to adaptive garbage collection, we give further data illustrating where the energy benefits come from. First, we give in Table 2 the number of times each version invokes garbage collection. From these results we see that the adaptive strategy calls garbage collection much more frequently than the base collector. This explain why it reduces the heap energy consumption significantly. In fact, we can see from this table that even the 500-allocation collector calls garbage collection more frequently (on average) than the base collector, indicating that the base collector is very slow in invoking the collector. As mentioned earlier however, calling garbage collection more frequently incurs an extra runtime overhead. Therefore, the garbage collector should not be invoked unless it is necessary (i.e., unless it is expected to save energy by powering off bank(s)). We see from Table 2 that the adaptive strategy calls collection less frequently than the k-allocation strategy when k is 10, 40, 75, and 100. Therefore, its runtime overhead is less than these collectors. Considering that the adaptive strategy is competitive with all these k-allocation collectors, one can see that the adaptive strategy achieves the same energy behavior as these allocators with much less performance penalty. As compared to the remaining versions, it calls collection more frequently on the average, but one application (`Kwml`) is responsible for this average.

Figure 6 shows the heap footprints of different versions for three applications: `Calculator`, `Dragon`, and `ManyBalls`. In these graphs, the x-axis corresponds to the number of allocations made during the course of execution. The y-axis, on the other hand, represents the occupied memory space in heap (by alive or dead objects). Each horizontal line on the y-axis corresponds to a bank boundary, starting from the bottom with the first bank. In these graphs, for sake of clarify, we show only a 75-allocation collector, the base collector and our adaptive collector. We can clearly observe the impact of changing the frequency of collections on the number of active power banks required. Let us first focus on `Calculator`. In this application, the base collector does not invoke any collection. We observe that the adaptive collector invokes collection less frequently than the 75-allocation collector (this is also true for all studied k-allocation collectors except the 500-allocation collector). When using the adaptive collector, the GC is invoked just in time before the new bank is activated. Therefore, it's energy savings are competitive with the other k-allocation collectors that invoke GC more frequently. It should also be noted that the 500-allocation collector (not shown in figure) activates collections a bit late (after the second bank is accessed). In `Dragon`, on the other hand, we observe a different behavior. This application experiences frequent object allocations, putting a pressure on the heap memory. Therefore, the best energy results are obtained by calling the GC frequently. Consequently, the adaptive strategy and 10-allocation collector (not shown in figure) achieve the best result. Finally, in `ManyBalls`, the adaptive collector strikes a balance between very frequent and very slow garbage collections.

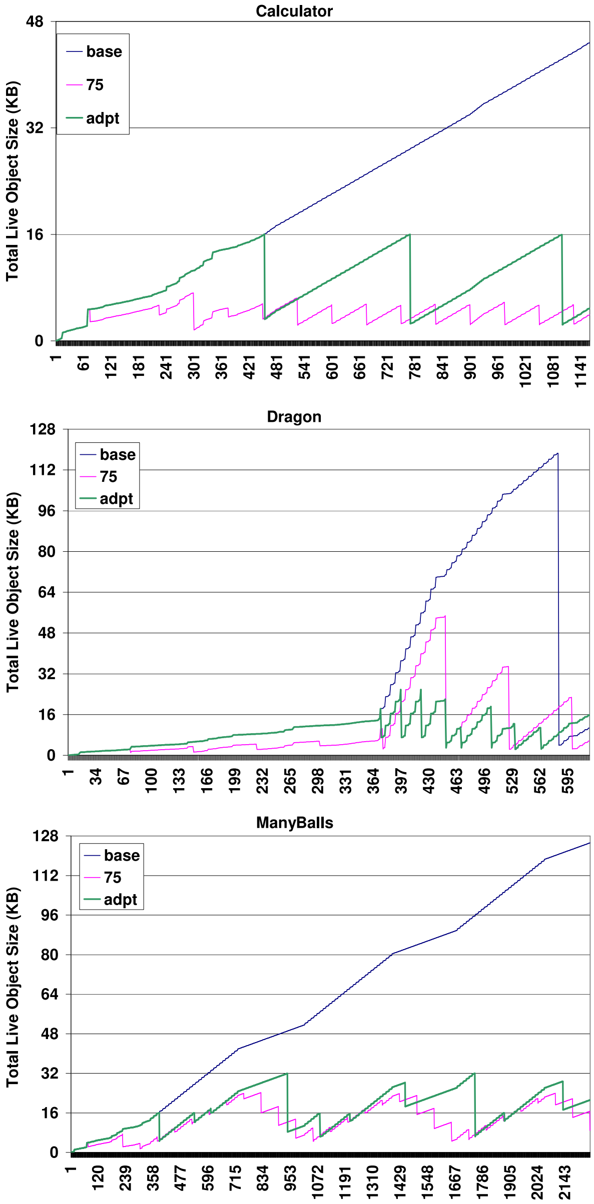

Figure 6: Execution footprints of `Calculator`, `Dragon`, and `ManyBalls` under different collection frequencies.

Table 2: The number of GC activations for each benchmark.

Application	10	40	75	100	250	500	base	adpt
`Calculator`	104	24	12	9	3	1	0	3
`Crypto`	595	147	78	58	23	11	3	26
`Dragon`	84	15	8	6	2	1	1	8
`MathFP`	825	205	109	81	32	16	2	21
`ManyBalls`	214	48	25	19	7	3	0	10
`Kshape`	2280	593	338	266	136	93	50	100
`Kvideo`	102	23	12	9	3	1	0	4
`Kwml`	4936	1099	571	425	167	83	19	285
`Scheduler`	973	240	127	95	37	18	2	28
Average:	1063.7	266	142.2	107.5	45.6	25.2	8.6	53.8

4.4 Performance Behavior

It is also important to evaluate the performance impact of our adaptive garbage collection strategy. Table 3 gives, for each version, the percentage increase in execution cycles with respect to the base collector. We see that the average (across all benchmarks) execution time increase due to our adaptive strategy is only 4.20%, which is less than the increases in execution times when k-allocation strategies with $k = 10, 40, 75,$ and 100 are employed. Considering the fact that our strategy is competitive with these allocators as far as the energy behavior is concerned, the performance overhead of the adaptive scheme is acceptable. As an example, in `Dragon`, our approach is competitive with the 10-allocation collector as far as the energy consumption is concerned. However, the 10-allocation collector increases the execution time of this application by 14.53%, a much larger figure than 0.68%, the corresponding increase in the adaptive collector.

4.5 Bank Size Variations

Recall that so far all experiments have been performed using a heap size of 128KB and a bank size of 16KB. In this subsection, we change the bank size (by keeping the heap size fixed at 128KB) and measure the sensitivity of our results to the bank size variation. Figure 7 gives the normalized heap energy consumptions for two of our benchmarks: `Kvideo` and `Dragon`. Our experiments with other benchmarks showed similar trends; so, they are not reported here in detail. The results given in Figure 7 are values normalized with respect to the heap energy consumption of the base allocator with 16KB bank size and 128KB heap size. We could not run `Dragon` with 4KB bank size due to the large inter-bank objects allocated by this benchmark (currently, our implementation cannot allocate objects larger than bank size). We observe two trends from this graph. First, the effectiveness of our adaptive collection strategy as well as that of k-allocation collectors increase with the reduced bank size. This is because a smaller bank size gives more opportunity to GC to turn off heap memory regions in a finer-grain manner. Consequently, more heap memory space can be turned off. Second, our adaptive strategy continues to follow the best k-allocation collector even when different bank sizes are used; that is, it can be used with different bank sizes.

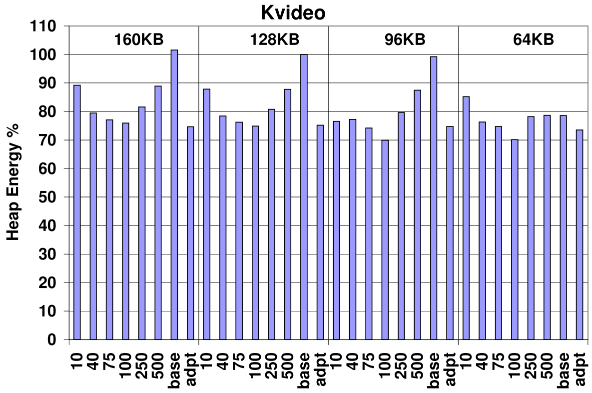

Figure 8: Results with different heap sizes (`Kvideo`).

4.6 Heap Size Variations

In this subsection, we fix the bank size at 16KB (our default value) and report experimental results obtained when different heap sizes are used. We focus on `Kvideo` only; but, the results observed with other benchmarks are similar. The heap sizes used in this set of experiments are 64KB, 96KB, 128KB (our default), and 160KB. We observe from the results shown in Figure 8 that our adaptive strategy continues to follow the best k-allocation collector and that its performance is not very sensitive to the heap size variation.

Table 3: Percentage increase in execution cycles (with respect to the base collector).

Application	10	40	75	100	250	500	adpt
`Calculator`	92.58	21.71	11.78	8.01	3.00	1.88	2.86
`Crypto`	80.57	19.89	10.34	7.73	2.73	1.08	3.29
`Dragon`	14.53	1.08	0.54	0.40	0.07	0.03	0.68
`MathFP`	47.82	11.05	5.34	3.64	0.65	-1.21	0.60
`ManyBalls`	25.71	6.41	8.52	2.56	1.19	0.66	6.93
`Kshape`	40.83	31.97	14.58	9.74	1.47	-1.79	2.16
`Kvideo`	38.20	8.59	4.11	3.11	1.39	0.59	1.68
`Kwml`	92.60	42.63	20.48	13.98	3.64	2.22	9.64
`Scheduler`	70.98	20.53	16.83	9.13	10.80	0.27	9.96
Average:	55.98	18.21	10.28	6.48	2.77	0.41	4.20

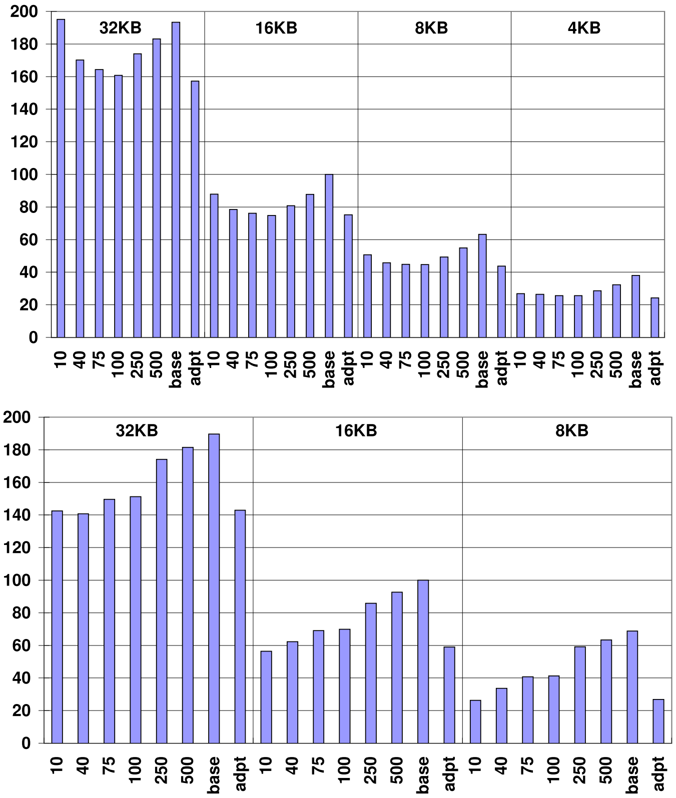

Figure 7: Results with different bank sizes: Top: `Kvideo`, Bottom: `Dragon`.

5 Related Work

Most efforts concerning the garbage collector and Java virtual machine have not focused on optimizing the energy consumption. However, there have been recent efforts at characterizing the energy behavior of Java applications. Flinn et al. [7] quantify the energy consumption of a pocket computer when running Java virtual machine. In [15], the energy behavior of a high-performance Java virtual machine is characterized. Diwan et al. [6] analyzed four different memory management policies from the performance as well as energy perspectives. Our work is most closely related to the work presented in [3]. In [3], the energy impact of various GC parameters was characterized using the KVM but makes no attempt to design an adaptive garbage collector. In contrast, this work proposes an adaptive GC strategy that tailors its invocation frequency to maximize energy savings.

6 Conclusions

Unnecessary leakage energy can be expended in maintaining the state of garbage objects until they are recognized to be unreachable using a collector. This wasted leakage energy is proportional to the duration between when an object becomes garbage and the time when it is recognized to become garbage. While invoking the GC more frequently can reduce this wasted leakage that needs to be balanced with the energy cost of invoking the GC itself. In this paper, we show the design of an adaptive GC strategy that tailors its invocation frequency to account for these tradeoffs based on the object allocation and garbage creation frequencies. We implemented this adaptive GC within a commercial JVM and showed that it is effective in reducing the overall system energy.

References

[1] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems,* 5(2), pp.115-192, April 2000.

[2] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[3] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded Java environment. In Proc. *the 8th International Symposium on High-Performance Computer Architecture,* Cambridge, MA, February 2-6, 2002.

[4] CLDC and the K Virtual Machine (KVM). http://java.sun.com/products/cldc/.

[5] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In Proc. *ACM Sigmetrics Conference on the Measurement and Modeling of Computer Systems*, pp. 128-137, May 1994.

[6] A. Diwan, H. Li, D. Grunwald and K. Farkas. Energy consumption and garbage collection in low powered computing. http://www.cs.colorado.edu/∼diwan. University of Colorado-Boulder.

[7] J. Flinn, G. Back, J. Anderson, K. Farkas, and D. Grunwald. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In Proc. *International Conference on Measurement and Modeling of Computer Systems,* June 2000.

[8] W-M. W. Hwu. Embedded microprocessor comparison. http://www.crhc.uiuc.edu/IMPACT/ece412/public_html/Notes/412_lec1/ppframe.htm.

[9] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons. 1999.

[10] S. Kaxiras, Z. Hu, M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In Proc. *The 28th International Symposium on Computer Architecture,* June 2001.

[11] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *International Symposium on Low Power Electronics and Design*, page 143, August 1997.

[12] M. D. Powell, S-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In Proc. *the ACM/IEEE International Symposium on Low Power Electronics and Design,* August 2000.

[13] R. Riggs, A. Taivalsaari and M. VandenBrink. *Programming Wireless Devices with the Java 2 Platform*. Addison Wesley, 2001.

[14] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *the International Symposium on Computer Architecture,* Vancouver, British Columbia, June 2000.

[15] N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam and M. J. Irwin. Energy Characterization of Java Applications from a Memory Perspective. In Proc. *USENIX Java Virtual Machine Research and Technology Symposium,* April 2001.