Tuning Garbage Collection for Reducing Memory System Energy in an Embedded Java Environment

G. CHEN, R. SHETTY, M. KANDEMIR, N. VIJAYKRISHNAN and M. J. IRWIN
Microsystems Design Lab, The Pennsylvania State University
and
M. WOLCZKO
Sun Microsystems, Inc.

Java has been widely adopted as one of the software platforms for the seamless integration of diverse computing devices. Over the last year, there has been great momentum in adopting Java technology in devices such as cellphones, PDAs, and pagers where optimizing energy consumption is critical. Since, traditionally, the Java virtual machine (JVM), the cornerstone of Java technology, is tuned for performance, taking into account energy consumption requires reevaluation, and possibly redesign of the virtual machine. This motivates us to tune specific components of the virtual machine for a battery-operated architecture. As embedded JVMs are designed to run for long periods of time on limited-memory embedded systems, creating and managing Java objects is of critical importance. The garbage collector (GC) is an important part of the JVM responsible for the automatic reclamation of unused memory. This article shows that the GC is not only important for limited-memory systems but also for energy-constrained architectures.

This article focuses on tuning the GC to reduce energy consumption in a multibanked memory architecture. Tuning the GC is important not because it consumes a sizeable portion of overall energy during execution, but because it influences the energy consumed in the memory during application execution. In particular, we present a GC-controlled leakage energy optimization technique that shuts off memory banks that do not hold live data. Using two different commercial GCs and a suite of thirteen mobile applications, we evaluate the effectiveness of the GC-controlled energy optimization technique and study its sensitivity to different parameters such as bank size, the garbage collection frequency, object allocation style, compaction style, and compaction frequency. We observe that the energy consumption of an embedded Java application can be significantly more if the GC parameters are not tuned appropriately. Further, we notice that the object allocation pattern and the number of memory banks available in the underlying architecture are limiting factors on how effectively GC parameters can be used to optimize the memory energy consumption.

Categories and Subject Descriptors: B.3.m [**Memory Structure**]: Miscellaneous

General Terms: Design, Measurement

This work was supported in part by NSF CAREER Awards 0093082 & 0093085; NSF Awards 0073419, 0082064, 0103583; and an award from GSRC.

Authors' addresses: G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, Microsystems Design Lab, The Pennsylvania State University, University Park, PA; email: {gchen,shetty,kandemir,vijay,mji}@cse.psu.edu; M. Wolczko, Sun Microsystems, Inc., Palo Alto, CA; email: mario@eng.sun.com.

Additional Key Words and Phrases: Garbage collector, Java Virtual Machine (JVM), K Virtual Machine (KVM), low power computing

1. INTRODUCTION

Java is becoming increasingly popular in embedded/portable environments. It is estimated that Java-enabled devices such as cellphones, PDAs and pagers will grow from 176 million in 2001 to 721 million in 2005 [Takahashi 2001]. One of the reasons for this is that Java enables service providers to create new features very easily as it is based on the abstract Java Virtual Machine (JVM). Thus, it is currently portable to 80 to 95% of platforms and lets developers design and implement portable applications without the special tools and libraries that coding in C or C++ normally requires [Paulson 2001]. Second, Java has important security features for downloadable code [Knudsen 2001]. In addition, Java allows application writers to embed animation, sound, and other features within their applications easily, an important /advantage in web-based portable computing.

Running Java in an embedded/portable environment, however, is not without its problems. First, most portable devices have very small memory capacities. Consequently, the memory requirements of the virtual machine should be reduced and, accordingly, the application code should execute with a small footprint. Second, along with performance and form factor, energy consumption is an important optimization parameter in battery-operated systems. Since, traditionally, the virtual machine is tuned for performance [Smith et al. 1998], taking into account energy consumption requires reevaluation, and possibly redesign of the virtual machine from a new perspective. Third, the JVM in a portable environment is not as powerful as the JVM in a general-purpose system as many native classes are not supported. All these factors motivate us to tune specific components of the JVM (e.g., garbage collector, class loader) for a portable environment.

As embedded JVMs are designed to run for long periods of time on limited-memory embedded systems, creating and managing Java objects is of critical importance. The JVM supports automatic object reclamation, removing objects that are no longer referenced. Existing embedded JVMs such as Sun's KVM [KVM; Riggs et al. 2001] and HP's ChaiVM [chaivm] are already finely tuned to conform with three important requirements of embedded systems: soft real-time, limited memory, and long-duration sessions. However, currently, there is little support for analyzing and optimizing energy behavior of such systems. This is of critical importance for more widespread adoption of this technology in battery-constrained environments. In particular, the energy consumption in the memory system is a significant portion of overall energy expended in execution of a Java application [Vijaykrishnan et al. 2001]. Thus, it is important to consider techniques to optimize memory energy consumption. There are two important components of memory energy: dynamic energy and leakage energy. Dynamic energy is consumed whenever a memory array is referenced or precharged. Recent research has focused on the use of memory

banking and partial shutdown of the idle banks in order to reduce dynamic energy consumption [Delaluz et al. 2001; Lebeck et al. 2000]. However, leakage energy consumption is becoming an equally important portion as supply voltages and thus threshold voltages and gate oxide thicknesses continue to become smaller [Chandrakasan et al. 2001]. Researchers have started to investigate architectural support for reducing leakage in cache architectures [Kaxiras et al. 2001; Yang et al. 2001]. In this article, we show that it is possible to also reduce leakage energy in memory by shutting down idle banks using an integrated hardware–software strategy.

The garbage collector (GC) [Jones and Lins 1999] is an important part of the JVM and is responsible for automatic reclamation of heap-allocated storage after its last use by a Java application. Various aspects of the GC and heap subsystems can be configured at JVM runtime. This allows control over the amount of memory in the embedded device that is available to the JVM, the object allocation strategy, how often a GC cycle is triggered, and the type of GC invoked. We exploit the interaction of these tunable parameters along with a banked-memory organization to effectively reduce the memory energy (leakage and dynamic) consumption in an embedded Java environment. Since garbage collection is a heap-intensive (i.e., memory-intensive) operation and directly affects application performance, its impact on performance has been a popular research topic (e.g. see Jones and Lins [1999] and the references therein). In an embedded/portable environment, however, its impact on energy should also be taken into account. There are three questions we need to take into consideration when designing garbage collectors for energy sensitive systems:

—Since garbage collector itself consumes energy, how to reduce energy consumption during GC?
—Since garbage collector scans the memory, very detailed information about current memory usage can be obtained with a relatively small overhead right after each GC invocation. How can we make use of this information to reduce memory energy consumption?
—Some garbage collectors move objects to compact the heap. Is it possible to relocate objects during compaction phase to further enhance memory energy savings?

This article studies the energy impact of various aspects of a mark-and-sweep (M&S) garbage collector (commonly employed in current embedded JVM environments) in a multibank memory architecture. The experiments are carried out using two different (compacting and noncompacting) collectors in Sun's embedded JVM called KVM [Riggs et al. 2001; KVM]. Further, the virtual machine is augmented to include features that are customized for a banked-memory architecture. We also measure the sensitivity of energy behavior to different heap sizes, cache configurations, and number of banks. In order to investigate the energy behavior, we gathered a set of thirteen applications frequently used in hand-held and wireless devices. These applications include utilities such as calculator and scheduler, embedded web browser, and game

programs.[1] We observe that the energy consumption of an embedded Java application can be significantly more if the GC parameters are not tuned appropriately. Further, we notice that the object allocation pattern and the number of memory banks available in the underlying architecture are limiting factors on how effectively GC parameters can be used to optimize the memory energy consumption.

The remainder of this paper is organized as follows. The next section summarizes the K Virtual Machine and its GCs. Section 3 explains the experimental setup used for our simulations. Section 4 gives the energy profile of the current KVM implementation and discusses the impact of dividing memory into multiple banks. This section also investigates the energy impact of different features of our garbage collectors from both the hardware and software perspectives. Section 5 discusses related work. Finally, Section 6 concludes the article by summarizing our major contributions and giving an outline of planned future work.

2. KVM AND MARK-AND-SWEEP GARBAGE COLLECTOR

K Virtual Machine (KVM) [KVM; Riggs et al. 2001] is Sun's 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. Target devices for KVM technology include smart wireless phones, pagers, mainstream personal digital assistants, and small retail payment terminals. The KVM technology does not support Java Native Interface (JNI). The current implementation is interpreter-based and does not support JIT (Just-in-Time) compilation.

An M&S collector makes two passes over the heap. In the first pass (called 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. M&S collectors are widely used due to their ease of implementation and simple interface. As compared to other garbage collectors such as reference counting and generational collectors [Jones and Lins 1999], the M&S collector has both advantages and disadvantages. For example, no write-barrier overhead is necessary in M&S collectors while reference counting collectors rely on write-barrier mechanism to maintain reference counters. Similarly, generational collectors rely on write-barriers to keep track of intergenerational references. Further, in many real implementations of reference counting and generational collectors, M&S collectors are still used for resolving cyclic references and for collecting objects in older generations, respectively.

The KVM implements two M&S collectors, one without compaction and one with compaction [Riggs et al. 2001]. In the noncompacting collector, in the mark phase, all the objects pointed at by the root objects, or pointed at by objects that are pointed at by root objects are marked *live.* This is done by setting a bit in the object's header called MARK BIT. In the sweep phase, the object headers of all

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

objects in the heap are checked to see if the MARK BIT was set during the mark phase. All unmarked objects (MARK BIT = 0) are added to the free list and for the marked objects (MARK BIT = 1), the MARK BIT is reset. While allocating a new object, the free list is checked to see if there is a chunk of free memory with enough space to allocate the object. If there is not, then garbage collector is called. After garbage collection (mark and sweep phases), object allocation is tried again. If there is still not any space in the heap, an out-of-memory exception is thrown. Note that since this collector does not move objects in memory, the heap can easily get fragmented and the virtual machine may run out of memory quickly.

In an embedded environment, this heap fragmentation problem brings up two additional issues. First, since the memory capacity is very limited, we might incur frequent out-of-memory exceptions during execution. Second, a fragmented heap space means more active banks (at a given time frame) and, consequently, more energy consumption in memory. Both of these motivate for compacting live objects in the heap. Compacting heap space, however, consumes both execution cycles and extra energy which also need to be accounted for.

In the implementation of KVM, some objects (e.g., instances of java.lang. Class) and internal data structures (e.g., the memory blocks containing bytecodes of the applications' classes) are not allowed to move in the memory. These objects and internal data structures are called *permanent objects* because they remain alive till the application terminates. Permanent object do not require special treatment in the noncompacting mark-and-sweep collector since the collector does not move any objects in the heap. However, compacting mark-and-sweep collector may move objects to compact the heap, permanent objects should be distinguished from nonpermanent objects. In the compacting collector, a certain amount of space from the end of the heap is allocated for permanent objects and is called *permanent space*. The permanent space is not marked, swept, or compacted. The mark and sweep part of this collector is the same as the noncompacting collector. Compaction takes place on two occasions:

—after the mark and sweep phase if the size of the object to be allocated is still bigger than the largest free chunk of memory obtained after sweeping;

—when the first permanent object is allocated, and, as needed, when future permanent objects are allocated. Space for a permanent object is always allocated in steps of 2 KB. If the object needs more space, then another 2 KB chunk is allocated, and so on until its space requirement is satisfied.

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 sweep phase), it is checked whether the largest free chunk of memory (obtained after sweep phase) satisfies the size to be allocated. If not, then the collector enters compaction phase. After compaction, object allocation is attempted again. If there still is not any space, an out-of-memory exception is signaled.

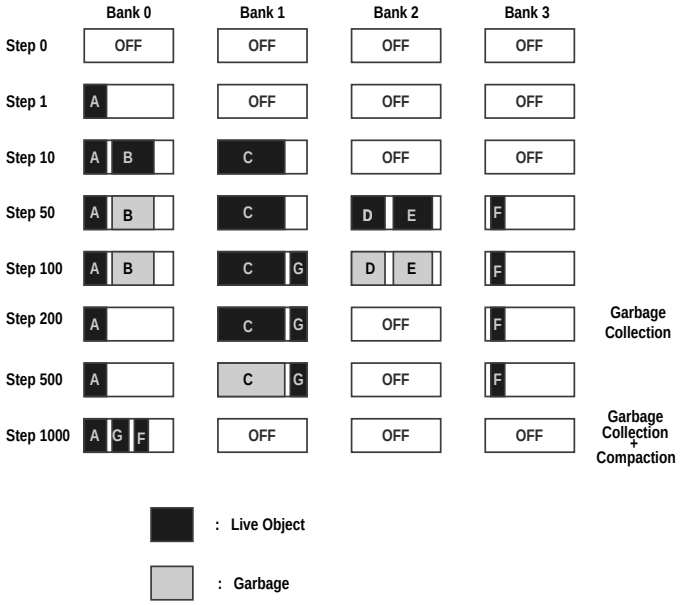

Fig. 1. Operation of garbage collector and compactor.

The default compaction algorithm in KVM is a Break Table-based algorithm. In this method, instead of using an additional field in the object's header to store the new address, a table containing relocation information is constructed in the free space. Thus, there is no extra space used to maintain the addresses. This table is called the Break Table. The live objects are moved to one end of the heap, and as they are moved, an entry is made in the Break Table consisting of two fields: (i) the old start address of the object and (ii) the total free space found until then. The Break Table may need to be shifted around if it gets in the way as live objects get compacted. If the Break Table rolls, it is sorted. After the objects are shifted, the pointers within the live objects are updated to point to the new address of the object. Advantages of this algorithm are that no extra space is needed to maintain the relocation information, objects of all sizes can be handled, and the order of object allocation is maintained. The disadvantage is that both sorting the break table and updating the pointers are costly operations both in terms of execution time and energy.

In the rest of the article, we will refer to these compacting and noncompacting collectors as M&S and M&C, respectively. It should be noted that both the collectors are not optimal in the sense that they do not reclaim an object immediately after the object becomes garbage (as an object is not officially garbage until it is detected to be so).

Figure 1 shows the operation of garbage collection and compaction in our banked memory architecture that contains four banks for the heap. Each step corresponds to a state of the heap after an object allocation and/or garbage collection/compaction. Step 0 corresponds to initial state where all banks are empty (turned off). In Step 1, object `A` is allocated and in Step 10, two more

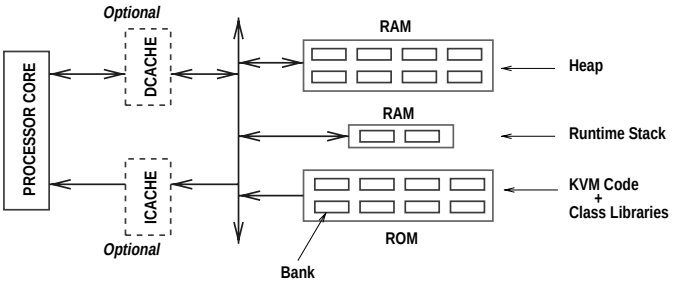

Fig. 2. Major components of our SoC. Note that cache memories are optional.

objects (B and C) are allocated. In Step 50, object B becomes garbage and three new objects (D, E, and F) are allocated. In Step 100, both D and E become garbage and G is allocated. Note that at this point all the banks are active despite the fact that Bank 2 holds only garbage. In Step 200, the garbage collector is run and objects B, D, and E are collected (and their space is returned to free space pool). Subsequently, since Bank 2 does not hold any live data, it can be turned off. In Step 500, object C in Bank 1 becomes garbage. Finally, Step 1000 illustrates what happens when both garbage collection and compaction are run. Object C is collected, live objects A, G, and F are clustered in Bank 0, and Banks 1 and 3 can be turned off. Two points should be emphasized. Energy is wasted in Bank 2 between steps 100 and 200 maintaining dead objects. Thus, the gap between the invocation of the garbage collection and the time at which the objects actually become garbage is critical in reducing wasted energy. Similarly, between steps 500 and 1000, energy is wasted in Banks 1 and 3 because the live objects that would fit in one bank are scattered in different banks. This case illustrates that compaction can bring additional energy benefits as compared to just invoking the garbage collector.

3. EXPERIMENTAL SETUP

3.1 Banked Memory Architecture

The target architecture we assume is a system-on-a-chip (SoC) as shown in Figure 2. The processor core of the system is based on the microSPARC-IIep embedded processor. This core is a 100 MHz, 32-bit five-stage pipelined RISC architecture that implements the SPARC architecture v8 specification. It is primarily targeted for low-cost uniprocessor applications. The target architecture also contains on-chip data and instruction caches that can be selectively enabled. Further, it contains an on-chip ROM and an on-chip SRAM. Figure 2 also shows both logical and physical views of the portion of the memory system of interest. This portion is divided into three logical parts: the KVM code and class libraries, the heap that contains objects and method areas, and the nonheap data that contains the runtime stack and KVM variables. Typically, the KVM code and the class libraries reside in a ROM. The ROM size we use is 128 KB for the storage of the actual virtual machine and libraries themselves [KVM]. 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 128 KB) holds both application bytecodes and application data, and is the target of our energy management strategies. An additional 32 KB of SRAM is used for storing the nonheap 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. 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.

3.2 Energy Models

For obtaining detailed energy profiles, we have customized an energy simulator and analyzer using the Shade [Cmelik and Keppel 1994] (SPARC instruction set simulator) tool-set and simulated the entire KVM executing a Java code. Shade is an instruction-set simulator and custom trace generator. Application programs are executed and traced under the control of a user-supplied trace analyzer. Current implementations run on SPARC systems and, to varying degrees, simulate the SPARC (Versions 8 and 9) and MIPS I instruction sets.

Our simulator tracks energy consumption in the processor core (datapath), on-chip caches, and the on-chip SRAM and ROM memories. The datapath energy is further broken into energy spent during execution and energy spent during GC. The GC energy, itself, is composed of energy spent in mark phase, sweep phase, and compaction phase (if used). Similarly, the memory 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. The 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.

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 our in-house cycle accurate energy simulator [Vijaykrishnan et al. 2000]. The simulator is configured to model a five-stage pipeline similar to that of the microSPARC-IIep architecture. The energies consumed by caches are evaluated using an analytical model that has been validated to be highly accurate (within 2.4% error) for conventional cache systems [Kamble and Ghose 1997]. All energy values reported in this article are based on parameters for

0.10 μm, 1 V technology. The dynamic energy consumption in the cache depends on the number of cache bitlines, wordlines, and the number of accesses. In this article, we model the SRAM-based memory using energy models similar to those used for caches. The number of banks and size of the banks in the SRAM-based memory are parameterizable.

In our model, a memory bank is assumed to be in *one of three modes* (*states*) at any given time. In the *read/write mode*, a read or write operation is being performed by the memory bank. In this mode, dynamic energy is consumed due to precharging the bitlines and also in sensing the data for a read operation. For a write operation, dynamic energy is consumed due to the voltage swing on the bitlines and in writing the cells. In the *active mode*, the bank is active (i.e., holds live data) but is *not* being read or written. In this mode, we consume dynamic precharge energy as there is no read or write into the bank. In addition, leakage energy is consumed in both these modes. Finally, in the *inactive mode*, the bank does not contain any live data. Thus, the bank is not precharged. Further, in this mode, we assume the use of a leakage control mechanism to reduce the leakage current. Thus, a bank in this mode consumes only a small amount of leakage energy and no dynamic energy.

In optimizing leakage current, we modify the voltage down converter circuit [Jou and Chen 1998] already present in current memory chip designs to provide a gated supply voltage to the memory bank. Whenever the *Sleep* signal is high, the supply to the memory bank is cut off, thereby essentially eliminating leakage in the memory bank. Otherwise, the *Gated* V_{DD} signal follows the input supply voltage (V_{DD}). The objective of our optimization strategy is to put as many banks (from the heap portion of memory) as possible into the inactive mode (so that their energy consumption can be optimized). This can be achieved by compacting the heap, colocating objects with temporal affinity, invoking the garbage collector more frequently, adopting bank-aware object allocation strategies, or a combination of these as will be studied in detail in Section 4. When a bank in the inactive mode is accessed to allocate a new object, it incurs a penalty of 350 cycles to service the request. The turn-on times from the inactive mode are dependent on the sizing of the driving transistors. Note that the application of this leakage control mechanism results in the data being lost. This does not pose a problem in our case as the leakage control is applied only to unused (inactive) banks.

The dynamic energy consumption for each of the modes is obtained by using scaled parameters for 0.10 μm technology from 0.18 μm technology files applying scaling factors from [Borkar 1999]. An analytical energy model similar to that proposed in [Kamble and Ghose 1997] is used, and a supply voltage of 1 V and a threshold voltage of 0.2 V are assumed. 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 future. Leakage becomes the dominant part of energy consumption for 0.10 micron (and below) technologies for the typical internal junction temperatures in a chip [Chandrakasan et al. 2001]. When our gated supply voltage scheme is applied, leakage energy is reduced to 3% of the original amount. This number is obtained through circuit simulation for 0.18 micron technology for a

Application	Brief Description	Footprint	Base Energy (mJ)
Calculator	Arithmetic calculator www.cse.psu.edu/~gchen/kvmgc/	18,024 14,279	0.68
Crypto	Light weight cryptography API in Java www.bouncycastle.org	89,748 60,613	8.40
Dragon	Game program comes with Sun's KVM	11,983 6,149	5.92
Elite	3D rendering engine for small devices home.rochester.rr.com/ohommes/Elite	20,284 11,908	3.67
Kshape	Electronic map on KVM www.jshape.com	39,684 37,466	13.52
Kvideo	KPG (MPEG for KVM) decoder www.jshape.com	31,996 14,012	1.52
Kwml	WML browser www.jshape.com	57,185 49,141	34.97
Manyballs	Game program comes with Sun's KVM	20,682 13,276	6.19
MathFP	Fixed-point integer math library routine home.rochester.rr.com/ohommes/MathFP	11,060 8,219	6.91
Mini	A configurable multi-threaded mini-benchmark www.cse.psu.edu/~gchen/kvmgc/	31,748 16,341	1.46
Missiles	Game program comes with Sun's KVM	26,855 17,999	4.28
Scheduler	Weekly/daily scheduler www.cse.psu.edu/~gchen/kvmgc/	19,736 17,685	9.63
Starcruiser	Game program comes with Sun's KVM	13,475 11,360	4.58

Fig. 3. Brief description of benchmarks used in our experiments. The two footprint values of each application are the maximal and effective footprint sizes (in bytes), respectively.

64-bit RAM when using the scheme explained above with driver sizing to maintain the same read time.

3.3 Benchmark Codes and Heap Footprints

In this study, we used thirteen applications ranging from utility programs used in hand-held devices to wireless web browser to game programs. These applications are briefly described in Figure 3. The first number in the third column of each application gives the maximum live footprint of the application; that is, the minimum heap size required to execute the application without an out-of-memory error if garbage is identified and collected immediately. The actual heap size required for executing these applications are much larger using the default garbage collection mechanism without compaction. For example, `Kwml` requires a minimum heap size of 128 KB to complete execution without compaction. The second number in the third column of each application in the figure gives the effective live heap size; that is, the average heap size occupied by live objects over the entire duration of the application's execution. A more detailed characterization of the live heap size over the entire application execution is shown in Figure 4. It should be noted that y-axis in these graphs represents the total size of live objects currently in the heap, not the actual memory usage of each application. Lack of variation in some graphs does not necessarily mean the memory usage of the application remains unchanged. Instead, it means that the objects of the application die as quickly as they are created. Actually, y-axis indicates the minimal memory requirement of each application, which

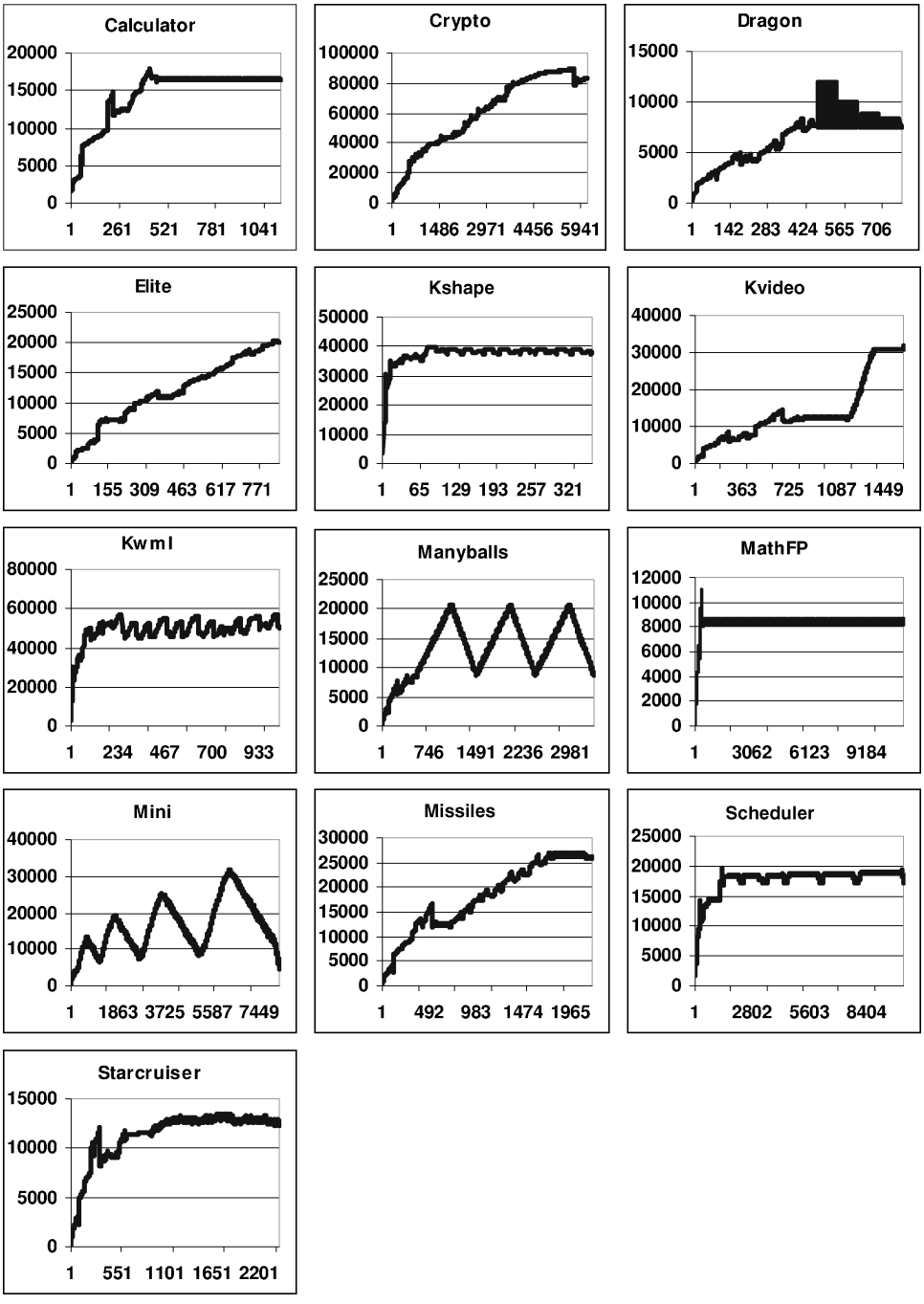

Fig. 4. Heap footprints (in bytes) of our applications. For each graph, x-axis denotes the time and y-axis gives the cumulative size of live objects.

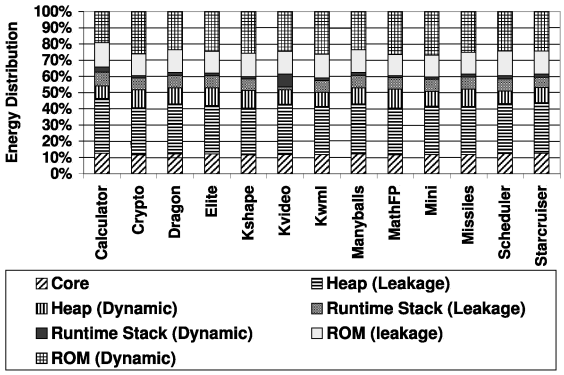

Fig. 5. Energy distribution.

determines the potential of shutting down portions of the heap memory. However, the ability to exploit this potential depends on various factors. These factors include the bank size, the garbage collection frequency, object allocation style, compaction style, and compaction frequency as will be discussed in the next section.

4. ENERGY CHARACTERIZATION AND OPTIMIZATION

4.1 Base Configuration

Unless otherwise stated, our default bank configuration has eight banks for the heap, eight banks for the ROM, and two banks for the runtime stack (as depicted in Figure 2). All banks are 16 KB. In this base configuration, by default, all banks are either in the active or read/write states, and *no* leakage control technique is applied. The overall energy consumption of this cacheless configuration running with M&S (GC without compaction) is given in the last column of Figure 3. The energy distribution of our applications is given in Figure 5. The contribution of the garbage collector to the overall datapath energy is 4% on average across the different benchmarks (not shown in the figure). We observe that the overall datapath energy is small compared to the memory energy consumption. We also observe that the heap energy constitutes 39.5% of the overall energy and 44.7% of the overall memory (RAM plus ROM) energy on the average.

Note that the memory energy consumption includes both the normal execution and garbage collection phases and is divided into leakage and dynamic energy components. On average, 75.6% of the heap energy is due to leakage. The leakage energy is dependent on the duration of the application execution while the dynamic energy is primarily determined by the number of references. Considering this energy distribution, reducing the heap energy through leakage control along with efficient garbage collection and object allocation can be expected to be very effective.

We also note from Figure 5 that overall ROM energy is less than the overall heap energy. This is mainly due to the following reasons. First, the dynamic energy for accessing a ROM is less than the corresponding value for a same

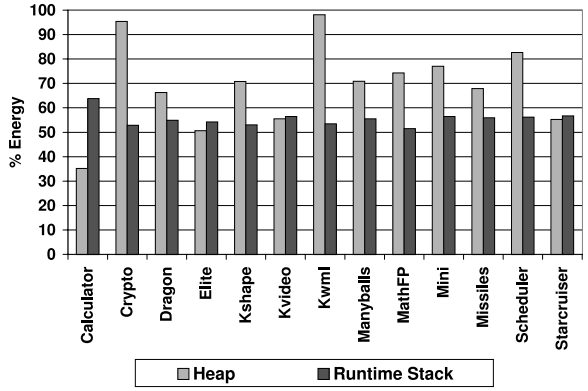

Fig. 6. Normalized energy consumption in heap and runtime stack due to mode control (M&S).

size RAM. This difference results from the smaller capacitive load on the wordlines and bitlines. In the ROM, only the memory cells that store a value of zero contribute a gate capacitance to the wordline. Further, only these cells contribute a drain capacitance to the bitline [Angel and Swartzlander 1997]. In addition, the number of bitlines is reduced by half with respect to the RAM configuration and a single-ended sense amplifier is used for the ROM array as opposed to a differential sense amplifier in the RAM array. Our circuit simulations show that the per access energy of a RAM array can thus be as large as 10 times that of a ROM array. However, the difference is dependent on the actual bit pattern stored in the array. In our experiments, we conservatively used a dynamic energy cost for accessing the ROM to be half that of a corresponding RAM array access. Since the effective transistor width in the ROM array is also smaller than that in a correspondingly sized RAM array, the leakage energy of the ROM is also smaller. Another reason that the ROM energy is less than the heap energy is because of using a ROM configuration that implements a simple but effective energy optimization. In particular, we use a banked ROM configuration and activate the supply voltage selectively to only those banks that contain libraries that are accessed by the application. Note that this incurs a penalty at runtime when the bank is accessed the first time. However, we found this overhead to be negligible.

Another interesting observation is the relative leakage and dynamic energy consumption breakdowns in the heap memory and the ROM. We found that the dynamic energy of the ROM is 63.7% of overall ROM energy which is much higher than the corresponding value in the heap. This difference is due to high access frequency of the ROM banks that contain the KVM code as well as class libraries.

4.2 Impact of Mode Control

Turning off a heap bank when it does not contain any live object can save energy in two ways. First, leakage energy is reduced as a result of the leakage reduction strategy explained earlier. Second, the precharge portion of dynamic energy is also eliminated when the bank is powered off. Figure 6 gives the

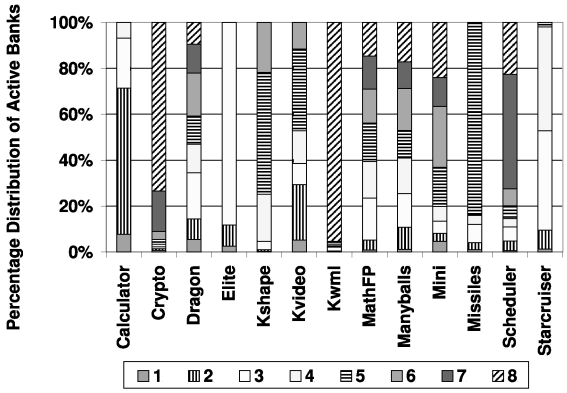

Fig. 7. Percentage distribution of active banks (M&S).

heap energy consumption due to M&S when mode control (leakage control) is employed, normalized with respect to the heap energy due to M&S when no mode control is used (i.e., all partitions are active all the time). We observe from this figure that turning off unused banks reduces the heap energy consumption by 31% on the average (with savings ranging from 2% to 65%). On average, 90% of these savings come from leakage energy reduction. Figure 7 explains the energy savings due to leakage control. This figure shows the percentage distribution of active banks for the applications in our suite. We observe that many applications execute with a small number of active banks most of the time, meaning that the remaining banks are turned off. We also observe, however, that some applications use all eight banks at some point during their executions. Considering this behavior and the heap footprints of live data shown in Figure 4, it can be clearly seen how badly live objects can be scattered throughout our 128 KB heap memory (although their cumulative sizes are much smaller than 128 KB).[2]

Figure 6 also shows the normalized runtime stack energy. This energy gain in runtime stack is achieved by not activating one of the banks of the runtime stack when it does not contain any useful data. Since we have two banks allocated to runtime stack (and the KVM variables) and many applications in our suite can operate most of the time with one bank only, on the average, we achieve around 50% energy saving on these banks.

These energy savings, however, do not come for free. As discussed earlier, accessing a powered off bank requires an extra 350 cycles for the supply voltage to be restored. During this time, a small amount of energy is also expended. Figure 8 shows the extra execution cycles and extra energy as both absolute values and percentages of overall execution time and memory energy, respectively. We can see from this figure that both of these overheads are negligible. Therefore, we can conclude that applying leakage control mechanism to the

[2]As an example, Figure 7 shows that, for more than 70% of the execution time of Kshape, five or six banks (16 KB each) are turned on. However, in Figure 4, we find that the total size of live objects of Kshape never exceeds 40 KB. Some banks cannot be turned off because they contain live objects, although the total size of live objects contained in this bank is much smaller than the bank size.

Application	Performance	Energy
Calculator	211,050 cycles (2.88%)	0.437 nJ (< 0.1%)
Crypto	80,850 cycles (0.10%)	0.167 nJ (< 0.1%)
Dragon	212,450 cycles (0.36%)	0.440 nJ (< 0.1%)
Elite	210,700 cycles (0.60%)	0.436 nJ (< 0.1%)
Kshape	215,600 cycles (0.17%)	0.446 nJ (< 0.1%)
Kvideo	211,750 cycles (1.40%)	0.438 nJ (< 0.1%)
Kwml	253,050 cycles (0.07%)	0.524 nJ (< 0.1%)
Manyballs	245,350 cycles (0.39%)	0.508 nJ (< 0.1%)
MathFP	81,200 cycles (0.12%)	0.168 nJ (< 0.1%)
Mini	22,050 cycles (0.16%)	0.045 nJ (< 0.1%)
Missiles	248,850 cycles (0.59%)	0.515 nJ (< 0.1%)
Scheduler	213,850 cycles (0.22%)	0.443 nJ (< 0.1%)
Starcruiser	250,250 cycles (0.55%)	0.518 nJ (< 0.1%)

Fig. 8. Energy and performance overhead of bank turning-off.

inactive heap banks can reduce energy consumption significantly without too much impact on execution time.

4.3 Impact of Garbage Collection Frequency

The M&S collector is called by default when, during allocation, the available free heap space is not sufficient to accommodate the object to be allocated. It should be noted that between the time that an object becomes garbage and the time it is detected to be so, the object will consume heap energy as a dead object. Obviously, the larger the difference between these two times, the higher the wasted energy consumption if collecting would lead to powering off the bank. It is thus vital from the energy perspective to detect and collect garbage as soon as possible. However, the potential savings should be balanced with the additional overhead required to collect the dead objects earlier (i.e., the energy cost of garbage collection).

In this subsection, we investigate the impact of calling the garbage collector (without compaction) with different frequencies. Specifically, we study the influence of a *k-allocation collector* that calls the GC once after every k object allocations. We experimented with five different values of k: 10, 40, 75, 100, and 250. The top graph in Figure 9 illustrates the heap energy (normalized with respect to M&S heap energy without mode control) of the k-allocation collector. The impact of pure mode control is reproduced here for comparison.

We clearly observe that different applications work best with different garbage collection frequencies. For example, the objects created by Dragon spread over the entire heap space very quickly. However, the cumulative size of live objects of this benchmark most of the time is much less than the available heap space. Consequently, calling the GC very frequently (after every 10 object allocations) transitions several banks into the inactive state and reduces heap energy by more than 40%. Reducing the frequency of the GC calls leads to more wasted energy consumption for this application. In Kvideo, we observe a different behavior. First, the energy consumption is reduced by reducing the frequency of collector calls. This is because each garbage collection has an energy

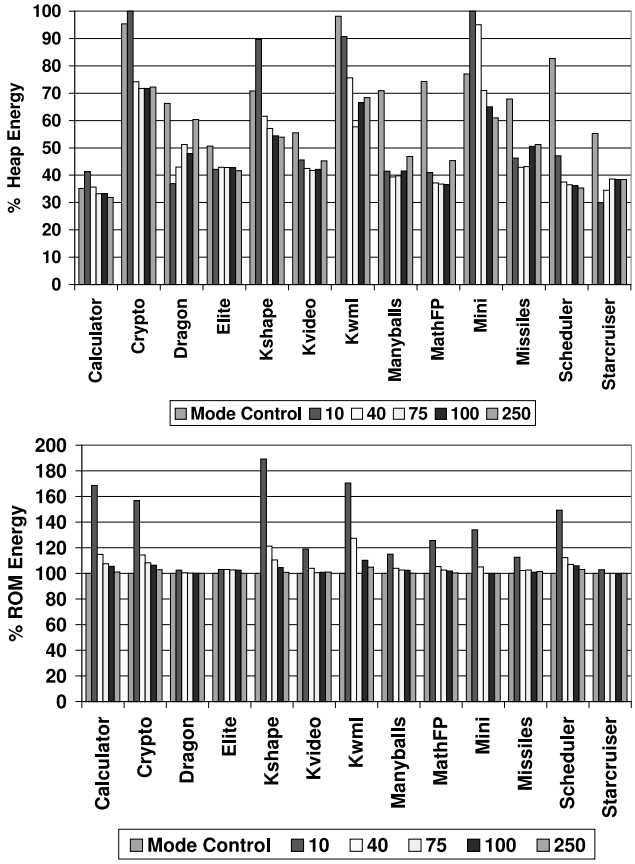

Fig. 9. Normalized energy consumption in heap and ROM memory when M&S with mode control is used with different garbage collection frequencies.

cost due to fact that mark and sweep operations access memory. In this application, the overhead of calling GC in every 10 allocations brings an energy overhead that cannot be compensated for by the energy saving during execution. Therefore, calling the GC less frequently generates a better result. Beyond a point ($k = 75$), however, the energy starts to increase as the garbage collections become so less frequent that significant energy is consumed due to dead but not collected objects. Applications such as `Mini`, on the other hand, suffer greatly from the GC overhead and would perform best with much less frequent garbage collector calls. Overall, it is important to tune the garbage collection frequency based on the rate at which objects become garbage to optimize energy consumption.

The GC overhead also leads to increased energy consumption in the ROM, runtime stack, and processor core. The energy increase in the ROM is illustrated on the bottom graph of Figure 9. Each bar in this graph represents the energy consumption in the ROM normalized with respect to the energy consumption of the ROM with M&S with mode control. It can be observed that the case with

$k = 10$ increases the energy consumption in ROM significantly for many of the benchmarks. On the other hand, working with values of k such as 75, 100, and 250 seems to result in only marginal increases, and should be the choice, in particular, if they lead to large reductions in heap energy. We also found that the energy overheads in the core and runtime stack were negligible and have less than 1% impact on overall energy excluding cases of $k = 10$. To summarize, determining globally optimal frequency demands a trade-off analysis between energy saving in the heap and energy loss in the ROM. Except for cases when $k = 10$, the energy savings in the heap clearly dominate any overheads in the rest of the system.

A major conclusion from the discussion above is the following. Normally, a virtual machine uses garbage collector only when it is necessary, as the purpose of garbage collection is to create more free space in the heap. In an energy-sensitive, banked-memory architecture, on the other hand, it might be a good idea to invoke the collector even if the memory space is not a concern. This is because calling GC more frequently allows us to detect garbage *earlier*, and free associated space (and turn off the bank). This early detection and space deallocation might result in large number of banks being transitioned to the inactive state.

4.4 Impact of Object Allocation Style

M&S in KVM uses a global free list to keep track of the free space in the heap. When an object allocation is requested, this free list is checked, the first free chunk that can accommodate the object is allocated, and the free list is updated. While in a nonbanked architecture, this is a very reasonable object allocation policy, in a banked-memory based system it might be possible to have better strategies. This is because the default strategy does not care whether the free chunk chosen for allocation is from an already used (active) bank or inactive bank. It is easy to see that everything else being equal, it is better to allocate new objects from already active banks.

To experiment with such a strategy, we implemented a new bank allocation method where each bank has its own private free list. In an object allocation request, first, the free lists of active banks are checked and, only if it is not possible to allocate the space for the object from one of these lists, the lists of inactive banks are tried. This strategy is called the *active-bank-first allocation*.

Figure 10 gives the energy consumption for three different versions. M&S with leakage control (denoted Mode Control), active-bank-first allocation (denoted Active Bank), and a version that combines active-bank-first allocation with a strategy that activates the GC only when the new object cannot be allocated from an already active bank (denoted Active Bank+). All values in this figure are normalized with respect to the heap energy consumption of M&S without mode control. We see from these results that Active Bank does not bring much benefit over Mode Control in most cases (except that we observe a 6% heap energy improvement in `MathFP`).

This can be explained as follows. Objects with long lifetime are typically allocated early (before the first GC is invoked) and occupy the first few banks.

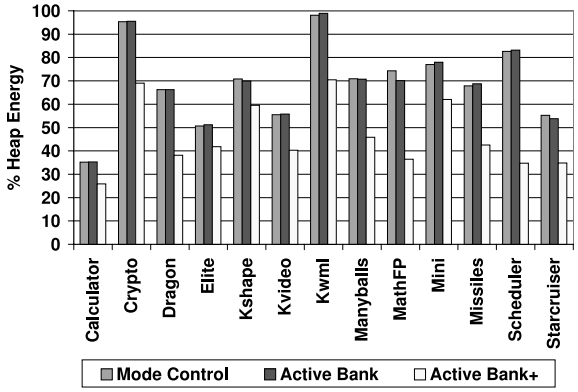

Fig. 10. Normalized energy consumption in heap (active bank allocation versus default allocation).

The younger objects that occupy banks with higher addresses seldom survive the next garbage collection. From the traces of bank occupation, we observe that after each GC, the banks with lower address are always occupied and the higher addresses are typically free. Consequently, the default allocation acts like active-bank-first allocation. `MathFP` is an exception to this allocation behavior. In `MathFP`, after each GC, the occupied banks are not always contiguous. In this case, active-bank-first allocation can save energy by postponing the turning on a new bank. In contrast, in benchmarks such as `Kwml` and `Scheduler`, the energy overhead of maintaining multiple free lists shows up as there is almost no gain due to the allocation strategy itself.

Thus, it is important to modify the default garbage collection triggering mechanism in addition to changing allocation policy to obtain any benefits. Active Bank+ combines the active-bank-first allocation mechanism along with a strategy that tries to prevent a new bank from being turned on due to allocation. As it combines an energy aware allocation and collection policy, Active Bank+ can lead to significant energy savings as shown in Figure 10. The causes for these savings are three fold. First, Active Bank+ invokes the GC more frequently, and thus banks without live objects are identified and turned off early. Second, during allocation, it reduces the chances of turning on a new bank. Third, it colocates permanent objects more densley, thereby increasing the opportunities of turning off banks.

4.5 Impact of Compaction

As explained earlier in the article, the compaction algorithm in KVM performs compaction only when, after a GC, there is still no space for allocating the object. In a resource-constrained, energy-sensitive environment, compaction can be beneficial in two ways. First, it might lead to further energy savings over a noncompacting GC if it can enable turning off a memory bank that could not be turned off by the noncompacting GC. This may happen as compaction tends to cluster live objects in a smaller number of banks. Second, in some cases, compaction can allow an application to run to completion

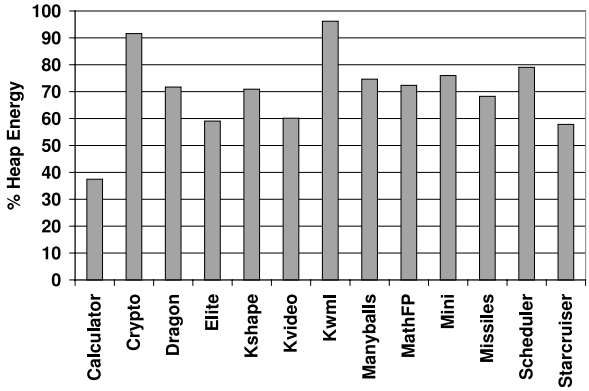

Fig. 11. Energy consumption in heap due to mode control (M&C) normalized with respect to M&C without mode control.

(without out-of-memory error) while the non-compacting algorithm gives an out-of-memory error. In this subsection, we study both these issues using our applications.

Let us first evaluate the energy benefits of mode control when M&C (the default compacting collector in KVM) is used. The results given in Figure 11 indicate that mode control is very beneficial from the heap energy viewpoint when M&C is employed. Specifically, the heap energy of the M&C collector is reduced by 29.6% over the M&C without mode control. The top graph in Figure 12 compares heap energy of M&S and M&C with mode control. Each bar in this graph represents heap energy consumption normalized with respect to M&S without mode control. It can be observed that M&C does not bring significant savings over M&S (denoted Mode Control in the graph). First, moving objects during compaction and updating reference fields in each object consumes energy. In addition, compacting may increase the applications running time, which also means more leakage energy consumption. Therefore, a trade-off exists when compaction is used. In our implementation, to lessen the performance impact, compaction is performed only when the object to be allocated is larger than any of the available free chunks, or if it can turn off more banks. `Kwml` is one of the benchmarks where compaction brings some energy benefits over M&S with mode control. The execution trace of this code indicates that there are many scenarios where Mode Control does not turn off banks because all banks contain some small-sized permanent objects. M&C, on the other hand, turns off some banks after garbage collection due to the fact that it both compacts fragmented live objects with short lifetimes and clusters permanent objects in a smaller number of banks. In some benchmarks such as `Dragon`, on the other hand, M&C does not create sufficient number of free banks to offset the extra energy overhead due to additional data structures maintained.

The original allocation policy in the compacting version distinguishes between permanent and dynamic objects as mentioned earlier. In the banked-memory architecture, the default allocation policy is slightly modified to allocate the permanent objects and regular objects in separate banks. This eliminates

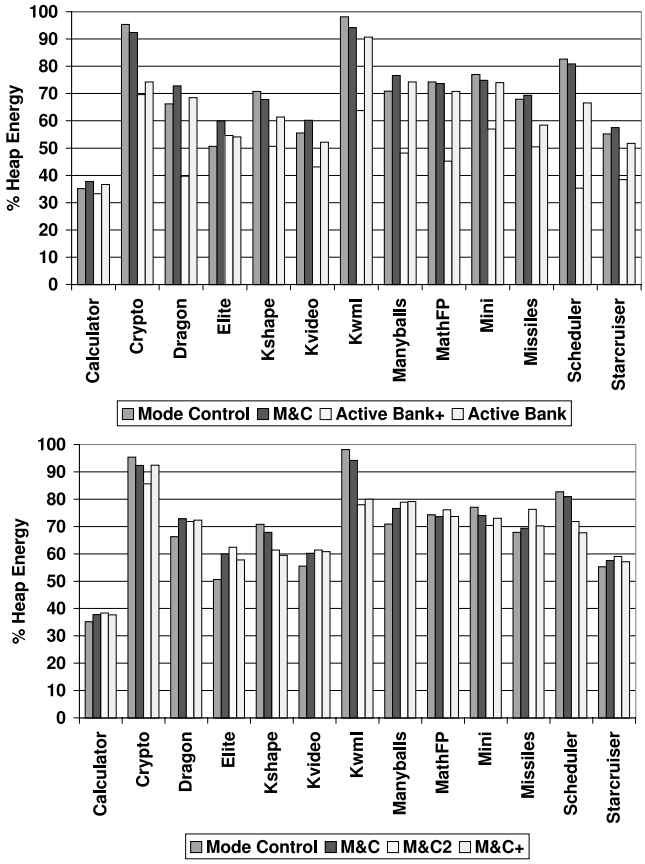

Fig. 12. Top: Comparison of M&C and M&S. Bottom: Comparison of different compacting collectors.

the need to move the already allocated dynamic objects when a new permanent object is allocated. However, this strategy requires activating at least two banks when both permanent and dynamic objects are present. The active-bank-first allocation strategy, on the other hand, colocates both the permanent and dynamic objects together and saves energy. However, it incurs the cost of moving the already allocated dynamic objects to a new bank when a new permanent object is allocated. Fortunately, this operation is very infrequent. Consequently, as opposed to the case without compaction, the Active Bank version (that is, allocating object from an already active bank if it is possible to do so) combined with M&C generates better results than M&C with default allocation, and consumes 10% less heap energy on the average. That is, compacting the heap improves the energy impact of the active-bank-first allocation strategy. Finally, as before, the Active Bank+ outperforms other versions for most of the cases.

The bottom graph in Figure 12 compares heap energy consumption of three different compaction algorithms. M&C is the default compactor in KVM. The M&C+ version differs from M&C in that it performs compaction after each garbage collection (whether or not it is actually needed from the viewpoint

of free space). Our results show that in some benchmarks such as `Kshape` and `Scheduler`, it generates better results than both M&S (denoted Mode Control in the figure) and M&C. This result means that unlike general-purpose systems, in an energy-sensitive system, extra compactions might bring energy benefits for some applications. M&C2, on the other hand, is a collector that uses the Lisp2 Algorithm, as opposed to the default break table-based algorithm in KVM. In the Lisp2 algorithm, during compaction, first, the new addresses for all objects that are live are computed. The new address of a particular object is computed as the sum of the sizes of all the live objects encountered until this one, and is then stored in an additional "forward" field in the object's header. Next, all pointers within live objects that refer to other live objects are updated by referring to the "forward" field of the object they point to. Finally, the objects are moved to the addresses specified in the "forward" field, and then the "forward" field is cleared so that it can be used for the next garbage collection. The advantages of this algorithm are that it can handle objects of varying sizes, it maintains the order in which objects were allocated, and it is a fast algorithm with an asymptotic complexity of $O(M)$, where M is the heap size. Its disadvantage is that it requires an additional four-byte pointer field in each object's header that increases the heap footprint of the application.

There are two potential energy benefits due to this compaction style. First, objects can be relocated accounting for temporal affinities and object lifetimes, instead of sliding-only compaction as in M&C. For example, clustering objects with similar lifetime patterns increases the potential for deactivating an entire bank (when the objects it holds die together). Secondly, reference fields can be updated more efficiently as compared to M&C and M&C+, where updating each reference field needs to look up the Break Table. Finally, the extra forward field can be used as a stack in the marking phase to reduce the overhead during the scanning phase.

In case that the heap is severely fragmented, M&C2 will out perform M&C+ because it treats each object individually, and does not need to copy the Break Table (in this case, the Break Table will be large) when moving objects. On the other hand, when most live objects are placed contiguously, M&C+ will perform better because it can move objects in fewer chunks. Further, the smaller Break Table reduces the look up cost (whose time complexity increases logarithmically with respect to the Break Table size) when updating each reference field during compaction. Obviously, if the total number of reference fields is large, M&C+'s performance will suffer a lot during the updating phase.

`Crypto` is an example application with rather big heap footprint that benefits from M&C2's cheaper marking and updating. In contrast, `Elite` is an application with very small footprint. Due to the 4-byte's overhead in each objects, M&C2 turns on a new bank much earlier than M&C+. Specifically, M&C2 turns on the third bank about 5.6 seconds after program initialization while the corresponding value for M&C+ is 6.2 seconds. Initializing the forwarding fields of each objects also consumes some extra energy.

As we mentioned earlier, a compacting GC can run an application in a smaller heap memory than a corresponding noncompacting version. For example, `Missiles` can run using a 32 KB heap when M&C is employed while

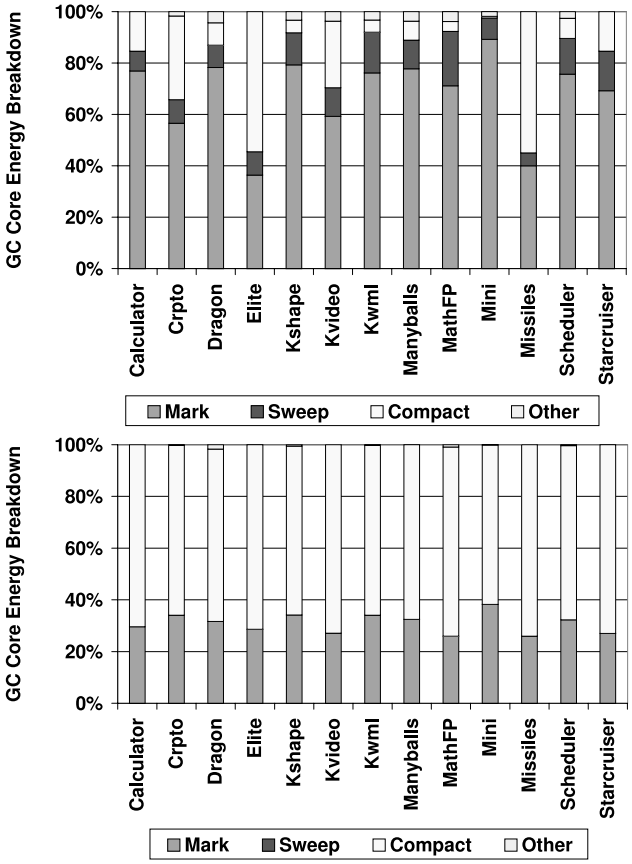

Fig. 13. Energy distribution in core due to GC. Top: M&C; Bottom: M&C2.

requiring a minimum of 64 KB heap when executing using M&S. Comparing the energy consumption for systems with these configurations, we found that the M&S that uses a 64 KB heap with four 16 KB-banks consumes a heap energy of 1.02 mJ, which is much larger than 0.71 mJ, the heap energy consumed by M&C2 when using a 32 KB heap using two 16 KB-banks. Similarly, `Kwml` can run using a 64 KB heap when M&C is employed, while requiring a minimum of 128 KB heap when executing using M&S. For this application, the M&S that uses a 128 KB heap with eight 16 KB-banks consumes a heap energy of 13.15 mJ, which is much larger than 7.66 mJ, the heap energy consumed by M&C2 when using a 64 KB heap using four 16 KB-banks.

It is also interesting to study how much energy the compaction itself contributes relative to the other portions of the collector. Figure 13 shows the core energy breakdown due to the garbage collection activity with M&C and M&C2. Both M&C and M&C2 have four major phases. For M&C the phases are mark, sweep, relocate, and update. We combine the last two into a part called "compaction" as they are invoked only during compaction. For M&C2, the phases are mark and the three phases associated with compaction: compute,

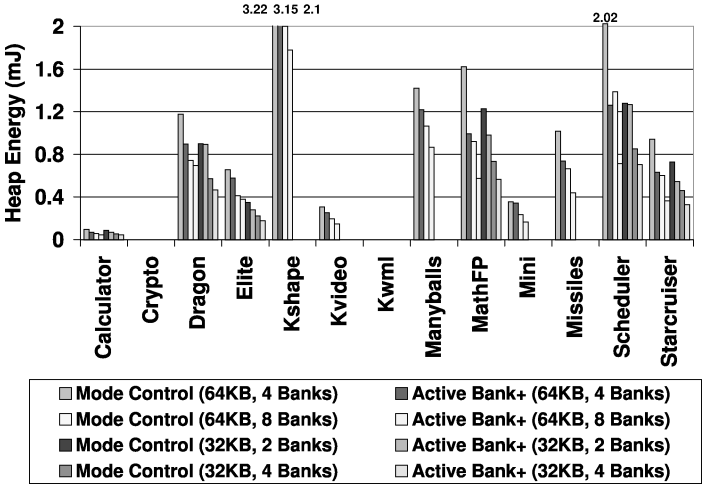

Fig. 14. Impact of number of banks and heap size (M&S). Note that `Crypto` and `Kwml` do not run with 32 KB and 64 KB heap sizes. `Kvideo`, `Manyballs`, `Kshape`, `Missiles`, and `Mini` do not run with 32 KB heap size.

update, and relocate. We see that in M&C that the mark phase consumes the bulk of the energy, mainly, because it is more expensive than the sweep operation. The contribution of compact energy varies from application to application depending on the number of times the compaction is invoked. When we consider M&C2, however, the energy behavior changes. First, since there is no explicit sweep activity, the energy consumption is distributed mainly between compact and mark phases. Second, since this collector performs compaction every time GC is called (as opposed to M&C that performs compaction only when an object still cannot be allocated after GC), the compaction energy constitutes a larger portion in most of the benchmarks.

4.6 Impact of Number of Banks and Heap Size

The heap size and bank size can influence the effectiveness of mode control. Let us consider the example in Figure 1 once again to illustrate the influence of bank size. Instead of four banks, if we had only two larger banks (Bank 0 + Bank1 and Bank 2 + Bank 3), at step 200, the garbage collector would not be able to turn off any of the banks. Similarly, the heap size would also influence the frequency of GC invocations in the default version. For example, if the heap size is reduced by half in Figure 1 (i.e., only Banks 0 and 1 are available), the garbage collector will be invoked at step 50 to find space to accommodate object `D`. Further reducing the heap size will also reduce overall leakage energy as we have fewer leaking transistors. Thus, it is important to evaluate the energy impact of varying these parameters.

Figure 14 shows the impact of varying the bank and heap sizes when using the M&S garbage collector with Mode Control and Active Bank+. It must be noted that many applications cannot complete with a smaller heap size. Only six of the applications can execute using both 64 KB and 32 KB heap sizes.

In general, reducing the heap size reduces the overall energy consumption. There are two reasons for this behavior. With a smaller heap, the effort expended in allocating an object and also marking and sweeping the heap during garbage collection reduce. It must be reiterated that the M&S algorithm uses a nonstack implementation which means that the cost of garbage collection is proportional to the size of the heap. However, a smaller heap will also increase the frequency of garbage collection making the overhead of garbage collection more significant. As an example, the number of collections increase from 2 to 20, when heap size is reduced from 128 KB to 32 KB when executing `Scheduler`. For most cases, this overhead is more than compensated for by the energy savings due to more frequent garbage collection. In other words, the dead objects are collected closer to when they become dead.

Secondly, when we increase the number of banks, mode control has the ability to exploit a finer granularity of turn-off. In addition, a smaller bank has a smaller capacitive load and hence a smaller per access dynamic energy cost. This leads to smaller heap energy consumption when using smaller banks. On the average, for a 64 KB heap, when using 8 KB banks the energy consumption is only 65% of the energy consumed when using 16 KB banks. Similar trends are observed for a 32 KB heap. We also observe that the Active Bank+ scheme brings energy benefits over simple mode control across all configurations (around 20%, on the average, across all benchmarks and configurations). We also evaluated the impact of bank and heap sizes on other garbage collection algorithms. We found similar trends in their behavior.

While smaller banks are found to be beneficial, the overheads of banking make them unattractive at very small granularities. A more detailed characterization of the influence of number of banks for different size memory arrays can be found in [Wilton and Jouppi 1994]. In order to achieve additional savings, it might be important to exploit finer granular turn-offs such as those at the word level [Kaxiras et al. 2001] instead of the bank level.

4.7 Impact of Cache Memory

The presence of a cache influences the energy behavior in two ways. First, the number of references to the memory, both the ROM and RAM, are reduced. This reduces the dynamic energy consumed in the memory. In particular, we find that the heap energy reduces to 23% of the overall energy in the presence of the 4 KB data and 4 KB instruction caches. Note that embedded cores typically have small caches. Second, the cache can account for a significant portion of the overall system energy. In particular, the instruction cache is a major contributer as it is accessed every cycle. In the context of this work, it is important to evaluate how the cache influences the effectiveness of mode control strategy and the additional gains that energy-aware allocation and collection provide over pure mode control. Figure 15 shows the normalized heap energy in the presence of a 4 KB 2-way instruction cache and a 4 KB 2-way data cache when a 64 KB heap is used. Pure mode control with M&S reduces 15% of heap energy on the average across all benchmarks. An additional 28% heap energy saving is obtained through the energy-aware active bank allocation and garbage

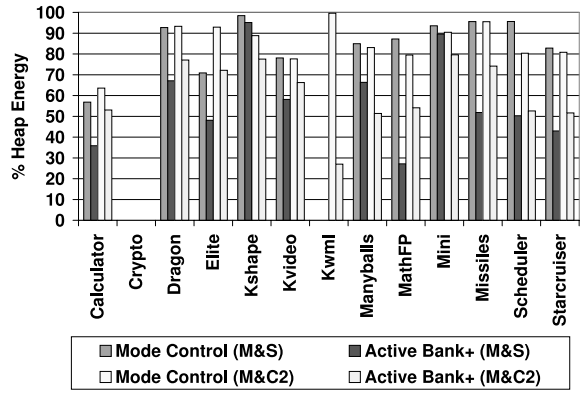

Fig. 15. Impact of cache memory. All numbers are normalized with respect to the heap energy consumed using the same configuration with no mode control. Note that `Crypto` does not run with 64 KB heap. Also, `Kwml` cannot complete using the M&S GC.

collection before new bank activation. The corresponding figures when M&C2 is used are 14% and 25%, respectively. These results show that the proposed strategies are effective even in the presence of a cache.

5. RELATED WORK

Automatic garbage collection has been an active research area for the last two decades. The current approaches to garbage collection focus on locality-aware garbage collection (e.g., D. Grunwald and Henderson [1993] and Chilimbi and Larus [1998]), concurrent and hardware-assisted garbage collection (e.g. Heil and Smith [2000]), and garbage collection for Java (e.g. Agesen et al. [1998]) among others. A comprehensive discussion of different garbage collection mechanisms can be found in [Jones and Lins 1999]. All these techniques are geared towards improving performance rather than energy consumption. We showed in this article that for an energy-aware collection, different GC parameters should be tuned. Diwan et al. analyzed four different memory management policies from the performance as well as energy perspectives. Our work differs from theirs in that we focus on a banked-memory architecture, and try to characterize and optimize energy impact of different garbage collection strategies when a leakage control mechanism is employed.

As Java is becoming a popular programming language for both high-end and low-end systems, researchers are focusing on different aspects of Java-based systems, including Just-in-Time compilation [Cierniak et al. 2000; Lee et al. 2000; Yang et al. 1999], garbage collection [Agesen et al. 1998; Stichnoth et al. 1999], heap allocation behavior of Java codes [Dieckmann and Holzle 1999], and synchronization optimization. Most of the Java-specific optimizations proposed so far focus on improving performance whereas we target improving energy consumption without unduly increasing execution time.

Recently, energy optimization has become a topic of interest in the software community. Catthoor et al. [1998], Kandemir et al. [2000], and Vijaykrishnan et al. [2000] show that program transformations can be very effective in reducing memory energy of array-dominated embedded applications. Lebeck et al.

[2000] and Delaluz et al. [2001] specifically focused on banked-memory architectures and suggested, respectively, operating system based and compiler/hardware based optimization strategies for reducing dynamic power. Our work differs from these in that we specifically target embedded Java environments and focus mainly on exploiting leakage control mechanisms for reducing energy. We also illustrate how garbage collector can be tuned to maximize the effectiveness of leakage control. Flinn et al. [2000] quantifies the energy consumption of a pocket computer when running Java virtual machine. In [Vijaykrishnan et al. 2001], the energy behavior of a high-performance Java virtual machine is characterized. In contrast to these, our work targets a banked-memory architecture and tunes garbage collector for energy optimization. Finally, numerous papers attempt to optimize energy consumption at the circuit and architectural levels. In particular, the leakage optimization circuit employed here tries to reduce leakage current and is similar to that used in [Kaxiras et al. 2001; Yang et al. 2001]. We employ a design that is a simple enhancement of existing voltage down converters present in current memory designs. Further, the circuit with the differential feedback stage helps to respond to load variations faster during normal operation.

6. CONCLUSIONS AND FUTURE WORK

As battery-operated Java-enabled devices continue to grow, it is becoming important to design resource-constrained Java virtual machines. Simply porting a desktop JVM to run on an embedded device can produce a large fixed memory overhead and result in a large energy consumption that are unacceptable in most embedded products. Therefore, it is important to design virtual machines components afresh for embedded environments. In embedded environments, memory leaks combined with the limited memory capacity can be potentially crippling. Thus, garbage collection that automatically reclaims dead objects is a critical component. In this work, we characterized the energy impact of GC parameters built on top of Sun's embedded Java virtual machine, KVM. Further, we showed how the GC can be tuned to exploit banked memory architectures for saving energy. The major conclusions from this work are as follows:

— Our characterization of energy consumption shows that the heap energy consumption is 39.5% of the overall energy consumption of an embedded system-on-a-chip when executing typical Java applications. Further, we observe that the leakage energy is the dominant portion, accounting for 75.6% of the heap energy.
— In an energy-constrained environment, the GC can be used to identify unused heap memory banks and apply energy control mechanisms. Our results show that GC-controlled energy mode control can save 31% of the heap energy on the average.
— The duration between when an object becomes dead and when it is garbage collected determines the wasted leakage energy in maintaining these dead objects. Thus, the frequency of garbage collection has a profound impact on how much of this wasted energy can be reduced. The more frequent the

garbage collection, the less the wasted energy. Thus, in a banked-memory environment, it will be beneficial from an energy perspective to invoke garbage collection even before the traditional invocation time (when space cannot be found for allocating an object). However, garbage collection itself incurs an energy cost that must be considered.

— The energy savings of GC-controlled energy optimization are influenced by both the object allocation and garbage collection policies. In particular, we find that a strategy that allocates objects only on active banks (if possible) and activates garbage collection before turning on a new bank provides consistent improvements over pure mode control.

— Clustering live objects in small number of banks using compaction can reduce heap energy. While some applications benefit from this clustering, the energy overhead of moving the live objects during compaction negates the potential benefits in others. As in the case of garbage collection frequency, compacting more often than when only running out of heap space to allocate an object provides energy savings in some benchmarks. The compaction style also influences the overhead and overall effectiveness of compaction. Specifically, taking object reference relations into account (M&C2) improves the energy impact of compaction in some cases.

— The proposed GC-controlled energy management is effective across different heap, bank, and cache configurations. We observe that while decreasing heap size can prevent some applications from running to completion, it generally reduces the overall heap energy consumption. In addition, our experiments show that smaller bank sizes result in less energy consumption due to reduced capacitive load when accessing the banks and the increased potential for finer granular leakage control. Finally, when caches are enabled, the GC-controlled energy management is still shown to be effective.

This work opens up many interesting aspects of tuning the memory allocation and management features in battery-operated environments. First, it would be interesting to design techniques that identify dead objects as soon as possible. Along this direction, variants of reference counting mechanisms can be reinvestigated from an energy perspective. It will be interesting to balance the reduction in wasted energy (during the time between the object becomes garbage and the time it is detected to be so) with the additional energy overheads to implement such a scheme. Second, we plan to investigate collectors that may require larger heap sizes but can exploit memory bank turn-off more efficiently. Towards this, we plan to study copying collectors that combine the phases of collection and compaction. Finally, we plan to adopt more sophisticated leakage control mechanisms that can maintain the actual data when leakage control mechanism is in use.

REFERENCES

AGESEN, O., DETLEFS., D., AND MOSS, J. E. B. 1998. Garbage collection and local variable type-precision and liveness in java virtual machines. In *Proceedings of the SIGPLAN'98 ACM Conference on Programming Languages and Implementation* (Montreal, Canada), 269–279.

ANGEL, E. D. AND SWARTZLANDER, E. E. 1997. Survey of low-power techniques for roms. In *Proceedings of International Symposium on Low Power Electronics and Design*. 7–11.

BORKAR, S. 1999. Design challenges of technology scaling. *IEEE Micro 19*, 4 (Jul/Aug.), 23–27.

CATTHOOR, F., WUYTACK, S., GREEF, E. D., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998. *Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, Deventer.

CHAIVM. Chaivm for jornado. http://www.hp.com/products1/embedded/jornado/index.html.

CHANDRAKASAN, A., BOWHILL, W. J., AND FOX, F. 2001. *Design of High-Performance Microprocessor Circuits*. IEEE Press, New York.

CHILIMBI, T. M. AND LARUS, J. R. 1998. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of 177 International Symposium on Memory Management*. 17–19.

CIERNIAK, M., LUEH, G.-Y., AND STICHNOTH, J. M. 2000. Practicing judo: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Vancouver B.C., Canada).

CMELIK, B. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*. 128–137.

D. GRUNWALD, B. Z. AND HENDERSON, R. 1993. Improving the cache locality of memory allocation. In *Proceedings of ACM Conference on Programming Languages and Implementation*. 177–186.

DELALUZ, V., KANDEMIR, M., VIJAYKRISHNAN, N., SIVASUBRAMANIAM, A., AND IRWIN, M. J. 2001. Dram energy management using software and hardware directed power mode control. In *Proceedings of the 7th International Conference on High Performance Computer Architecture* (Monterrey, Mexico).

DIECKMANN, S. AND HOLZLE, U. 1999. A study of the allocation behavior of the specjvm98 java benchmarks. In *Proceedings of European Conference on Object-Oriented Programming*.

DIWAN, A., H. LI, D. G., AND FARKAS, K. Energy consumption and garbage collection in low powered computing. http://www.cs.colorado.edu/ ~diwan University of Colorado-Boulder.

FLINN, J., BACK, G., ANDERSON, J., FARKAS, K., AND GRUNWALD, D. 2000. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems*. 252–263.

HEIL, T. AND SMITH, J. E. 2000. Concurrent garbage collection using hardware assisted profiling. In *Proceedings of International Symposium on Memory Management*.

JONES, R. AND LINS, R. D. 1999. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York.

JOU, S. AND CHEN, T. 1998. On-chip voltage down converter for low-power digital systems. *IEEE Trans. Circuits Systems-II: Analog and Digital Signal Processing 5* (May), 617–625.

KAMBLE, M. AND GHOSE, K. 1997. Analytical energy dissipation models for low power caches. In *Proceedings of International Symposium on Low Power Electronics and Design*. 143.

KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M. J., AND YE, W. 2000. Influence of compiler optimizations on system power. In *Proceedings of the Design Automation Conference* (Los Angeles, California USA).

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture*.

KNUDSEN, J. 2001. *Wireless Java: Developing with Java 2, Micro Edition*. At press.

KVM. Cldc and the k virtual machine (kvm). http://java.sun.com/products/cldc/.

LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. S. 2000. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*.

LEE, S. I., YANG, B. S., KIM, S., PARK, S., MOON, S. M., EBCIOGLU, K., AND ALTMAN, E. 2000. Efficient java exception handling in just-in-time compilation. In *Proceedings of ACM 2000 Java Grande Conference* (San Francisco).

PAULSON, L. D. 2001. Handheld-to-handheld fighting over java. *IEEE Comput. 34*, 7 (July), 253–257.

RIGGS, R., TAIVALSAARI, A., AND VANDENBRINK, M. 2001. *Programming Wireless Devices with the Java 2 Platform*. Addison-Wesley, Reading, Mass.

SMITH J. E., HEIL T., S. S., AND BEZENEK, T. M. 1998. Achieving high performance via co-designed virtual machines. In *Proceedings of International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems*. 77–84.

STICHNOTH, J. M., LUEH, G.-Y., AND CIERNIAK, M. 1999. Support for garbage collection at every instruction in a java compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.

TAKAHASHI, D. 2001. Java chips make a comeback. *Red Herring*.

VIJAYKRISHNAN, N., KANDEMIR, M., IRWIN, M. J., KIM, H. Y., AND YE, W. 2000. Energy-driven integrated hardware-software optimizations using simplepower. In *Proceedings of the International Symposium on Computer Architecture* (Vancouver, British Columbia).

VIJAYKRISHNAN, N., KANDEMIR, M., TOMAR, S., KIM, S., SIVASUBRAMANIAM, A., AND IRWIN, M. J. 2001. Energy characterization of java applications from a memory perspective. In *Proceedings of USENIX Java Virtual Machine Research and Technology Symposium*.

WILTON, S. AND JOUPPI, N. 1994. An enhanced access and cycle time model for on-chip caches. Tech. Rep. 93/5, DEC WRL Research Report.

YANG, B. S., MOON, S. M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOGLU, K., AND ALTMAN, E. 1999. Latte: A java vm just-in-time compiler with fast and efficient register allocation. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*. 128–138.

YANG, S., POWELL, M. D., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. N. 2001. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Proceedings of ACM/IEEE International Symposium on High-Performance Computer Architecture*.

Received January 2002; accepted July 2002