Journal of Circuits, Systems, and Computers, Vol. 11, No. 5 (2002) 537–555


# USING MEMORY COMPRESSION FOR ENERGY REDUCTION IN AN EMBEDDED JAVA SYSTEM*

G. CHEN†, M. KANDEMIR‡, N. VIJAYKRISHNAN§, and M. J. IRWIN¶

*Microsystems Design Lab, Pennsylvania State University,*
*University Park, State College, PA 16802, USA*
†*gchen@cse.psu.edu*
‡*kandemir@cse.psu.edu*
§*vijay@cse.psu.edu*
¶*mji@cse.psu.edu*

W. WOLF

*Department of Electrical Engineering, Princeton University,*
*Princeton, NJ 08544, USA*
*wolf@princeton.edu*

The Java programming language is being increasingly used for application development for mobile and embedded devices. Limited energy and memory resources are important constraints for such systems. Compression is an useful and widely employed mechanism to reduce the memory requirements of the system. As the leakage energy of a memory system increases with its size and because of the increasing contribution of leakage to overall system energy, compression also has a significant effect on reducing energy consumption. However, storing compressed data/instructions has a performance and energy overhead associated with decompression at runtime. The underlying compression algorithm, the corresponding implementation of the decompression and the ability to reuse decompressed information critically impact this overhead.

In this paper, we explore the influence of compression on overall memory energy using a commercial embedded Java virtual machine (JVM) and a customized compression algorithm. Our results show that compression is effective in reducing energy even when considering the runtime decompression overheads for most applications. Further, we show a mechanism that selectively compresses portions of the memory to enhance energy savings. Finally, a scheme for clustering the code and data to improve the reuse of the decompressed data is presented.

*Keywords*: Java virtual machine; memory compression; leakage energy; dynamic energy; embedded system.

## 1. Introduction and Motivation

Java has become a popular vehicle for portable network programming, spanning not just resource-rich server and desktop environments, but resource constrained

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

environments as well. It is estimated that the market for Java-enabled devices for resource-constrained environments such as cell-phones, PDAs and pagers will grow from 176 million in 2001 to 721 million in 2005.[1] Various embedded Java virtual machines (JVMs) and Java accelerators have been proposed to target this potential market over the past year.

A Java system for an embedded/portable environment needs to meet an entirely different set of constraints as compared to executing on a high-performance or desktop environment. Three important aspects to which current embedded JVMs such as Sun's KVM[2] and HP's ChaiVM[3] conform are soft real-time, restricted memory size, and long-duration sessions requirements. Energy consumption is also an important design consideration for such battery-driven systems. However, currently, there is little support for analyzing and optimizing energy behavior of such embedded JVMs. In particular, the energy consumption in the memory system is a significant portion of overall energy expended in execution of a Java application.[4] 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. Leakage energy is consumed as long as the device is powered and is consumed even when the device is not being accessed. While dynamic energy has been the traditional focus of most optimizations, leakage is becoming an equally important portion as supply voltages and thus threshold voltages and gate oxide thicknesses continue to scale.[5] Recent energy estimates for 0.13 micron process indicate that leakage energy accounts for 30% of L1 cache energy and as much as 80% of L2 cache energy.[6] Leakage energy is of particular concern in the dense memory structures as it increases with the size of the memory. In contrast, the effect of larger SRAM sizes on dynamic energy can be controlled by partitioning large SRAMs into smaller structures. Also, it is customary to use multiple levels of memory hierarchy to confine most accesses in the smallest memory.

In this work, we use a system-on-a-chip (SoC) with two-level memory hierarchy where a software-managed memory known as scratch pad memory (SPM) is used between the memory and the processor core. The SPM, due to its smaller size, has a lesser per access dynamic energy cost associated with it. Hence, confining most accesses to the smaller SPM (instead of large main memory) reduces the overall dynamic energy. However, the increased memory space due to the two-level memory hierarchy can increase the overall leakage energy of the system. Various compression schemes have been widely used to reduce the memory requirements. In this work, we use compression to reduce the size of the required memory. Specifically, we store the code of the embedded JVM system and the associated library classes in a compressed form in the memory. Thus, the effective number of active transistors used for storage and the associated leakage is reduced. We employ a mechanism that turns off power supply to the unused portions of the memory to control leakage. Whenever the compressed code or classes are required by the processor core, a mapping structure stored in a reserved part of the SPM serves to locate the required

block of data in the compressed store. Then, the block of data after decompression is brought into the SPM. Thus, the use of scratch pad memory in conjunction with a compressed memory store targets the reduction of both dynamic and leakage energy of a system. The focus of this paper is on investigating the influence of different parameters on the design of such a system. The issues addressed in this work are listed below:

(i) Storing compressed code or data has an associated decompression cost from both the energy and performance aspects. To obtain any energy savings, the energy overhead of decompression must be smaller than the leakage energy savings obtained through storage of compressed code. The underlying compression algorithm and the corresponding implementation of the decompression critically impact the energy and performance overhead. We explore this idea using a specific hardware compression scheme and also experiment with different decompression overheads to account for a range of possible implementations from customized hardware to software.

(ii) The size of the compressed block influences both the compression ratio and the overhead involved in indexing the compressed data. A larger granularity of compression, typically, provides a better compression ratio. In turn, this provides an ability to turn off power supply to more unused memory blocks, thereby providing larger leakage energy savings. Also, it reduces the mapping overhead for indexing into the compressed store. However, a larger block also occupies a larger space in the SPM and increases the storage pressure. This can lead to more frequent conflicts in the scratch pad memory resulting in more frequent decompressions.

(iii) Entities (native functions, Java method bytecodes and constant pools, etc.) in Java virtual machine are not equally used. So different portions are decompressed different number of times during runtime. Based on their hotness, we determine whether compression is beneficial (or not) and selectively compress the beneficial portions of the code. This technique reduces the overall system energy by up to 10%.

(iv) A longer reuse is essential to amortize the additional energy expended in decompressing when transferring from the memory to the scratch pad. Based on the traces of the applications, we identify native functions in the virtual machine and Java methods in the classes library that are frequently used together. Clustering items that are frequently used together improves reuse and thus saves energy.

The rest of this paper is organized as follows. Section 2 introduces the virtual machine used in this study, our SoC, and our embedded applications. Section 3 discusses our strategy for saving leakage energy through compression. Section 4 presents our simulation environment and Sec. 5 gives our experimental results. Finally, Sec. 6 concludes the paper with a summary of our major contributions.

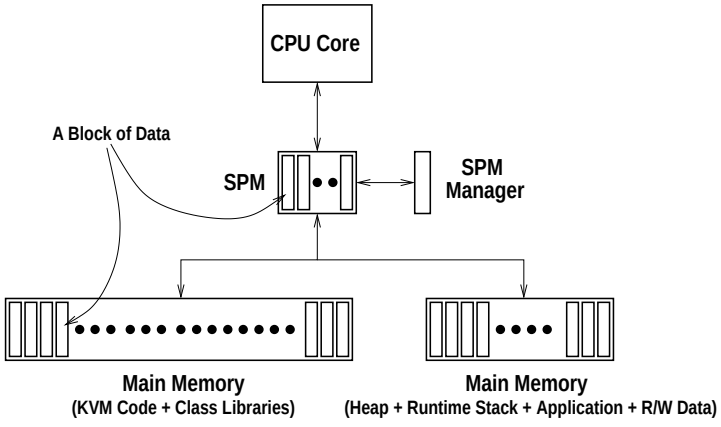
## 2. KVM, SoC Architecture, and Applications

### 2.1. *Virtual machine*

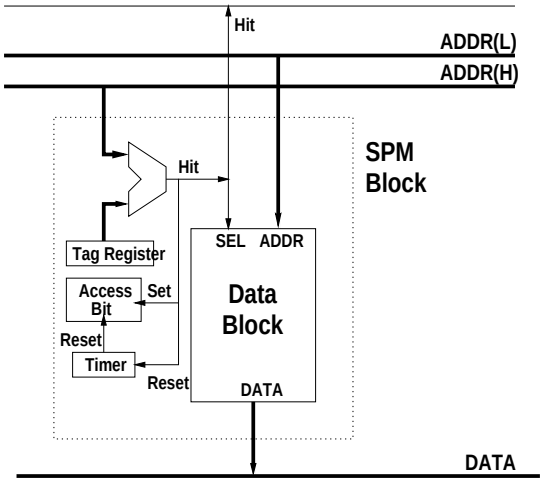
In this study, we used K Virtual Machine (KVM),[2] Sun's virtual machine designed for resource-constrained (e.g., battery-operated) environments. It targets embedded computing devices with as little as a few kilobytes total memory, including actual virtual machine and Java class libraries themselves. These devices include smart wireless phones, pagers, mainstream personal digital assistants, and small retail payment terminals.

### 2.2. *Architecture*

A system-on-a-chip (SoC) is an integrated circuit that contains an entire electronic system in a single sliver of silicon. A typical SoC contains a library of components designed in-house as well as some cores from chipless design houses also known as intellectual property. In this work, we focus on an SoC-based system that executes KVM applications. Figure 1(a) depicts the high level (logical) view of the relevant parts of our SoC architecture. This architecture has a CPU core, a scratch-pad memory (SPM), and two main memory modules. The CPU core in our SoC is a 100 MHz, 32-bit five-stage pipelined RISC that implements the SPARC architecture V8 specification. It is primarily targeted for low-cost uniprocessor applications. Both main memory and SPM are SRAMs which are organized as blocks. Unlike conventional SPM with fixed address,[7] each block of our SPM is dynamically mapped into a virtual address space which is as large as main memory. Each main memory block can be dynamically loaded into one SPM block. Each SPM block has a tag register indicating its virtual address. The tag registers are set by SPM manager. When the CPU generates an address, the high-order bits of this address are compared with each tag in parallel. If one of the tags generates a match, the corresponding SPM block is selected and low-order bits of the address are used to access the contents of the block. If no tag match occurs, then the "Hit" signal line (shown in Fig. 1(b)) is disabled and an interrupt is generated. The corresponding interrupt service routine activates the SPM manager which brings the faulted block from main memory to the SPM. In case no free SPM block is available, a timer-based block replacement policy is used. Specifically, for each SPM block, there is a timer and an access bit. Whenever the block is accessed, its access bit is set and its timer is reset. When a block is not accessed for a certain period of time, the timer goes off and the access bit is reset. When a block replacement is to be performed, the SPM Manager always tries to select a block whose access bit is reset. If no such block exists, the manager selects a block in a round-robin fashion. The main memory is composed of two parts: one part which contains the KVM code and class libraries and the other part which contains all writable data including heap and C stack as well as application code.




(a)




(b)

Fig. 1. High-level view of the SoC memory architecture. (b) Details for an SPM block.

It should be mentioned that a number of parameters in this architecture are tunable. For example, the capacities of SPM and main memory can be modified. Also, by playing with the width of the timers associated with each block, we can modify the behavior of the block replacement policy. Finally, the SPM block size can be changed. Note that changing the block size affects the block replacement rate as well as the overhead (per block) when a replacement occurs.

Instead of an SPM, a cache could also have been employed. In our experiments, we found that using 2-way associate 32 KB instruction and data caches both with line sizes of 32 bytes consumed 11% more energy than an equivalent 64 KB SPM configuration.

Table 1. Brief description of the benchmarks used in our experiments.

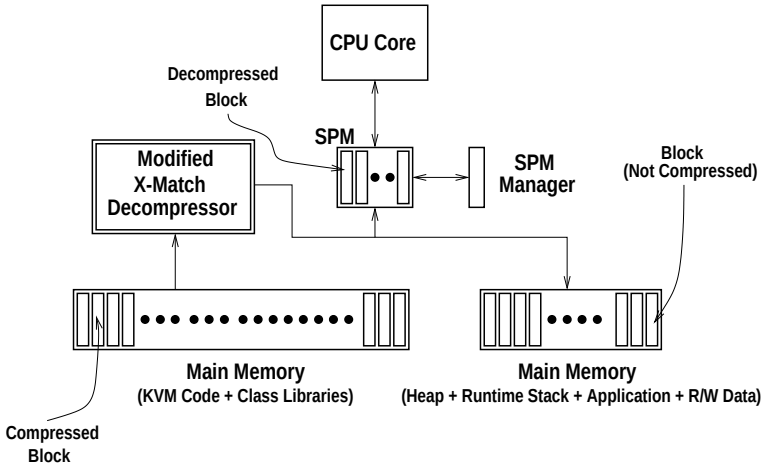
| Application | Brief description | Source |
|---|---|---|
| `Calculator` | Arithmetic calculator | www.cse.psu.edu/~gchen |
| `Crypto` | Cryptography | www.bouncycastle.org |
| `Dragon` | Game program | comes with Sun's KVM |
| `Elite` | 3D rendering | home.rochester.rr.com |
| `Kshape` | Electronic map | www.jshape.com |
| `Kvideo` | KPG decoder | www.jshape.com |
| `Kwml` | WML browser | www.jshape.com |
| `ManyBalls` | Game program | comes with Sun's KVM |
| `MathFP` | Math lib | home.rochester.rr.com |
| `Missiles` | Game program | comes with Sun's KVM |
| `Scheduler` | Weekly/daily scheduler | www.cse.psu.edu/~gchen |
| `StarCruiser` | Game program | comes with Sun's KVM |

### 2.3. *Applications*

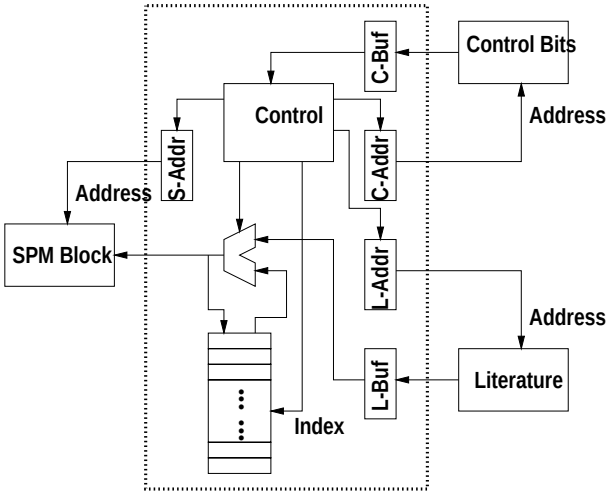
In this subsection, we describe the applications used in this study. To test the effectiveness of our energy saving strategy, we collected twelve applications shown in Table 1. These applications represent a group of codes that are executed in energy-sensitive devices such as hand-held computers and electronic game boxes, and range from utilities such as calculator and scheduler, embedded web browser to game programs. These applications represent a good mix of codes that one would expect to run under KVM-based environments.

## 3. Compressing KVM Code and Class Libraries

As noted earlier, leakage energy consumption of SRAM blocks is proportional to their size as well as the duration of time that they are powered on. In this work, we try to reduce the number of active (powered on) memory blocks by storing read-only data, including KVM binary codes and Java class libraries, in compressed form. To avoid incurring the cost of runtime compression, writable data are stored in original form. The high-level view of our architecture with the decompression support is shown in Fig. 2(a). When a data item belonging to a compressed memory block is requested by the processor, the whole block is decompressed by the decompressor and is then written into the SPM. The advantage of this strategy is that since data in read-only memory is in the compressed form, it occupies fewer memory blocks. This, in turn, reduces the leakage energy consumption in read-only part of the main memory system. Note that the amount of this saving is determined by the compression rate of the algorithm used. The drawback is that decompressing a block (at runtime) incurs both time and energy penalty. The magnitude of this penalty depends on how frequently decompression is required and how much time/energy it takes to decompress a block of data. The number of decompressions is directly related to the number of misses in the SPM (which is a characteristic of application behavior and SPM configuration). The time/energy expended in decompressing the




(a)




(b)

Fig. 2. (a) High-level view of the SoC memory architecture with decompressor. (b) Details for the modified X-Match decompressor.

block depends on the decompression method used and can be reduced by an efficient implementation.

The compression/decompression algorithm to be used in such a framework should have the following important characteristics: (1) good compression ratio for small blocks (typically less than 4 KB), (2) fast decompression, and (3) low energy consumption in decompression. Since compression is performed offline, its speed and energy overhead are not constrained. The first characteristic is desirable because the potential leakage energy savings in memory are directly related to the

number of memory blocks that need to be powered on. Note that a compressed memory block needs to be decompressed before it can be stored in the SPM. Consequently, a decompression overhead is incurred in every load to SPM and, in order for this scheme to be effective, we should spend very little time and energy during decompression.

Kjelso *et al.*[8] presented a dictionary-based compression/decompression algorithm called X-Match. This algorithm maintains a dictionary of data previously seen, and attempts to match the current data element (to be compressed) with an entry in the dictionary. If such a match occurs, the said data element is replaced with a short code word indicating the location of data in the dictionary. Data elements that do not generate a match are transmitted in full (literally), prefixed by a single bit. Each data element is exactly 4 bytes in width and is referred to as a tuple. A full match occurs when all characters in the incoming tuple fully match a dictionary entry. A partial match occurs when at least two of the characters in the incoming tuple match exactly a dictionary entry; the characters that do not match are transmitted literally. The coding function for a match encodes three separate fields: (1) match location, (2) match type indicating which characters from the incoming tuple matched the dictionary entry, and (3) any characters from the incoming tuple which did not match the dictionary entry at the match location (i.e., those transmitted without encoding). The decompressor of X-Match is implemented as a 3-staged pipeline. Compared to the baseline scheme (Fig. 1), the performance digression introduced by the decompressor is negligible (2 cycles' delay for each block loading).

In the original X-Match algorithm, the dictionary is maintained using a move-to-front strategy, whereby the current tuple is placed at the front of the dictionary and other tuples move down by one location. If the dictionary becomes full, the tuple occupying the last location is simply discarded. The move-to-front operation is implemented with content addressable memory, which is expensive from the energy consumption perspective. In our implementation, we replaced the move-to-front strategy with a simple round-robin strategy, i.e., the new tuple is always appended to the end of current dictionary entries. When the dictionary is full, the replacement pointer is moved to the first dictionary entry and the entry becomes the one that will be replaced next time. The elimination of the move-to-front strategy may cause a slight degradation in the compression ratio, but the implementation is simpler and energy-efficient. We also separate the literal bytes from the control bits (i.e., prefixes, match types, and dictionary locations), which allows the control bits and literal bytes to be fed into the decompressor as separate streams. We refer to this modified algorithm as the modified X-Match algorithm in the rest of the paper. The hardware block diagram of the modified X-Match decompressor is shown in Fig. 2(b).

While this modified X-Match implementation is used in our evaluation, the idea of trading additional decompression energy with reduced memory leakage energy is applicable using other compression schemes such as Lempel-Ziv and Huffman.

Haris Lekatsas *et al.*[9] has reported energy savings through cache-based memory compression. Our SPM approach is more flexible than cache in that SPM has its own address space and is managed by software. We can customize the management policy according to specific applications. For example, we can bypass SPM when accessing infrequently used data, or we can pin some frequently accessed data in the SPM, or we can use it as a conventional SPM by fixing the values of the tag registers.

## 4. Simulation Methodology

### 4.1. *Energy model*

The energy numbers reported in this paper are obtained by a simulator implemented on the SPARC simulation tool-set, Shade,[10] augmented with energy models. The simulator takes as input the KVM system executing a Java application and computes performance as well as energy data. The current implementation runs on SPARC systems and simulates the SPARC V8 instruction set of our target processor. Our simulator tracks energy consumption in the processor core, SPM, and main memory blocks. 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 base energy consumptions of the different instruction types are obtained using a customized and validated version of our in-house cycle accurate energy simulator.[11] The simulator is configured to model a five-stage pipeline similar to that of the target embedded SPARC V8 architecture.

The energy consumption in SPM and main memory is divided into two components: dynamic energy and leakage energy. In computing per access dynamic energy consumptions for SPM and main memory, we used CACTI 2.0[12] assuming a 0.10 micron technology. In computing the leakage energy, we assumed that the leakage energy per cycle of the entire main memory is equal to the dynamic energy consumed per access. This assumption tries to capture the anticipated importance of leakage energy in the future. It should be stressed that leakage becomes the dominant part of energy consumption for 0.10 micron (and finer) technologies for the typical internal junction temperatures in a chip.[5] Note that, as opposed to dynamic energy which is expended only when an access occurs, leakage energy is spent as long as memory is powered on.

In computing the overall energy consumption in main memory and SPM, we assumed that a memory block (or an SPM block) can be in one of three states (modes) at any given time: $R/W$, *active*, or *inactive*. In the R/W (read/write) mode, memory is being read or written and consumes full dynamic energy as well as full leakage energy. In the active state, on the other hand, the memory is powered on but not being accessed. In this state, it consumes no dynamic energy but full leakage energy. Finally, the memory modules that are not needed by the system are not powered on, i.e., in the inactive state, consequently, no energy consumption at all. Obviously, one would want to place as many memory blocks as possible to

the inactive state so that the energy consumption can be minimized. One way of achieving this is to reduce the amount of data stored in memory, which can be achieved using compression.

### 4.2. *Base configuration and energy distribution*

Table 2 gives the simulation parameters used in our base configuration. Table 3 shows (in columns two through five) the energy consumptions (in micro-joules) for our applications executing on base configuration without decompression. The energy consumption is divided into four components: dynamic energy in SPM, leakage energy in SPM, dynamic energy in main memory, and leakage energy in main memory. The contribution of the processor (not including cache) energy to the overall (main memory + SPM + processor) energy is around 10% and is not much affected by decompression. Consequently, we focus only on main memory and SPM energies. A memory block that contains no valid information throughout the application execution is turned off so that it does not consume any leakage energy. We see from these results that the memory leakage energy consumption (shown in the third column) constitutes a large percentage of the memory system (main memory + SPM) energy budget (61.74% on the average) and is a suitable target for optimization. The sixth column in Table 3 gives percentage of energy consumption due to read-only part of the memory. We see that, on the average, the read-only part of the memory is responsible for 62.42% of the overall memory energy consumption. Finally, Table 4 gives the number of SPM misses and the number of execution cycles (in millions) for each application.

Table 2. Simulation parameters and their values for our base configuration.

| Parameter | Value |
|---|---|
| SPM capacity | 64 KB<br>4 KB SPM management<br>40 KB for read-only data<br>20 KB for writable data |
| SPM block size | 1 KB for read-only data<br>512 bytes for writable data |
| Main memory capacity | 512 KB |
| SPM access time | 1 cycle |
| Main memory access time | 3 cycles |
| SPM dynamic energy/read | 0.5216 nJ |
| SPM dynamic energy/write | 0.6259 nJ |
| Main memory dynamic energy/read | 1.334 nJ |
| Main memory dynamic energy/write | 1.601 nJ |
| Main memory leakage energy/byte/cycle | $2.54 \times 10^{-6}$ nJ |
| SPM leakage energy/byte/cycle | $2.54 \times 10^{-6}$ nJ |
| SPM access bit reset time | 6000 cycles for read-only<br>4000 cycles for r/w clean<br>8000 cycles for r/w dirty |

Table 3. Energy consumptions for our applications under the base configuration.

| Application | Memory energy (nJ) | | SPM energy (nJ) | | Read-only contribution |
|---|---|---|---|---|---|
| | Dynamic | Leakage | Dynamic | Leakage | |
| `Calculator` | 1.81 | 7.46 | 3.08 | 0.87 | 66.00% |
| `Crypto` | 7.30 | 137.94 | 54.97 | 16.17 | 60.46% |
| `Dragon` | 1.10 | 68.70 | 27.18 | 8.05 | 60.86% |
| `Elite` | 1.10 | 64.42 | 25.49 | 7.55 | 60.88% |
| `MathFP` | 2.42 | 104.96 | 41.70 | 12.30 | 60.40% |
| `ManyBalls` | 5.90 | 73.08 | 29.22 | 8.56 | 59.98% |
| `Missiles` | 2.51 | 60.34 | 23.96 | 7.07 | 61.61% |
| `KShape` | 16.18 | 201.90 | 80.93 | 23.66 | 61.22% |
| `KVideo` | 2.21 | 12.69 | 5.17 | 1.49 | 64.87% |
| `KWML` | 121.56 | 584.96 | 240.78 | 68.55 | 64.35% |
| `Scheduler` | 32.96 | 140.51 | 57.83 | 16.47 | 66.50% |
| `StarCruiser` | 3.22 | 48.96 | 19.58 | 5.74 | 61.91% |

Table 4. Execution cycles and SPM misses for our applications under the base configuration.

| Application | Number of SPM misses | Number of cycles ($10^6$) |
|---|---|---|
| `Calculator` | 5258 | 5.59 |
| `Crypto` | 19848 | 103.58 |
| `Dragon` | 3064 | 51.58 |
| `Elite` | 3206 | 48.37 |
| `MathFP` | 6735 | 78.81 |
| `ManyBalls` | 11455 | 54.87 |
| `Missiles` | 7288 | 45.30 |
| `KShape` | 45321 | 151.61 |
| `KVideo` | 6396 | 9.53 |
| `KWML` | 339395 | 439.26 |
| `Scheduler` | 96035 | 105.50 |
| `StarCruiser` | 9356 | 36.76 |

## 5. Results

In this section, we present data showing the effectiveness of our strategy in saving energy and also measure the sensitivity of our strategy to different parameters such as SPM capacity, block size, and cost of decompression. All energy numbers reported here are values normalized to the energy consumption in the base case without any decompression (Table 3). Also, when a simulation parameter is modified, the remaining parameters maintain their original values given in Table 2.

The top part of Fig. 3 gives the normalized energy consumptions in read-only portion of the main memory and the SPM. It can be observed from this figure that the energy saving is 20.9% on the average. The bottom part of Fig. 3, on the other hand, shows the overall (normalized) energy consumption in main memory and SPM, including the energy expended during decompression. We see that most of

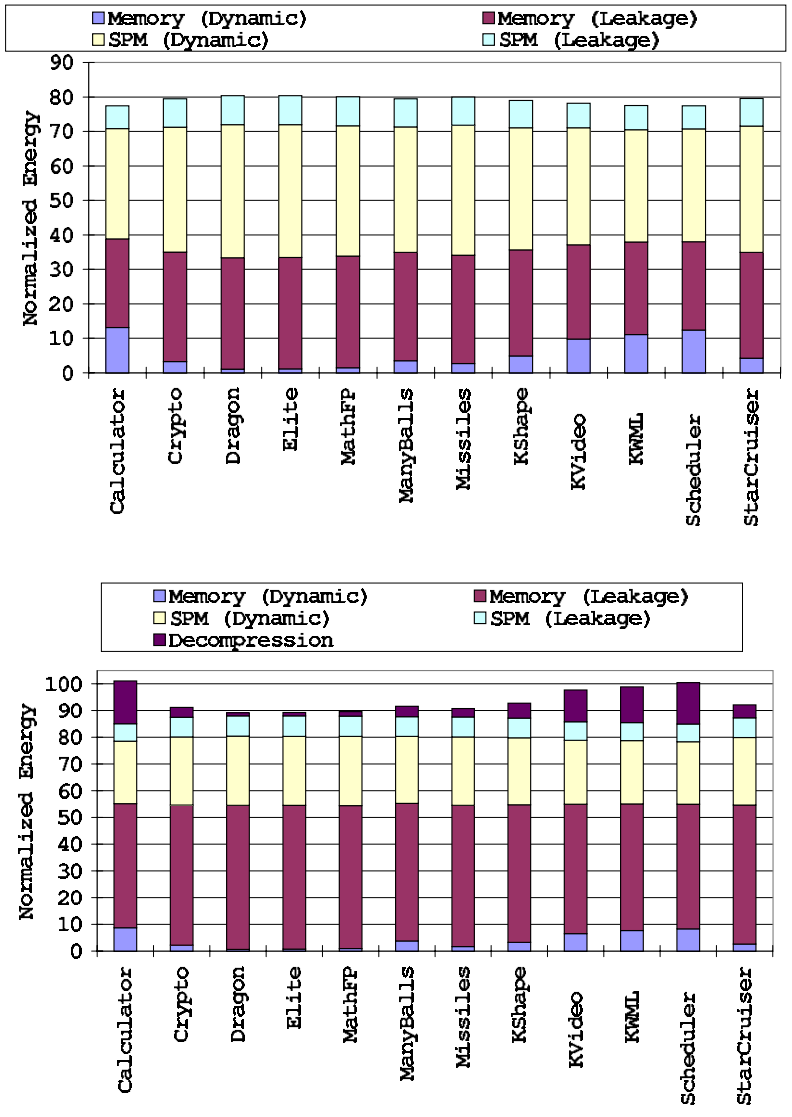



Fig. 3. Normalized energy consumption in read-only memory and SPM (top) and in overall memory and SPM (bottom).

the applications achieve an overall energy saving of 10% (an average of 7% across all applications). In two applications (`Calculator` and `Scheduler`), the decompression overhead (energy) plays a larger role and the overall energy consumption becomes worse than the original case. We also experimented with a 50% reduction in leakage energy per main memory cell to account for design variations that permit the slower main memory cells to operate using a higher threshold voltage. In this case, the overall memory system energy saving across all applications is 5.4% on the average.

In general, there are two application-related factors that determine the effectiveness of our energy saving strategy: (1) the overall running time of the application, (2) the number of SPM misses. Since the major energy gain in our strategy comes from the memory leakage energy, the longer the application runs, we can expect more energy benefits. Recall that each SPM misses invokes a decompression. Therefore, the number of SPM misses is an important factor in determining the energy spent in decompression during the course of execution. The reasons that `Calculator` and `Scheduler` do not get benefit from our strategy are different. In `Calculator`, the execution time is rather short (only 5.59 million cycles) and the energy spent on decompression does not pay off. On the other hand, although the execution time of `Scheduler` is not short (105.5 million cycles), it suffers from a high number of SPM misses (a total of 96 033).

Since our strategy focuses on energy savings in the read-only part of the main memory, in the rest of this section, we mainly present results pertaining only this part and the SPM (unless otherwise stated). However, to evaluate the impact of decompression, we also show the energy consumed during the decompression process.

### 5.1. *Sensitivity to the decompression cost*

To see how a more efficient or a less efficient implementation of the X-Match decompressor would impact our results, we performed another set of estimations. Assuming a decompression rate of 4 bytes/cycle and that the energy consumed in each stage is equal to one SPM access, we determined that the energy consumption for decompressing one word (4 bytes) is equal to three SPM accesses. We normalize this energy cost to 1 and experiment with its multiple as well as its fractions. The results shown in Fig. 4 are average values (over all benchmarks) and illustrate that the relative cost of decompression can change the entire picture. For example, with a relative cost of 2, the energy consumption exceeds that of the original case. In contrast, with a relative cost of 0.25, the energy consumption is around 80%, even including the decompression energy. These results clearly indicate the importance of efficient implementation of decompression.

### 5.2. *Sensitivity to the SPM size*

Figure 5 shows the impact of SPM capacity (size) on energy savings. As before, the values shown are averages computed over all benchmark codes in our experimental suite. We observe from this figure that, if the SPM size is too small, frequent SPM misses make energy consumption very high. But, we also see that a very large SPM also degrades energy behavior. There are two factors that together create this behavior. First, a larger SPM itself consumes more dynamic and leakage energy (compared to a smaller SPM). Second, for each application, there is an SPM capacity that captures the working set. Increasing the SPM size beyond this capacity does not reduce the number of misses further. So, this stability in the number of misses, combined with the first factor, leads to an increase in energy consumption.

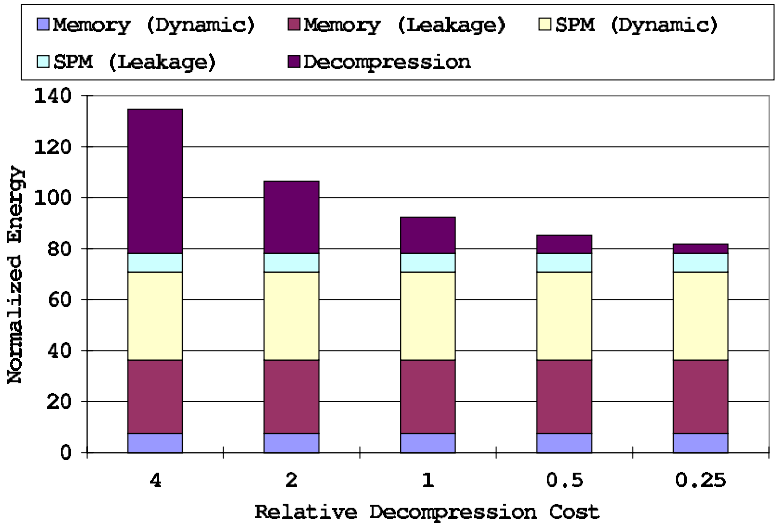



Fig. 4. Normalized energy consumption in read-only memory with varying decompression costs.

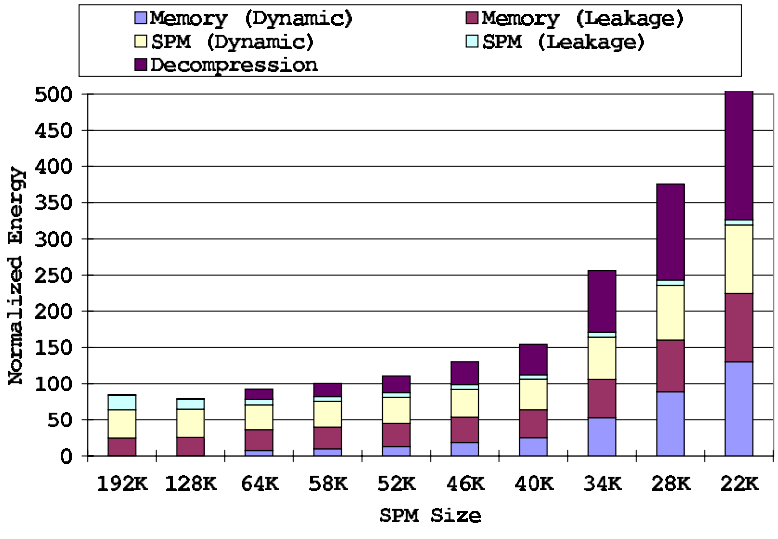



Fig. 5. Normalized energy consumption in read-only memory with varying SPM sizes.

In an embedded system design, the maximum possible SPM size is determined by the chip budget. Our experimentation indicates that the best SPM size depends on the application at hand. So, embedded system designers should select a suitable SPM size considering the applications in question as well as the impact of SPM size on energy consumption.

### 5.3. *Sensitivity to the block size*

In this set of experiments, we tried to measure the sensitivity of our energy savings to the block size used. Recall that our default block size was 1 KB. The results shown in Fig. 6 indicate that, given the SPM capacity, the size of each block has a great impact on the energy consumption. For most compression algorithms in the literature, a larger block size has, in general, a better compression ratio (Fig. 7). In addition, given the same SPM size, smaller block size increases block number, which complicates the mapping mechanism. However, it should be noted that, a very large

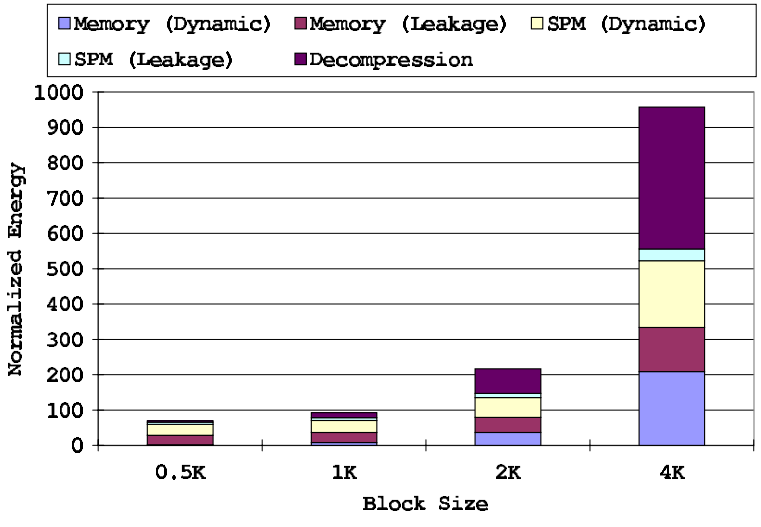



Fig. 6. Normalized energy consumption in read-only memory with varying block sizes.

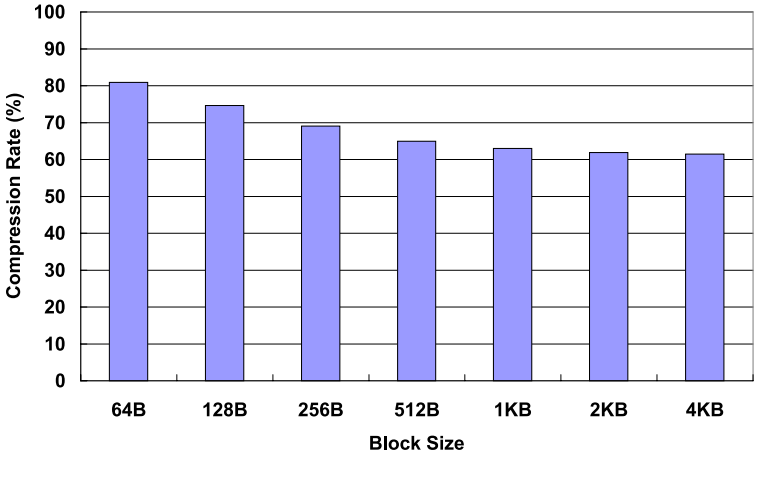



Fig. 7. Compression rate of Modified X-match algorithm with varying block sizes.

block size might increase both SPM miss rate and miss penalty (decompression cost). That is exactly the behavior we observed during our experiments. As shown in Fig. 6, a block size 0.5 KB generated better results than our default 1 KB blocks. In contrast, increasing the block size to 2 KB increased the original energy consumption by more than a factor of two.

### 5.4. *Selective compression*

Compression reduces the memory size and thus leakage energy, but it also introduces decompression overhead. The idea behind selective compression is that only the blocks for which compression brings net benefit are stored in a compressed format. The other blocks are in an uncompressed format. Specifically, a block $B$ is stored in compressed format only if $load(B) * cost(B) < L * T * size(B) * (1 - r(B))$, where $load(B)$ is the number of times that block $B$ is loaded from main memory, $cost(B)$ is the decompression cost of $B$, $L$ is the leakage energy per byte per cycle, $T$ is the overall execution time in cycles, $size(B)$ is the size of block $B$, and $r(B)$ is the compression rate of block $B$.

In this work, we evaluated two selection strategies: application-customized selection and global selection. The application-customized strategy is based on the traces of a specific application; that is, for each application, the blocks to be stored in the compressed form are determined considering only the KVM behavior of that application. Consequently, this strategy may achieve the optimal energy consumption of the particular application. However, since this optimization is completely for one application, it may degrade others. In other words, the KVM blocks that need to be compressed for the best behavior of one application may not be suitable candidates (for compression) for other applications. Our second strategy, global selection, is to select blocks for compression according to the traces of a group of applications. Each application has a specific weight based on the frequency of its usage. Tuning the weights with a group of carefully selected representative applications may bring general improvements on a variety of applications. The results shown in Fig. 8 indicate that the application-customized strategy generates the best results.

### 5.5. *Clustering*

Compared to a cache line, our SPM block has larger granularity, and thus is more sensitive to the applications' spatial locality. Putting items that are frequently used together into the same block can reduce accesses to main memory, and thus decompression energy. We exploit such clustering in both KVM binary and Java Class Library. Clustering in KVM binary is performed at the native function and Java method granularity. Using profiling information, we first build the weighted call graph, in which each node represents a native function in KVM binary or a Java method in the Java class library. The weight of each edge in this graph indicates how frequently each pair of native functions or Java methods are called in a row. Then a greedy algorithm is applied to clustering the functions and methods according to

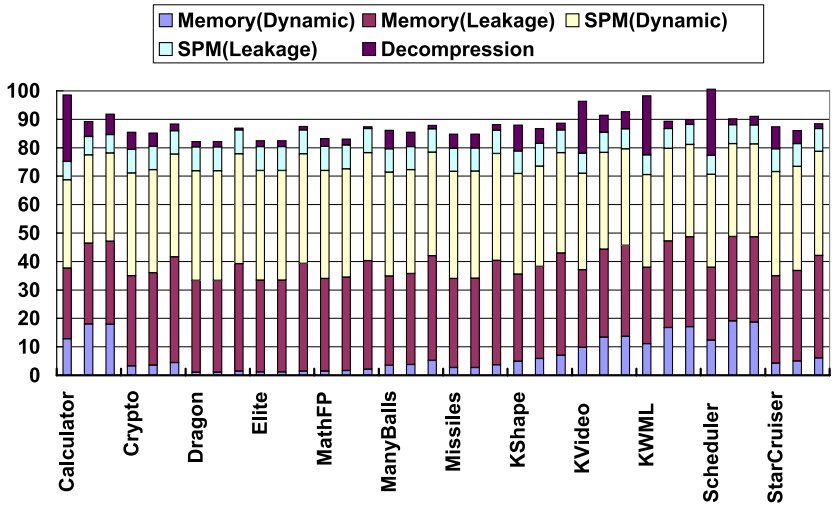



Fig. 8. Comparison of different compression strategies. The three bars for each application from left to right correspond to our default strategy, an application-customized selective strategy, and a global selective strategy.

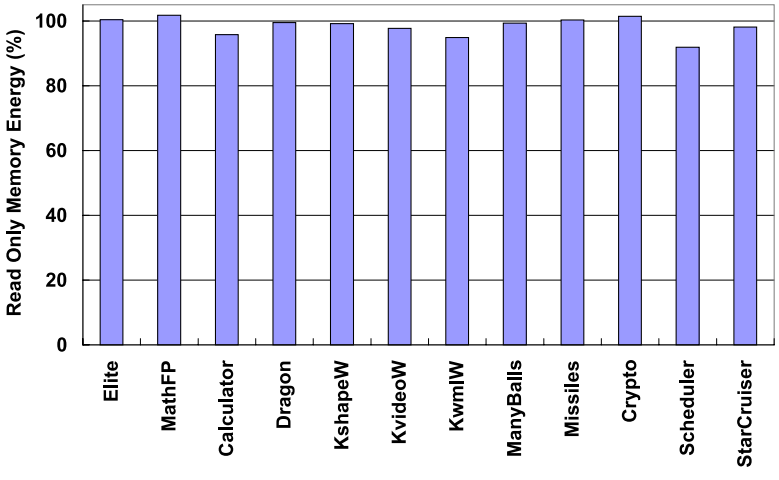



Fig. 9. Effectiveness of clustering. Energy values are normalized to the same configuration without clustering.

the weights of edges. Similar to selective compression, clustering can be performed in either application-customized or global manner. In this work, we report only the results of global clustering in Fig. 9. For most applications, clustering saves energy. However, since clustering is performed according to the common memory access patterns of all applications, the energy behaviors of some applications that do not conform to the common memory access patterns, such as MathFP, may degrade.

## 6. Conclusions

Storing compressed code or data has an associated decompression cost from both the energy and performance aspects. However, compression itself helps to reduce the portion of the memory to be powered on and the consequent leakage energy of the memory system. Our experiments with a set of embedded applications using a commercial embedded JVM and a specific compression scheme show that the proposed technique is effective in reducing system energy. We expect our findings to be applicable to other compression algorithms and implementations as well.

## References

1. D. Takahashi, "Java chips make a comeback", Red Herring, July 12, 2001.
2. R. Riggs, A. Taivalsaari, and M. VandenBrink, *Programming Wireless Devices with the Java 2 Platform*, Addison Wesley, 2001.
3. ChaiVM for Jornado, http://www.hp.com/products1/embedded/jornado/index.html.
4. N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam, and M. J. Irwin, "Energy characterization of Java applications from a memory perspective", *Proc. USENIX Java Virtual Machine Research and Technol. Symp.*, April 2001.
5. A. Chandrakasan, W. J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2001.
6. 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", *Proc. ACM/IEEE Int. Symp. Low Power Electron. and Design*, August 2000.
7. M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management", *Proc. Design Automation Conf.*, New Orleans, LA, June 2002.
8. M. Kjelso, M. Gooch, and S. Jones, "Performance evaluation of computer architectures with main memory data compression", *J. Syst. Architecture* **45** (1999) 571–590.
9. H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design", *Proc. 37th Design Automation Conf.*, Los Angeles, CA, June 2000.
10. B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling", *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Syst.*, May 1994, pp. 128–137.
11. N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye, "Energy-driven integrated hardware-software optimizations using SimplePower", *Proc. Int. Symp. Computer Architecture*, Vancouver, British Columbia, June 2000.
12. G. Reinman and N. Jouppi, "An integrated cache timing and power model", COMPAQ Wester Research Lab, Palo Alto, CA, 1999, http://www.research.compaq.com/wrl/people/jouppi/CACTI.html.
13. L. Benini and G. De Micheli, "System-level power optimization: Techniques and tools", *ACM Trans. Design Automation Electron. Syst.* **5**, 2 (2000) 115–192.
14. F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology — Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer Academic Publishers, June 1998.
15. G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded Java environment", *Proc. 8th Int. Symp. High-Performance Computer Architecture*, Cambridge, MA, February 2–6, 2002.
16. The future of SoC design, http://www.eetasia.com/ART_8800141212.HTM.

17. G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf, "Energy savings through compression in embedded Java environments", *Proc. 10th Int. Symp. Hardware/Software Co-Design*, Estes Park, CO, May 2002.
18. J. Flinn, G. Back, J. Anderson, K. Farkas, and D. Grunwald, "Quantifying the energy consumption of a pocket computer and a Java virtual machine", *Proc. Int. Conf. Measurement and Modeling of Computer Syst.*, June 2000.