Energy Behavior of Java Applications from the Memory Perspective*

N. Vijaykrishnan, M. Kandemir, S. Kim,
S. Tomar, A. Sivasubramaniam and M. J. Irwin
Dept. of Computer Science and Engg.
The Pennsylvania State University
University Park, PA 16802
vijay@cse.psu.edu

Abstract

With the anticipated dramatic growth of computing devices for mobile and embedded environments, energy conscious hardware and software design has taken center-stage together with performance. At the same time, there is an increasing need to provide a portable and seamless software environment for application development and interoperability. This paper takes an important step in the confluence of these two emerging trends, by examining the energy behavior of the memory system in the execution of Java applications. It is crucial to understand and optimize the energy behavior of the memory system since instructions referencing memory can contribute to a large fraction of the energy consumption when executing Java applications.

Using an off-the-shelf JVM, a validated memory energy model, and a detailed simulator, this paper presents a characterization of the energy consumption by the cache and main memory when executing the SPEC JVM98 benchmarks in the JIT and interpreter modes. The energy consumption is profiled for the different hardware components (instruction cache, data cache, memory) and software components (class loading, garbage collection, dynamic compilation). The results from such a characterization are useful to the hardware designer for cache organizations and architectural enhancements for reducing energy consumption. They are also useful to the application and runtime system designer to identify energy bottlenecks, and for code restructuring or algorithm redesign to alleviate these bottlenecks.

*This research is supported in part by grants from NSF CCR-0073419, Pittsburgh Digital Greenhouse and Sun Microsystems.

1 Introduction

Computing is becoming a pervasive and ubiquitous part of everyday life. This has led to important design considerations from the software, the need to provide a seamless and portable software platform that facilitates easy inter-operability, and hardware, the need to incorporate energy and form factor conscious designs and to reduce time-to-market and cost, angles. This paper brings these two design considerations together by examining the energy consumption of JVM implementations that form the cornerstone of Java [1], which is one of the software platforms for the seamless integration of diverse ubiquitous/embedded devices. In particular, this paper focuses on the energy consumed by the memory system when executing the SPEC JVM98 benchmarks [27].

Java provides portability across systems by specifying only the format and semantics of the bytecodes, without placing any restrictions on how they are executed. Consequently, the choice of the JVM implementation style depends on the performance, availability of hardware resources, as well as energy criteria. Previous studies have mainly looked at performance and hardware resource issues between JVM implementation styles. This is one of the first studies to examine JVM implementation styles from the energy viewpoint. Energy, measured in Joules, is the power consumed over time and is an important metric to optimize for prolonging battery life. The importance of optimizing for this metric has been further accentuated by the slow improvements in the energy capacity of batteries.

The focus of this paper is in studying

the energy consumption behavior of the Java codes from both software and hardware perspectives. While energy consumption is an important issue in mobile systems of varying degree of resource constraints, the experimental setup and evaluation benchmarks used in this work are representative of high-end mobile devices such as laptops. The JVM used in our experiments is the Sun Labs Virtual Machine for Research, formerly known as ExactVM (EVM) [31]. While current low-end mobile devices use small footprint JVMs that use simple interpretation, more sophisticated JVMs such as the one used in this work are attractive, due to their performance, for high-end mobile devices such as laptops and emerging low-end mobile devices such as palmtops with large memory modules.

We utilize the SPEC JVM98 benchmarks in this work and, specifically, focus on characterizing the memory system energy consumption for the following reasons:

- It has been observed [37, 4] that the memory system can consume a large fraction of the overall system energy, making this a ripe candidate for software and hardware optimizations.

- Among the different instructions executed by the SPARC architecture while executing the SPEC JVM98 benchmarks, the load/store instructions are the primary instructions that access the memory for their operands. On the average, we find the energy consumed by load/store instructions accounts for 58.7% and 52.2% of the total energy in the interpreted and JIT-compiled modes, respectively, when executing the SPEC JVM98 benchmarks (see left side of Figure 1). The energy consumed by load/store instructions is significant considering that on an average, these instructions are observed to constitute only 22.7% and 19.4% of the total instructions for the interpreted and JIT modes, respectively (see right side of Figure 1).

- Finally, Java executions are expected to stress the memory system more than traditional programs [17, 3, 9]. Bytecodes are treated as data, and need to be fetched from memory for interpretation or JIT-compilation and installed. Further, JVM features such as garbage collection make Java executions much more memory-intensive than normal programs.

We examine the energy consumption of the memory system when executing Java programs, with different JVM implementation styles, to understand the hardware and software bottlenecks. From the hardware viewpoint, such information can be used to suggest cache organizations and strategies for optimizing energy-delay criteria. Identifying energy bottlenecks for different software components of the execution, such as class loading, garbage collection or dynamic compilation, can lead to the design of better algorithms and mechanisms to reduce the energy demands. For instance, if garbage collection turns out to be energy hungry, one could either opt to invoke the collector less frequently or design more energy-conscious garbage collection algorithms. The information can also be used to decide when to interpret and when to compile, rather than base this decision purely on performance considerations.

To our knowledge, there has been only one prior study that has attempted to profile the energy consumption of Java programs [10]. However, their use of an actual pocket-computer to measure the current for calculating energy has limited their extent of profiling. It does not provide adequate information as to what hardware components are consuming the energy during the different phases of execution. On the other hand, the use of a detailed simulator helps us profile the energy consumption from both the hardware and software angles to isolate the fraction consumed by the memory system. Using the SPEC JVM98 benchmarks on off-the-shelf JVM implementations in the interpreted and JIT-compiled modes, this paper sets out to answer the following questions:

- How much energy is consumed by the memory system in the execution of Java programs? What fraction is consumed by the cache and what fraction by the main memory? What is the energy breakdown between instruction and data references?

- How do different cache configurations affect the energy consumption? In particular, what is the effect of the cache size and associativity? While cache configurations that traditionally favor lo-

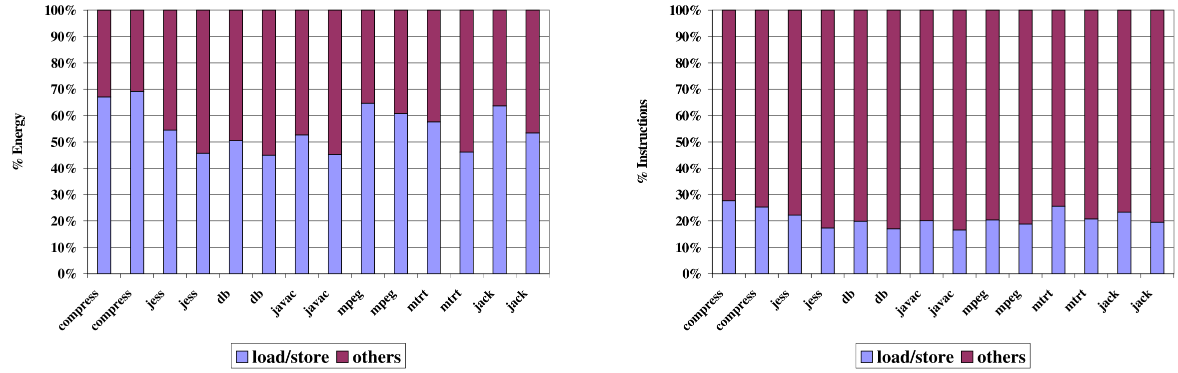

Figure 1: *Energy consumption distribution between load/store and other instructions (left). Distribution of dynamic instructions categorized as load/store and other instructions when executing on a SPARC architecture (right). The energy consumption for memory references are assumed to hit in the cache. Energy consumption numbers for each group of instructions are applied from [34]. Energy consumed in both datapath and memory is included.*

cality would reduce the references to main memory and reduce memory energy consumption, those configurations would also increase the energy cost per access of the cache. Studying these trade-offs would help decide on a good operating point.

- How do these results differ between the two JVM implementation styles — interpretation and JIT compilation? How much energy is consumed by different components of the JVM such as class loader, garbage collector? How do the cache parameters affect these components? What is the impact of application parameters, problem size in particular, on the energy consumption profile? For instance, a larger problem can result in the garbage collector being invoked more often.

2 Experimental Framework

In this section, we first give details on the specific JVM used in our simulations. We then describe the characteristics of the benchmarks used in our experiments, and the simulator used to gather memory access information. The energy model used for the memory system is also described.

2.1 Java Virtual Machine (JVM)

The JVM used in our experiments is the Sun Labs Virtual Machine for Research, EVM [31]. This is a high performance VM that has been designed to facilitate experiments in memory management. It is designed to provide a fast memory system, fast synchronization, and a fast Just-in-Time (JIT) compiler. It is also an adaptive VM, i.e., it detects and accelerates performance-critical or often-used code. Rather than compiling the entire program when it starts, as is done in virtual machines with pure JIT compilers, the EVM starts off running the application using an interpreter. At the same time, it gathers *profiling information* regarding the runtime characteristics of the application and uses this information to dynamically compile certain methods. In all our experiments, methods with loops are compiled at the first invocation itself, and methods without loops are compiled when the invocation count reaches 15, a value determined after extensive experimentation [22].

We identify three parts within the EVM, apart from the actual application execution, where the principal events of interest that stress the memory subsystem occur. The first is *class loading,* when the binary form of a class is brought into the virtual machine on the first invocation, incurring many compulsory cache misses. The second is *dynamic method compilation,* when a method is converted into native code, and the native code is installed in memory for subsequent execution. This event occurs only in the EVM executing as an adaptive compiler. Finally, there is *garbage collection,* which runs periodically as a separate thread and reclaims memory from non-referenced objects. A default heap size of 24 megabytes was used in the experiments. The remaining parts have been grouped together with the execution of native code and includes object accesses, thread creation, synchronization, etc.

2.2 SPEC JVM98 Benchmarks

We use all seven applications from the SPEC JVM98 benchmark suite [27] for our experiments that are briefly described in Figure 2. The benchmark programs can be run using three different problem sizes, which are named as s100, s10 and s1. While the problem sizes are larger for the s100 dataset size, the sizes do not scale as designated by the labels 100, 10 and 1. In interest of simulation time, especially, the interpreted mode in s100 mode, we use the smaller problem size (`s1`). We believe that the s1 data set is representative of shorter running applications or applets. While some of the results from the s1 observations would be applicable to the larger data sets, it must be noted that the impact of garbage collection and dynamic compilation impacts tend to change for the larger data sets. Thus, we provide s10 and s100 energy breakdowns in Figure 12 to emphasize this difference.

2.3 Memory System Energy Model

For obtaining detailed profiles, we have customized an energy memory simulator and analyzer using the Shade [6] tool-set. Our memory simulator models on-chip instruction cache (Icache), on-chip data cache (Dcache), and off-chip memory, and allows the user to modulate the various parameters for these components. We characterize the overall energy of the memory system by the energy consumed by five components: the instruction cache, the data cache, the buses, the I/O pads and the main memory.

Note that we focus only on the dynamic energy consumption, since in current technology, the dynamic energy accounts for 80% of the total energy whereas the rest, short circuit and leakage, takes only 20% [14]. The energy consumed by the Icache and by the Dcache are evaluated using an analytical model that has been validated to be highly accurate, within 2.4% error, for conventional cache systems [16, 25]. The energy consumed by the caches are based on technology related parameters for 0.8 micron technology. The energy consumption depends on the number of cache bit lines, word lines, and the number of accesses due to both hits and misses. The details of the model are quite involved and can be found in [22]. The address and data buses between the Icache/Dcache and the datapath account for the energy consumed by the on-chip buses. The bus energy consumption is evaluated by monitoring the switching activity on each of the bus lines using a capacitive load of 0.5pF per line [39]. The energy consumed by the I/O pads and the external buses to the main memory from the caches is evaluated similarly for a capacitive load of 20pF per line. The main memory energy is based on the model in [25] and uses a *per main memory access* energy, referred as E_m in the rest of the paper of 4.95×10^{-9} Joules. A 32 megabyte main memory recommended as smallest memory size for running SPEC JVM98 benchmarks [27] is used in this work. The energy consumed by main memory accesses is further broken down into that consumed due to instruction accesses (Imemory) and data accesses (Dmemory).

3 Energy Behavior

In this section, we present a detailed energy analysis of the SPEC JVM98 benchmarks. We first present the overall energy picture. Then, we zoom in on two codes, `javac` and `db`, and investigate their energy behavior by modifying different cache parameters such as cache size and associativity. These experiments give us the *hardware view* of energy consumption; i.e., they help us answer the question of which parts of the memory system consume the most energy. Subsequently, we present an energy breakdown for the benchmarks from the *software viewpoint.* This helps us understand which software components consume the most energy. Experiments are conducted for both JVM implementation styles: *interpreter mode* and *adaptive dynamic compilation mode* (*JIT mode*).

3.1 Overall Energy Picture

Figure 3 shows the percentage energy breakdown for the interpreter and the JIT modes assuming 32 KB two-way set associative instruction and data caches. We see from the figure on the left that most of the energy in the JIT mode is due to memory accesses for data (Dmemory). In contrast, the memory energy due to native SPARC instruction accesses is lower. This is because the in-

Benchmark	Brief Description
`compress`	A high-performance application to compress/uncompress large files; based on the Lempel-Ziv method (LZW)
`jess`	A Java expert shell system based on NASA's CLIPS expert shell system
`db`	A small database management program that performs several database functions on a memory-resident database
`javac`	JDK 1.0.2 Java compiler
`mpeg`	MPEG-3 audio file compression application
`mtrt`	Dual-threaded ray tracer; the only multi-threaded application in the suite
`jack`	A Java parser generator with lexical analyzers; an early version of what is now called JavaCC

Figure 2: *SPEC JVM98 benchmark codes used in the experiments.*

struction accesses exhibit much better data locality and cause less number of off-chip accesses. The same trend is also observed with the interpreter mode of operation as shown in the graph on the right. The instruction accesses in the interpreter mode have better locality than the JIT mode [23], and as a result, the percentage of the instruction cache energy is higher in the former, especially for `compress`. As an example, in the interpreter mode, instruction cache energy consumption for the `javac` benchmark is 12.3% of the overall memory system energy, whereas the corresponding figure in the JIT mode is 8.7%.

The energy consumption for a given unit is directly related to the number of accesses and per access energy cost [37]. It can be observed from Figure 4 that the interpreter consumes significantly more energy than in the JIT mode. A detailed breakdown of energy is shown in Figure 5. This is due to the fact that both the number of instructions and number of data accesses in the interpreter mode are higher than the JIT mode. For example, `javac` consumes 2.15 times more energy in the interpreter mode as compared to the JIT mode. Thus, the use of JIT compilers will be more beneficial not only in terms of performance (as is well-known [23]) but also from the energy viewpoint.

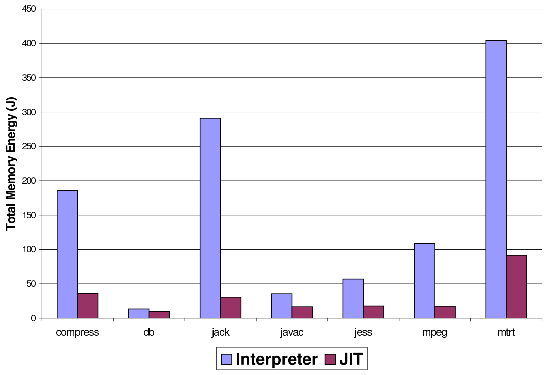

Figure 4: *A comparison of energy consumption when executing in JIT compile mode and interpreter mode. The Icache and Dcache are 32 KB two-way set associative and have 32 byte block size. s1 dataset was used.*

3.2 Hardware View – Impact of Cache Configuration

To investigate the influence of cache configurations on the energy consumption, we conducted experiments with different cache sizes and associativities. It is important to note that the energy consumption in the memory system is highly dependent on the number of cache misses as the off-chip memory energy cost is an order of magnitude larger than on-chip cache energy cost [37]. However, reducing the number of cache misses is normally achieved using caches of larger sizes or associativities that in turn add to the per access energy cost for the cache. Thus, it is essential to understand the energy trade-off between better cache locality and increased per access cost.

Figure 6 presents the energy spent in caches and memory for different cache configurations for `javac` and `db`. From these graphs, we can make the following observations. First, when the cache size is increased, we observe a decrease in the overall energy consumption up to a certain cache size. However, beyond a point the working set for instruction or data are contained in the cache, and a size larger than this does not help in improving the locality but does increase the per access cost due to the larger address decoder and larger capacitive load on the cache bit lines. A similar trend is also observed when we change the associativity for a fixed cache size, where increasing the associativity aggressively brings diminishing returns in energy saving. This effect is due to more complicated tag-matching hardware required to support higher associativities. Second, we observe that the instruction accesses seem to take advantage of larger caches better than data accesses. For example, in `javac` with the interpreter mode, when we move from a 4K direct-mapped cache to a 128K direct-

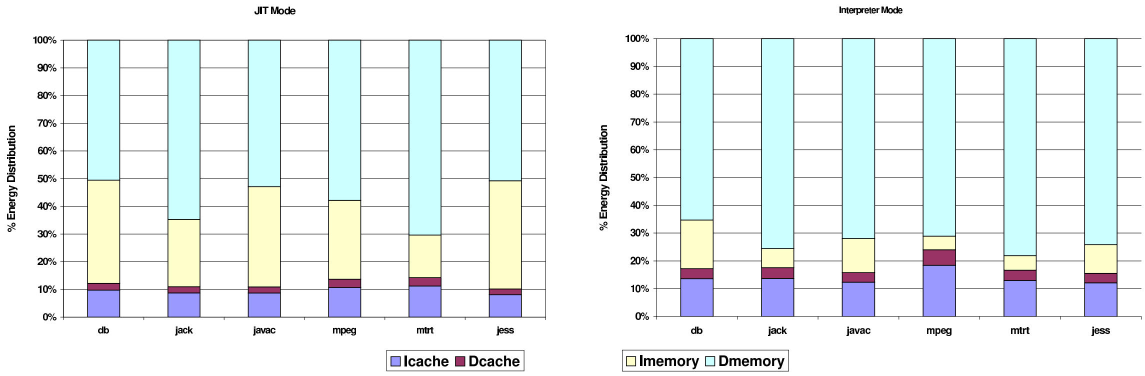

Figure 3: *Energy distribution for the JIT mode (left) and interpreter mode (right). The Icache and Dcache are 32 KB two-way set associative and have 32 byte block size. s1 dataset was used.*

Benchmark	**Interpreter Mode**								
	Icache		**Dcache**		**Imemory**		**Dmemory**		**Total**
	Acc.	**Eng.**	**Acc.**	**Eng.**	**Acc.**	**Eng.**	**Acc.**	**Eng.**	**Eng.**
`compress`	21,340.00	98.17	7211.6	33.17	0.6	0.77	42.4	53.71	185.82
`db`	450.5	1.82	118.6	0.48	1.8	2.34	6.9	8.74	13.38
`jack`	9,834.9	39.82	2,814.4	11.39	15.6	19.81	173.0	220.05	291.07
`javac`	1,076.3	4.36	301.1	1.22	3.4	4.34	20.0	25.44	35.36
`jess`	1,698.3	6.88	477.4	1.93	4.6	5.89	33.1	42.16	56.86
`mpeg`	4,730.6	19.15	1,432.3	5.80	4.0	5.06	58.1	73.87	108.88
`mtrt`	12,935.4	52.37	3,657.7	14.81	16.7	21.21	248.3	315.90	404.29
Benchmark	**JIT Mode**								
	Icache		**Dcache**		**Imemory**		**Dmemory**		**Total**
	Acc.	**Eng.**	**Acc.**	**Eng.**	**Acc.**	**Eng.**	**Acc.**	**Eng.**	**Eng.**
`compress`	922.4	4.24	330.9	1.52	1.8	2.22	22.2	28.11	36.09
`db`	238.2	0.97	59.5	0.24	2.9	3.70	3.9	5.00	9.91
`jack`	657.1	2.66	168.4	0.68	5.8	7.4	15.5	19.7	30.44
`javac`	353.1	1.43	88.4	0.36	4.7	5.95	6.8	8.68	16.42
`jess`	352.7	1.43	89.6	0.36	5.4	6.88	7.1	8.95	17.62
`mpeg`	456.0	1.85	129.0	0.52	3.9	4.94	7.9	10.02	17.33
`mtrt`	2,535.6	10.27	698.7	2.83	11.1	14.05	50.8	64.43	91.58

Figure 5: *Number of cache and memory accesses (denoted* **Acc.***) in millions and absolute energy values (denoted* **Eng.***) in Joules in interpreter and JIT mode.*

mapped cache, the instruction memory energy drops from 132.8J to 17.0J. On the other hand, except for the move from 4K to 8K, the data memory energy does not vary significantly across different cache configurations for the dataset size that is used in these experiments (`s1`). Finally, as far as the general energy trend is concerned, the JIT mode behaves similar to the interpreter mode except for the fact that the actual energy values are much smaller, less than half typically, and in some cases the cache configuration that results in the minimum energy consumption is different in the JIT mode from that of the interpreter mode.

It should be noted that although the number of memory accesses in the interpreter mode is higher than the JIT mode, the memory footprint of the former is smaller [23]. The increase in memory footprint for JIT compiler can be due to the additional libraries required for the JIT optimizations and dynamic code installation. For example, the SPARC and Intel versions of the JIT compiler proposed in [7] themselves require 176Kbytes and 120Kbytes. The influence of extra space required for compiled code in JIT mode is found to require 24% more memory space as compared to interpreter mode for the SPEC JVM98 benchmarks, on the average [23]. Consequently, in embedded environments where we have the flexibility of designing custom memory for a given application code [5, 2], we can potentially use a smaller memory for the interpreter mode of operation. In order to capture the effects of lower memory overheads due to the absence of dynamic compilation overheads, we scale the memory size of the interpreter relative to that of the JIT compilation mode. It must be noted that a smaller physical memory will incur less energy due to smaller decoders and less capacitive loading on the bitlines. We will assume that the energy cost of a memory

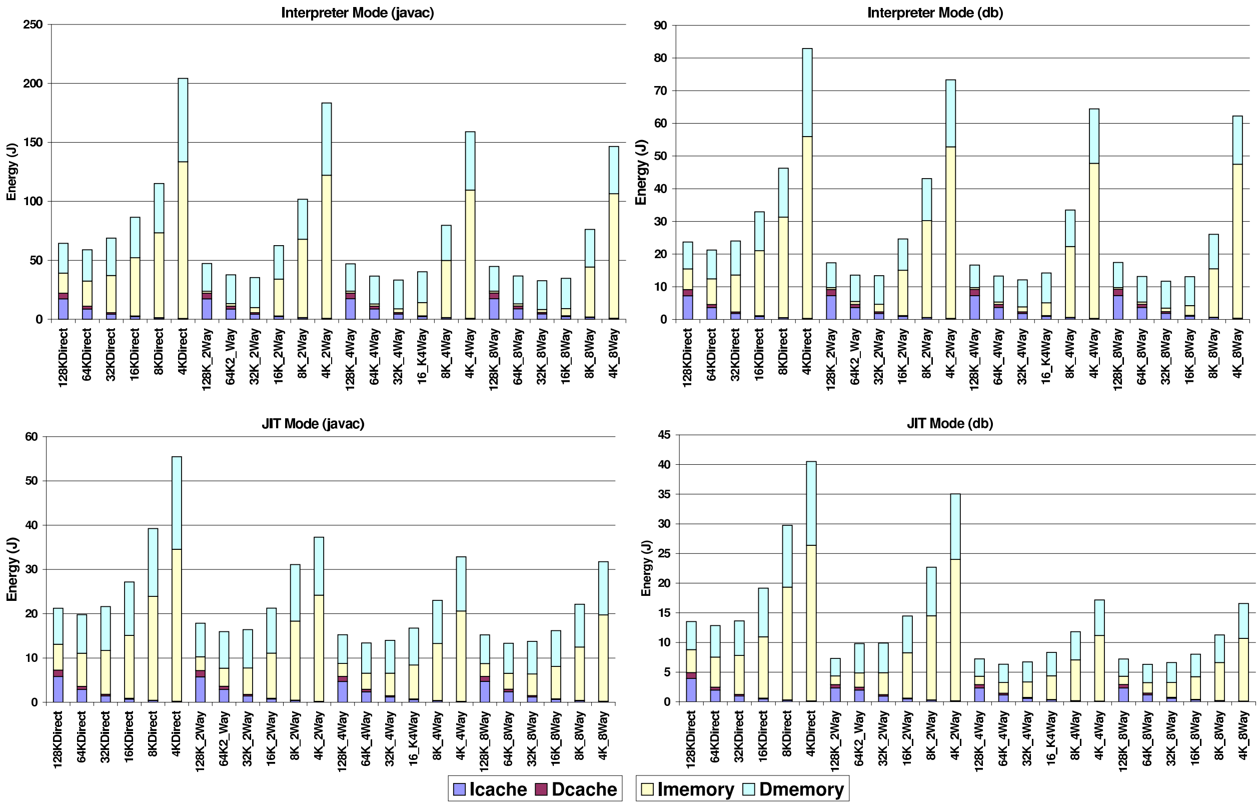

Figure 6: *Energy consumption (memory system) of* `javac` *and* `db` *for different cache configurations. The cache sizes and associativities on the x-axis are for both the instruction and data caches. s1 dataset was used.*

access decreases linearly with memory size for the purposes of this comparison.

Figure 7 gives the total energy consumptions for `db` for different ratios of memory footprint between the interpreter and JIT compiler. For the purposes of this approximation, we have neglected the effect of increased garbage collection overhead which would result when reducing the memory size. The way to interpret the graph is as follows. In `db`, the memory overhead of the JIT mode needs to be at least 1.67 (1/.6) times, (corresponding to 0.6 scaling factor) more than that of the interpreter mode before the interpreter becomes preferable from the energy viewpoint. Until then, JIT is preferable. The observed expansion in data segment for the JIT compilation mode is limited to 24% on an average [23] and the overhead of current JIT compilers is much smaller than the heap size (24 megabytes) needed for both modes. Hence, while one might think that reducing the memory size makes interpretation more attractive, the above observations show that the size expansion in JIT compilation mode is not significant enough to influence optimal energy consumption choice, even neglecting the increased GC overhead. However, if this footprint expansion becomes too large due to some JIT optimizations [38, 28, 21] that increase code size (e.g., in-lining), or the compiler becomes much larger compared to other resources such as heap space required in both modes one may need to re-evaluate this trade-off and select the suitable execution mode during memory system construction for an embedded device where physical memory space is limited.

Main memory has long been a major performance bottleneck and has attracted a lot of attention (e.g., [26]). Changes in process technology have made it possible to embed a DRAM on the same chip as the processor core. Initial results using embedded DRAM (eDRAM) show an order of magnitude reduction in energy cost per access [26]. Also, there have been significant changes in the DRAM interfaces that can potentially reduce the energy cost of external DRAMs. For example, unlike conventional DRAM memory subsystems that have multiple memory modules that are active for servicing data requests, the

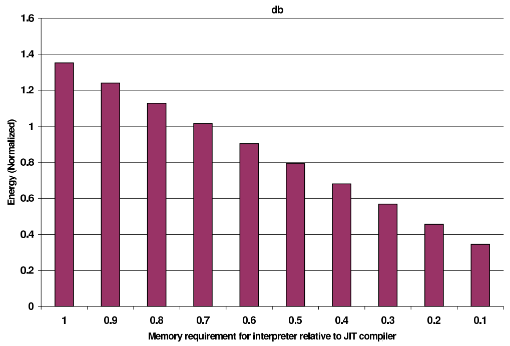

Figure 7: *Relative energy consumption of interpreters as compared to JIT Compiler. The memory size for the interpreter is varied relative to that of the JIT Compiler to capture the differences in the overheads associated with the storage associated with different compilers and the code expansion that can occur during native code generation and installation. An instruction cache and a data cache, both 32 KB, two-way associative with 32 byte block size, are used.*

direct RDRAM memory sub-system delivers the full bandwidth with only one RDRAM module active. Similarly, new technologies such as magnetic RAMs consume less than one hundredth the energy of conventional DRAMs [33]. Also, based on the particular low power operating modes that are supported by memory chips and based on how effectively they are utilized, the average energy cost per access for external DRAMs can be reduced by up to two orders of magnitude [8]. In order to study the influence of these trends, we performed another set of experiments using four different E_m values: 4.95×10^{-9}J (our default value), 2.45×10^{-9}J, 2.45×10^{-10}J, and 4.95×10^{-11}J. Each of these value might represent the per-access cost for a given memory technology. Figure 8 shows the *normalized* energy consumptions (with respect to the interpreter mode with the default E_m value). We observe that the ratio of the total memory energy consumed by the interpreter mode to that of the JIT mode varies between 1.05 (2.07) and 1.80 (2.85) for `db` (`javac`) depending on the E_m value used. We also observe that the relative difference between energy consumed in interpreter mode and JIT mode increases as E_m reduces due to better technologies. For example, the energy consumed in JIT compilation mode is half of that consumed in the interpreter mode for most energy-efficient memory while it is around 70% of interpreter energy for most energy consuming configuration when executing `db`. This indicates that the even when the process technology significantly improves in the future, the JIT will remain the choice of implementors from an energy perspective.

3.3 Software View – Energy Distribution across Software Components

In this part, we profile the energy consumption between different software components such as class loading (`load`), dynamic compilation (`compile`), garbage collection (`GC`), and the rest of the execution (`exec`). An understanding of this breakdown can provide the Java Virtual Machine (JVM) implementors and application designers with an indication of which components of their software consume the most energy and help them concentrate on optimizations to those particular components. For instance, if the energy cost due to object accesses is high, one could modify the object allocation strategies used by the JVM or apply object reuse strategies at the software level. Similarly, if the garbage collector [15, 29, 9] component is significant, one could opt for an improved algorithm or modulate the frequency of its invocation.

Figure 9 gives the energy distribution for the software components in both the interpreter and JIT modes. For example, `jack` executing the interpreter mode, the instruction accesses consume 60J and data accesses consume 232J. The corresponding energy numbers for the JIT mode are much lower at 10J and 20J respectively. These results are in consonance with the better locality of instruction accesses in the interpreter mode as discussed earlier. In the interpreter mode almost all the energy is spent in the interpretation and GC and class loading were found to be less than 2% of the overall energy consumption. Although execution takes the largest amount of energy in the JIT mode, the dynamic compilation also consumes a significant amount of energy. This is due to two main reasons [23]. First, there are abrupt changes in the working set during dynamic compilation as the code and data structures used by the compiler are different from that for the rest of the JVM. Thus, when we move to the code generation phase, we experience poor locality in the cache (data and instruction) accesses, and this in turn causes more references to the memory (both Imemory and Dmemory). Second, when code is installed af-

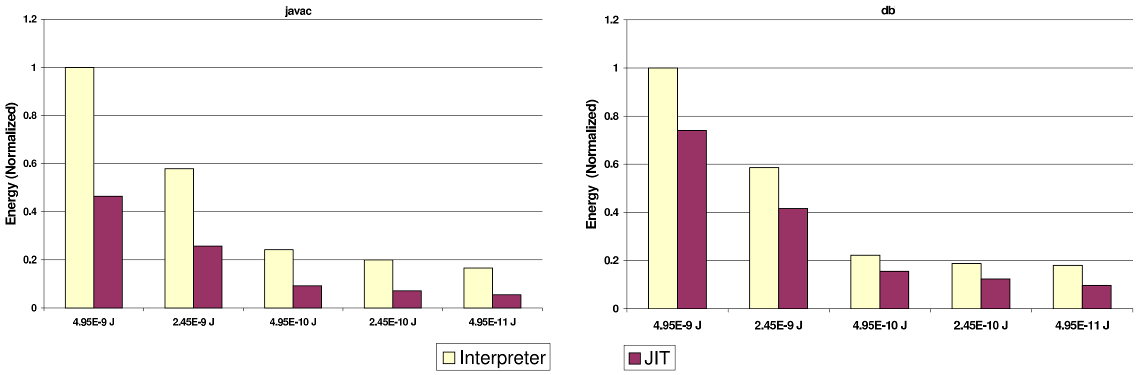

Figure 8: *Overall memory system energy consumption with different values of E_m for the interpreter and the JIT mode.*

ter dynamic compilation, it causes references to main memory. We observe that (in the JIT mode), on an average, the dynamic compilation consumes 24% of the overall energy across the benchmarks. Figure 11(d) breaks down the energy consumption by the hardware components during dynamic compilation and shows that Imemory and Dmemory are responsible for the bulk of the overhead.

In the rest of this discussion, we focus only on the JIT mode since the energy consumption for the interpreter is dominated entirely by the interpretation. Figure 10 gives the energy breakdown of `javac` into different software components with different cache configurations. We observe that (as opposed to class loading and garbage collection) the dynamic compilation and execution can take advantage of larger cache sizes. Data from other experiments [22] show that the energy consumption during loading is mainly dominated by compulsory misses. Hence, the number of total misses during loading is fairly constant across different cache configurations. However, there are small variations in energy consumption with changes in cache configuration as the per-access energy cost is affected not only by number of accesses but also by the energy cost of the tag-matching hardware and the capacitive load of bit lines. As can be observed in class loading profile in Figure 11(a), most of the energy is consumed by the data memory. It should be noted that some Java environments may be running multiple applications concurrently, in which some of the class loading costs can be amortized over the different applications [10].

We see from Figure 9 that the garbage collector consumes a very small fraction of the energy. Its energy consumption due to data accesses is higher than that due to instruction accesses as the garbage collector code itself is very small (i.e., good Icache locality) but the data accessed by the GC has a relatively poor locality. In fact, our detailed analysis shows that most of the energy expended in data memory is a result of the the cache misses. More innovation in improving the data locality of garbage collection will be valuable from energy perspective. While the absolute energy consumed by the garbage collector is small compared to overall execution in these experiments, we believe that the need for more aggressive garbage collection for limited memory embedded systems will make this component more important. It must be noted that the energy consumed in the garbage collection portion is also influenced by the choice of the algorithm and the size of the heap. The size of the heap can influence the number of times the garbage collector is invoked. For example, when we varied the heap size from 24M to 8M, the energy consumed by garbage collection increases eight fold when executing `mtrt` (s100 dataset and JIT compilation mode). The dataset of the application can also influence the energy consumed by the garbage collector. As an example, we found that the GC is responsible for nearly 14% of total data misses for `s100` data set (compared to 7% with `s10`) in the JIT mode, for `javac`, contributing to the overall energy more significantly. More detailed analysis of these tradeoffs in garbage collection energy consumption is beyond the scope of this work and is an interesting area of research in itself.

The execution of compiled code consumes the major chunk of the energy and Figure 11 shows the energy distribution for the different

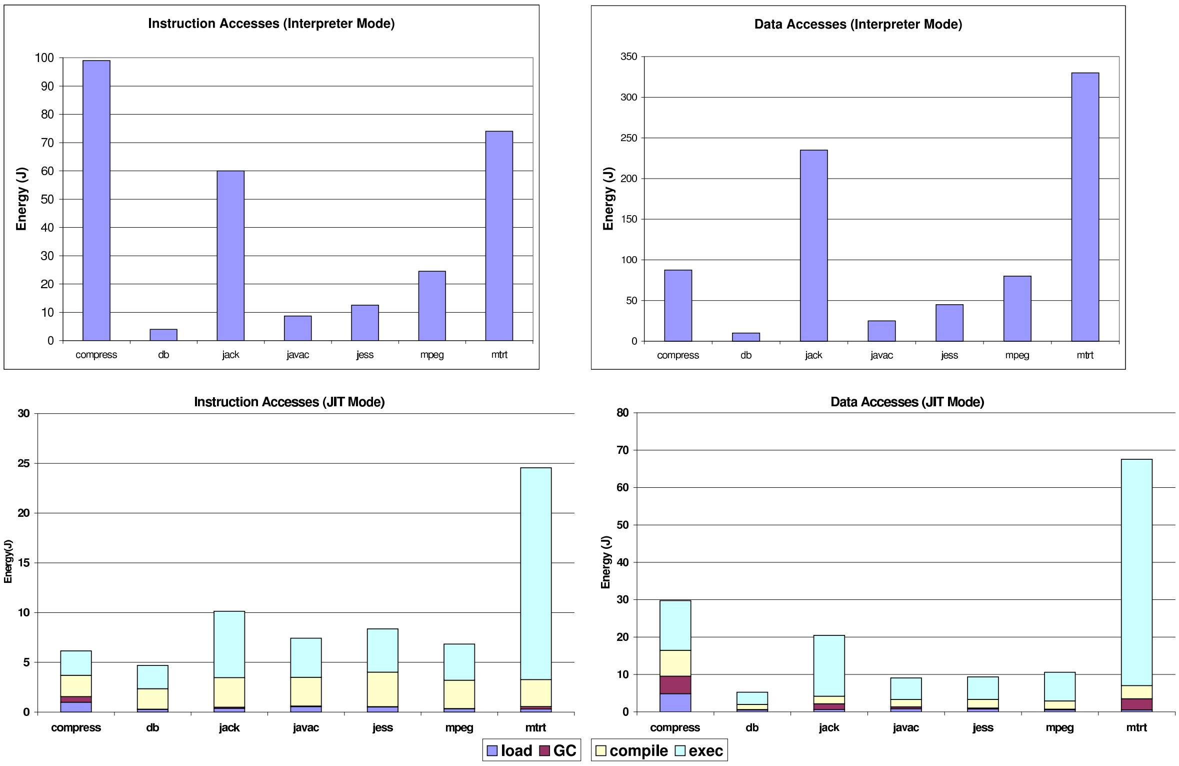

Figure 9: *Energy distribution based on software components. Instruction access energy involves Icache energy and Imemory energy, and data access energy involves Dcache energy and Dmemory energy. In the interpreter mode, the class loading and GC portions were small as compared to the actual interpretation of the code and hence are not shown separately in this breakdown. In the JIT mode, this component is mainly comprised of the energy spent in executing the native code after compilation.*

hardware and software components of the JIT mode. Overall, observing the trends shown in Figure 11, it is interesting to note that different applications in SPEC JVM98 exhibit different energy behaviors. For instance, while `mtrt` consumes the maximum energy during the execution phase, its energy consumption is smaller than that of `compress` during loading, garbage collection, and dynamic compilation. The energy consumption in different software components is a function of the number of classes loaded, the size of the classes, the number of methods compiled, the number of times a method is invoked after compilation, the heap size determining the frequency of GC invocation, the size of data set and the heap allocation, and memory access behavior during execution. Since the actual execution of the compiled code is the dominant component, we need to focus on developing techniques to reduce the energy consumed by this component. Optimizations during the JIT compilation phase (e.g., [28, 38]) can also potentially improve the energy efficiency of the execution phase, sometimes at the cost of increasing energy consumption due to dynamic compilation itself.

Finally, we would like to emphasize that the energy behavior in the different portions of the JVM is also dependent on the dataset size. We observe from Figure 12 that the share of class loading and dynamic compilation are comparatively smaller for the s100 dataset as compared to s10 dataset.

4 Concluding Remarks and Future Work

This paper has taken an important step towards the confluence of two emerging design considerations for ubiquitous and embedded computing: the need for a seamless and portable software platform for easy application design and interoperability, and the need for energy conscious system design. By focusing specifically on the Java runtime system and the SPEC JVM98 benchmarks, this paper has analyzed the energy consumption in the memory system for these appli-

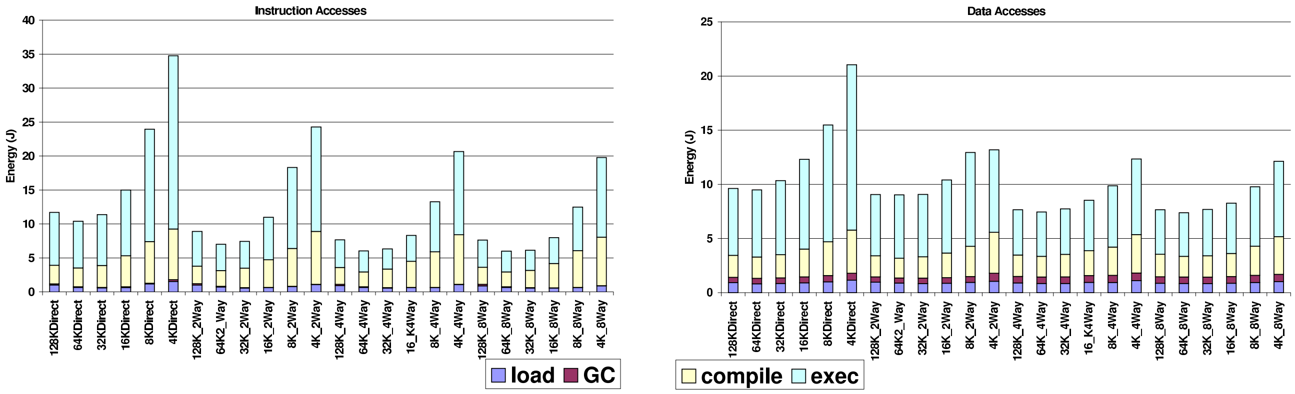

Figure 10: *Energy distribution (based on software components) for* `javac` *with different cache configurations (JIT mode).*

cations. The motivation for this study stems from the observation that instructions accessing the memory system account for over 50% of the energy consumption for these benchmarks. As applications get larger and become more data centric, they are likely to stress the memory system even more.

Using an off-the-shelf JVM, a validated energy model for the memory system, and a detailed simulator, this paper has presented a characterization of the energy consumption in the cache and main memory due to instruction and data references of the SPEC JVM98 benchmarks. The effect of the JVM implementation style (interpretation or JIT compilation) has also been studied, in addition to breaking down the energy consumption between different software components of the JVM — class loading, garbage collection, and dynamic compilation. The detailed profiles from this study can help towards hardware enhancements, in terms of cache and memory organization, and even algorithmic and software designs for energy conscious application and JVM designs.

This study has helped us make the following general observations:

- Main memory energy consumption is more dominant than that of the caches, with the main contributor being data references. Interpretation tends to stress the instruction cache more than JIT compilation, mainly because of the better locality. Overall, from the energy viewpoint, the JIT approach is a better alternative than interpretation.

- Interpretation has been a popular alternative for limited memory systems, since it requires less space than a JIT compiler. Lesser space, smaller memory, also implies a reduction in energy cost per access, which can be another argument one could use in favor of interpretation. However, our study reveals that the saving in energy per access due to smaller memory is not sufficient to compensate for the energy consumed by the larger number of accesses and longer execution time of interpretation compared to JIT.

- Cache organizations that favor locality can decrease the main memory energy consumption, while increasing the cost per cache access. These two contrasting influences make it necessary to decide on a good operating point for cache design that takes both locality and energy into account.

- In the interpreted mode, the energy for the actual interpretation of byte codes clearly dominates any other portion of the JVM. The energy consumed by the dynamic compilation in the JIT mode is quite significant, mainly due to the code installation and subsequent execution misses.

We believe the characterization study in this paper will be helpful for JVM implementors to understand the impact of their decisions on the energy consumption of the system. As this paper is one of the first attempts to characterize energy-behavior of the Java codes, we believe there is lot of scope for enhancements. First, we need to experiment more thoroughly on the effect of data set sizes and with different types of Java applications executing on mobile environments. Second,

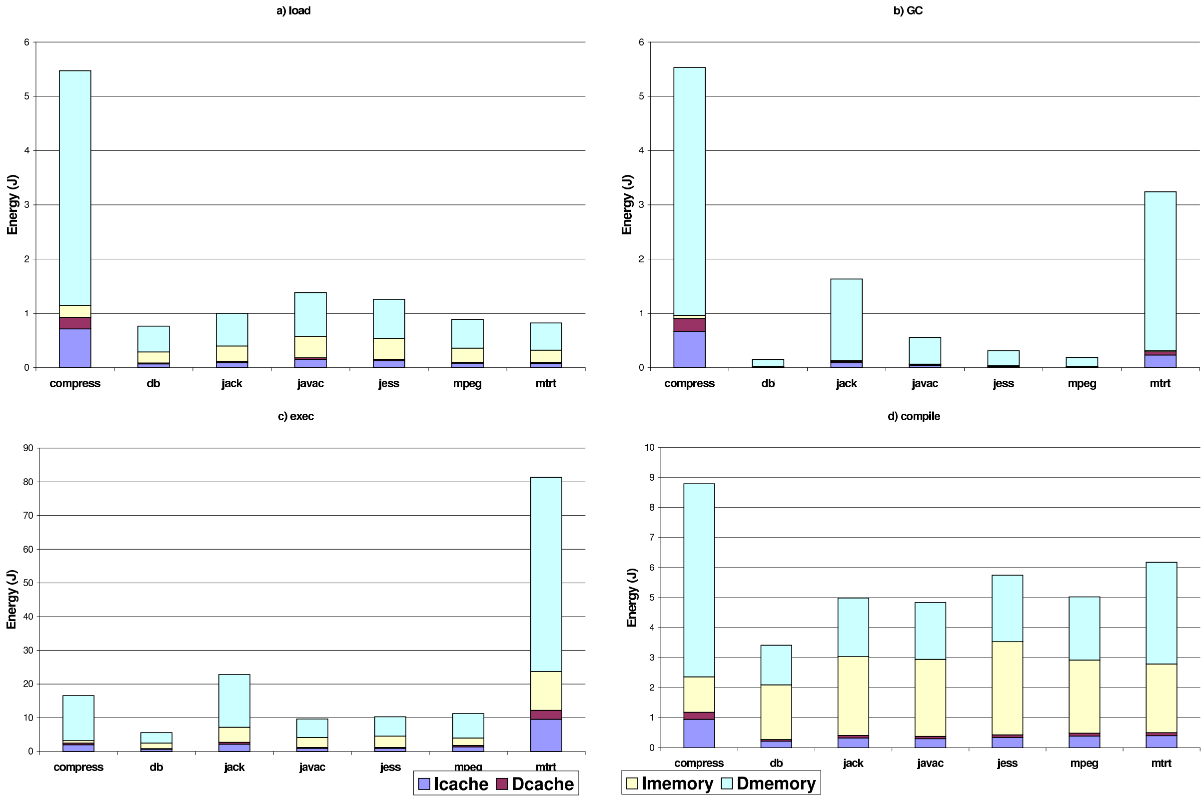

Figure 11: *Energy breakdown for different components. Note that the Y-axis scales for the different graphs are different. The Icache and Dcache are 32 KB two-way set associative and have 32 byte block size.*

we plan to study the impact of technology changes such as increased wire capacitances and leakage power on our study. We plan to address these in our future work.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language,* Addison-Wesley, 1998, Second Edition.

[2] F. Balasa, F. Catthoor, and H. De Man. Exact evaluation of memory area for multi-dimensional processing systems. In Proc. *the IEEE International Conference on Computer Aided Design,* Santa Clara, CA, pages 669–672, November 1993.

[3] B. D. Cahoon, and K. S. McKinley. Tolerating latency by prefetching Java objects. In Proc. *the Workshop on Hardware Support for Objects and Microarchitectures for Java,* October 10, 1999.

[4] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan. Global communication and memory optimizing transformations for low power signal processing systems. In Proc. *the IEEE Workshop on VLSI Signal Processing,* pages 178-187, 1994.

[5] 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.

[6] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In Proc. *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems,* pages 128–137, May 1994.

[7] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, M. Wolczko. Compiling Java just in time. IEEE Micro, 17(3), pp. 36-43, May-June 1997.

[8] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In Proc. *International Conference on High Performance Computer Architecture,* Jan 2000, To appear.

[9] S. Dieckmann and U. Holzle. A Study of the allocation behavior of the SpecJVM98 Java benchmarks. In Proc. *ECOOP'99,* 1999.

[10] J. Flinn, G. Back, J. Anderson, K. Farkas, and D. Grunwald. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In Proc. *International Conference on Measurement and Modeling of Computer Systems,* Santa Clara, California, June 17-21, 2000.

[11] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation. In Proc. *1999 International Symposium Low Power Electronics and Design,* 1999, pages 70–75.

[12] R. Gonzales and M. Horowitz. Energy dissipation in general purpose processors. *IEEE Journal of Solid-State Circuits,* 31(9):1277–1283, Sept 1996.

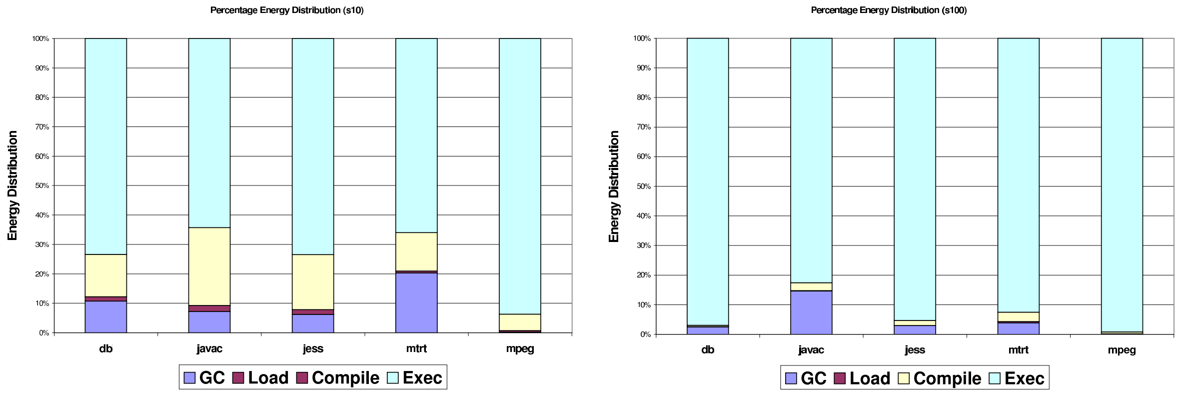

Figure 12: *Energy distribution (based on software components) for five of the benchmarks for (a) s10 (b) s100 dataset. 16K 2-way associative data and instruction cache were used.*

[13] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the desing of the Alpha 21264 microprocessor. In Proc. *the Design Automation Conference,* San Francisco, CA, 1998.

[14] M. J. Irwin and N. Vijaykrishnan. Low-power design: From soup to nuts. Tutorial Notes, *ISCA,* 2000.

[15] R. Jones and R. Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[16] M. B. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *International Symposium on Low Power Electronics and Design,* pages 143–148, 1997.

[17] Jin-Soo Kim and Yarsun Hsu. Memory system behavior of Java programs: Methodology and analysis. In Proc. *International Conference on Measurement and Modeling of Computer Systems,* Santa Clara, California, June 17-21, 2000.

[18] A. Krall. Efficient Java VM Just-in-Time compilation. In Proc. *Parallel Architectures and Compilation Techniques,* Paris, France, 1998.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification,* Addison Wesley, 1997.

[20] H. McGhan and M. O'Connor. PicoJava: A direct execution engine for Java bytecode. *IEEE Computer,* pp. 22–30, October 1998.

[21] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java Programming for High-Performance Numerical Computing. *IBM Systems Journal,* 39(2), 2000.

[22] Anupama Murthy. *Memory System Characterization of Java Applications,* Masters Thesis, Dept. of Computer Science and Engineering, The Pennsylvania State University, May 2000.

[23] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural issues in Java run-time systems. In Proc. *International Conference on High Performance Computer Architecture,* pages 387-398, January 2000.

[24] K. Roy and M. C. Johnson. Software design for low power. *Low Power Design in Deep Submicron Electronics,* Kluwer Academic Press, October 1996, Edt. J. Mermet and W. Nebel, pp. 433–459.

[25] W-T. Shiue and C. Chakrabarti, Memory exploration for low power, embedded systems, CLPE-TR-9-1999-20, Technical Report, Center for Low Power Electronics, Arizona State University, 1999.

[26] P. Song. Embedded DRAM finds growing niche. *Microprocessor Report,* pages 19–23, August 4, 1997.

[27] *SpecJVM98 Benchmarks.* http://www.spec.org/osg/jvm98

[28] V. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In Proc. *PLDI,* Van Couver, Canada, 2000.

[29] J. M. Stichnoth, G-Y. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a java compiler. In Proc. *Programming Language Design and Implementation,* Atlanta, 1999.

[30] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: A case study, In Proc. *International Symposium on Low Power Electronics and Design,* pp. 63–68, 1995.

[31] *The Sun Labs Virtual Machine for Research.* http://www.sun.com/research/java-topics

[32] Ali-Reza Adl-Tabatabai et al. Fast, effective code generation in a Just-in-Time Java compiler. In Proc. *PLDI,* 1998.

[33] Memories are Forever, *Technology Review,* May-June 2000, pp. 28.

[34] M. C. Toburen. Power Analysis and Instruction Scheduling for Reduced di/dt in the Execution Core of High-Performance Microprocessors, *Master's Thesis,* Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, May 1999.

[35] http://www.transmeta.com/articles/

[36] N. Vijaykrishnan. *Issues in the Design of a Java Processor Architecture.* Ph.D. thesis, College of Engineering, University of South Florida, 1998.

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

[38] B-S. Yang et. al. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In Proc. *PACT'99,* California, October 1999.

[39] W. Ye. *Architectural Level Power Estimation and Experimentation.* Ph.D. Thesis, Department of Computer Science and Engineering, The Pennsylvania State University, October 1999.