Techniques for Real-System Characterization of Java Virtual Machine Energy and Power Behavior

Gilberto Contreras, Margaret Martonosi
Department of Electrical Engineering
Princeton University
{gcontrer,mrm}@princeton.edu

Abstract— **The Java platform has been adopted in a wide variety of systems ranging from portable embedded devices to high-end commercial servers. As energy, power dissipation, and thermal challenges begin to affect all design spaces, Java virtual machines will need to evolve in order to respond to these and other emerging issues. Developing a power-conscious Java runtime system begins with a detailed per-component understanding of the energy, performance and power behavior of the system, as well as each component's impact on overall application execution.**

This paper presents techniques for characterizing Java power and performance, as well as results from applying these techniques to the Jikes RVM, for some of the most salient Java virtual machine components. Components studied include the garbage collector, the class loader, and the runtime compilation subsystem. Real-system measurements with our efficient, low-perturbation infrastructure offer valuable insights that can aid virtual machine designers in improving energy-efficiency. For example, our results show that JVM energy consumption can comprise as much as 60% of the total energy consumed. In addition, we find that generational garbage collectors offer the best energy-performance for small heap sizes and that this efficiency is challenged by non-generational collectors for large heaps. Overall, given the rising importance of Java systems and of power/thermal challenges, this paper's detailed real-systems examination can lend useful insights for many real-world systems.

I. Introduction

Java runtime environments are used in many systems with wide-ranging requirements. Java's popularity has created a wave of research heavily focused on improving the performance not only of the Java applications themselves, but also of the underlying virtual machine (software). Many components of the Java virtual machine, especially the garbage collector, have been the focus of performance optimization and system-resource usage [1][2][3]. Garbage collection in particular has attracted a considerable amount of research since the memory subsystem is in the critical path of most user applications.

While performance is clearly important, it is not the only metric to consider. Energy, power, and thermal issues are now seen in essentially all computer design spaces. Battery energy, for example, is a valuable resource in portable embedded systems; it must be conserved while still achieving the necessary performance. Maximum power consumption on the other hand is becoming the limiting factor for creating denser high-performance computer servers and clusters due to costly cooling equipment and unreliable power supplies [4]. Tightly associated with power consumption are temperature and thermal-related issues. Today's data center servers, for example, must be available 24 hours a day with an availability exceeding 99.99%, independent of the server load [5]. Achieving such reliability is many times thwarted by high operating temperatures of devices. If the cooling system (the heat sink and/or the packaging in case of modern microprocessors) cannot remove heat fast enough, excessive temperatures automatically trigger the processor's emergency thermal responses, such as throttling [6], and consequently, a decrease in performance. As an example, Figure 1 shows two scenarios on a real-word system. The first scenario represents normal operation conditions (Fan enabled). The second scenario shows processor temperature in the event of fan failure (Fan disabled). Under normal operation the processor's temperature remains fairly constant around 60°C. When the processor's fan is disabled however, executing the workload increases the temperature up to 99°C after 240 seconds. High temperatures trigger the processor's thermal emergency response, which reduces the clock duty cycle to 50%, proportionally decreasing performance.

There has been little prior work in studying power and energy requirements of JVMs, and what exists has mostly used simulation for gathering its results [7][8][9]. While simulation can in fact offer significant insight and important observations, the problem is that fast simulation techniques lack the comprehensiveness and accuracy provided by real-hardware measurements. More comprehensive simulation techniques are often quite slow, which prevents them from gaining wide use. For example, many existing simulators abstract operating system effects and I/O—critical factors that may affect the power/performance of the JVM. This abstraction is usually done to reduce the complexity of the simulator and increase its performance. However, it does so at the expense of a less accurate model. Full-system simulators that do capture OS and I/O effects such as SimOS [10] do so at the expense of simulation time, making them unpopular for detailed analysis of complex applications.

To accurately characterize energy and power consumption of Java virtual machines at real-hardware speeds, we develop a flexible infrastructure based on physical real-hardware power and performance measurements. Our methodology accounts for processor and main memory power consumption in order to provide a system-wide characterization of Java virtual machines and their corresponding running applications. We have applied this methodology to two popular Java virtual machines: IBM Jikes RVM [11] and Kaffe [12].

Our work not only is concerned with overall JVM power and energy behavior, but also with the behavior of important software sub-components of the virtual machine, such as the garbage collector, the dynamic class loader, and the runtime compiler subsystem. For the Jikes RVM, we compare power and energy requirements of the virtual machine while varying the heap size and the type of garbage collection being used. For Kaffe, we perform JVM power and performance analysis on two very different systems: a Pentium M based system, where it is compared with the Jikes RVM, and an Intel XScale-based development board, where we examine power and performance behavior differences seen on embedded platforms.

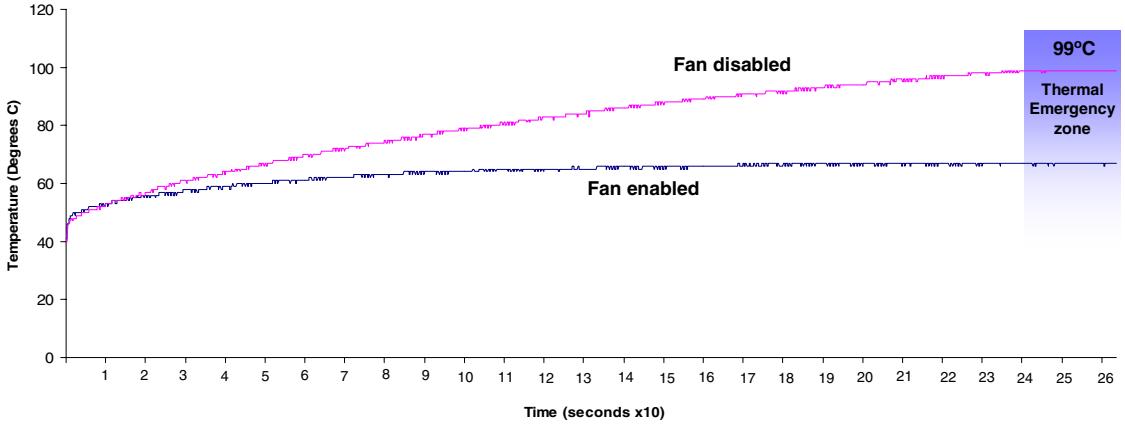

Fig. 1. Temperature behavior for a 1.6Ghz Pentium M processor running repetitive runs of `_222_mpegaudio` on the Jikes RVM using a generational copying collector. When the processor reaches $99°C$ it enters emergency throttling as a way to reduce chip temperature.

Here we list some of the major contributions of our work:

- We present a methodology that accounts for processor and main memory power consumption through the use of physical power measurements.
- We provide a per-component energy and power decomposition for two important Java virtual machines using two different platforms.
- We study the effect of the garbage collector algorithm and heap size on two JVMs. We demonstrate that the energy usage of the Java virtual machine can be as high as 60% of the total energy consumed.
- The work presented in this paper sets the stage for future in-depth physical characterization of Java virtual machines and their corresponding workloads in the context of emerging device design issues such as energy, power, and thermal characteristics.

Java virtual machines have the potential to dynamically manage not only application performance through aggressive online optimizations, but also to dynamically adapt to wider system changes such as energy availability, power consumption and even temperature variations. While fully addressing these optimizations is outside the scope of this study, we present a physical characterization of energy and power for modern virtual machines as a starting point.

The rest of the paper is structured as follows: Section II describes previous research in power and performance analysis of Java virtual machines, covering methodologies based on simulation and physical measurements. Subject background and terminology is given in Section III. Section IV describes our physical measurements methodology in detail. Section V describes the benchmarks used for this study. Section VI discusses our results, starting with Jikes RVM and Kaffe on a Pentium-based system and later with Kaffe running on an embedded processor. Future work is described in Section VII. We offer our summary and conclusions in Section VIII.

II. Previous Research

Research in the area of power, energy, and thermal requirements has gained considerable momentum due to the growing importance of these factors. Moreover, Java virtual machines and the behavior of applications running on them have received much attention since Java was first introduced.

For previous research, simulation has been the primary evaluation tool [13][14][15], and while simulation studies do provide important insights, they tend to omit or abstract real-system effects involving the OS and/or I/O, causing them to lack much of the detail of real systems.

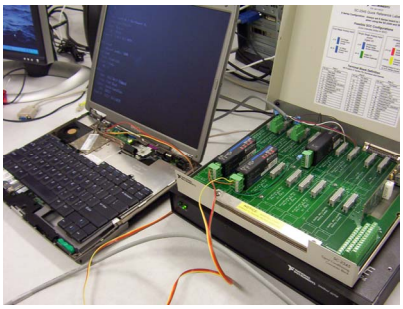

Fig. 2. This figure shows our P6 platform, which consists of a Pentium M 1.6GHz development board computer. We use this board to acquire physical measurements consisting of energy, power, and performance in the study of Java virtual machines.

In the past, simulation-based power and energy studies in the context of Java have been presented by Chen et al. [7] and Vijaykrishnan et al. [9]. Chen et al. presents an adaptive-garbage collector that disables certain memory banks when they are not being used by any allocated object. Vijaykrishnan et al. performs an in-depth study of memory energy requirements of Java applications. Here, the JVM is divided into three basic components: class loading, dynamic method compilation and garbage collection. For these three components, energy consumption of the memory-hierarchy is studied at various instruction and data cache configurations. Our study draws inspiration from this work in the sense that we also partition the JVM into its basic software components. However, our work utilizes physical measurements as the driving force of our study rather than a simulation approach.

On the hardware-side, we find several works with performance as their main goal, ranging from studies utilizing hardware performance counters to identify sources of program performance loss [16][17][18] to studies concerned with the algorithmic performance of garbage collectors [16]. Hardware-based studies of power and energy requirements have been mostly used in non-Java applications, such as in the case of *PowerScope* [19]. PowerScope is hardware/software tool capable of examining real-system software components through software-initiated statistical sampling of power. While flexible, PowerScope requires software components to have relatively

long execution times, potentially missing important fine-grain components.

Farkas et al. [20] performed pioneering work in the study of actual power consumption of a Java virtual machine running on a pocket computer. In this study, the authors investigate the energy implications of using multiple JVMs, the use of compressed and uncompressed JAR files, class loader strategies and the use of an interpreter versus a just-in-time mode JVM. Diwan et al. [21] analyze the energy consumption of different memory managing strategies with a platform similar to [20]. While Java-based, these two previous works do not address the energy requirements of high-performance virtual machines. Our study provides detailed power measurement results and a side-by-side comparison for both a high-performance platform and an embedded device platform.

III. Overview and Background

A. Metrics: Energy and Power

Energy, measured in Joules (J), is defined as the capacity to do useful work. In electrical devices, energy refers to charge movement and electrical current. The amount of battery capacity for an embedded device, for example, refers to the amount of energy it can store and then discharge. Power, measured in Watts (W) is defined as the time rate at which energy is consumed. Thus, the integral of power over time is total energy. Power is an important metric for thermal and reliability issues, since dissipating energy quickly (high power) can cause dangerously high temperatures.

When comparing the overall quality of two devices or software algorithms running on the same device, the metrics *energy consumption* and *power consumption* alone are not sufficient since they do not account for performance need. For example, a processor can consume very little power by running at very low frequencies but its performance will suffer. Similarly, an aggressive power-hungry processor might have good performance at the expense of very high power consumption. The metric *energy-delay* product (EDP) [22] is used to convey the combined attributes of energy and performance. EDP is the product of the total energy consumed by the running application and the time it takes to execute ($Joules \times seconds$). This favors low energy and low execution times. This way the slow-running processor will be penalized by its slow execution time and the aggressive processor will be penalized by its high energy consumption.

For devices with very stringent thermal requirements, peak power consumption becomes an important metric, as it indicates the maximum amount of power demanded by the device. Exceeding the maximum rated power consumption can cause device failure. Modern processors are designed under a safety power envelope. Knowledge of maximum power demanded by an expected workload, however, can yield devices with more relaxed power constraints, which in turn leads to less expensive packaging [23].

B. Java Terminology and Background

We can classify the garbage collection algorithms used in this study into two spaces as shown in Figure 3. In the non-generational space, we have the *SemiSpace* and *MarkSweep* garbage collectors [24]. The SemiSpace garbage collector works by dividing the heap into two sections. When the applications request memory from the virtual machine, the memory allocator uses the first half of the heap to allocate the new object. When the space in that half runs low and a new object cannot be allocated, garbage collection is initiated on that half of the heap and surviving objects are copied to the other half. After garbage collection, the roles of the semi-spaces are inverted. *MarkSweep* implements a Mark-and-Sweep collector, which uses a list of available fixed-size memory chunks to allocate new objects. Unlike a copying collector (such as SemiSpace), MarkSweep does not move objects around. Garbage collection is initiated when the allocator cannot find a memory chunk big enough to allocate a new object.

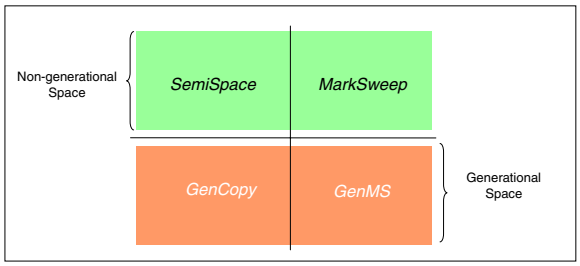

Fig. 3. We study four types of garbage collection algorithms in our study of the IBM Jikes RVM. This figure shows how garbage collectors vary according to the algorithm employed.

The bottom space of Figure 3 shows the generational collector space. These collectors work by allocating new objects into a specific region in the heap called the *nursery*. When the space in the nursery is exhausted, the nursery is garbage-collected and the surviving objects are moved to a *mature space* region in the heap. Generational collectors are based on the principle that older objects, or older generations, have higher probability of surviving, thus the mature space is garbage-collected less often since objects that die quickly release memory space in the nursery. Generational collectors differ in the way the mature space is collected. For *GenCopy*, a semi-space copying algorithm is used. For *GenMS*, a mark-and-sweep algorithm is employed.

The garbage collector plays an important role in the overall performance of Java applications as short garbage collection times reduce the overall application execution time. Consequently, the type of garbage collector being used also has a deep impact on the energy requirements of the overall system. In Section VI we characterize this energy requirement based on the type of garbage collector being employed and the amount of heap memory available.

IV. Methodology

This section describes the methodology used for obtaining physical power and performance measurements on a component-basis for two selected Java machines.

A. The IBM Jikes RVM and Kaffe Virtual Machines

For our physical power measurements, we use two distinct Java virtual machines: The IBM Jikes RVM version 2.4.1 [11], and Kaffe version 1.1.4 [12]. The IBM Jikes RVM is a high-performance virtual machine targeted for high-performance processors. It uses an adaptive runtime system that dynamically selects methods as candidates for higher compiler optimizations. When a method is loaded for the first time, a fast but simple baseline compiler is used to translate the Java bytecodes. Later, when a method is labeled “hot” by the adaptive system, the virtual machine determines if recompiling the method with higher (and costly) optimizations

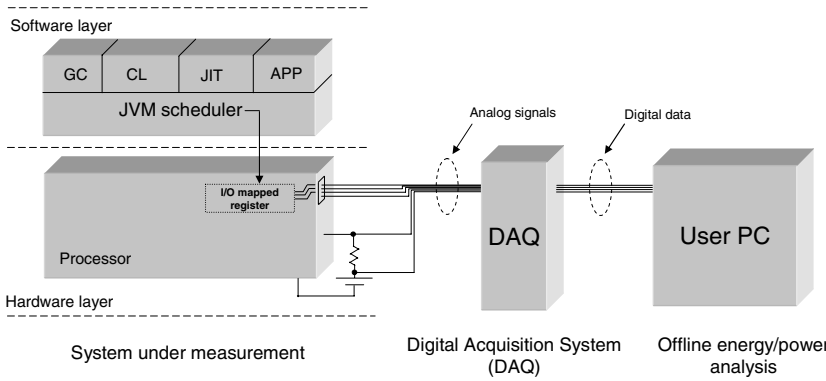

Fig. 4. Block diagram showing the three major components of our measurement infrastructure. The running JVM identifies the current running component by writing to an I/O mapped register (left). This ID, along with power measurements, are sampled by a high-speed data acquisition system (middle). Per-component energy and power behavior is analyzed offline, where it is matched with performance traces (right).

levels improves performance [25]. The Jikes RVM supports a multitude of garbage collectors including generational and non-generational collectors. We perform power and performance measurements on the four types of collectors previously discussed using fixed heap sizes of 32, 48, 64, 80, 96, 112 and 128MB.

The Kaffe JVM is a clean-room, open-source Java virtual machine with support for a wide variety of architectures. Because of its modularity and flexibility, Kaffe has been ported to many systems, including ARM-based processors found in many embedded portable devices. Kaffe can be configured as an interpreter machine, or with *Just-In-Time* (JIT) compiler support. It uses an incremental conservative mark-and-sweep, three-color garbage collector in conjunction with dynamic lazy class-loading for user and system classes. For this work we use the JIT version of Kaffe with Unix threads.

We compiled Kaffe for two different architectures: x86 and ARM. We did so because we want to investigate major power and performance differences of JVM components between two different platforms. We could not do the same for Jikes RVM as it currently does not have support for the ARM instruction set architecture.

B. Hardware Platforms

We use two different platforms for our experiments:

Pentium M development board (P6): The board consists of a 1.6GHz Pentium M processor and 512MB of RAM. This system was selected for two reasons: (i) the Pentium M processors is a performance processor designed under a stringent power (and thermal) environment while still offering high-performance, and (ii) physical power measurements are possible due to exposed components. The Pentium M processor contains an on-chip primary 32KB instruction cache and a 32KB write-back data cache. The on-die Level 2 (L2) cache is 1MB. The operating system is a custom-compiled 2.6.11 Linux kernel. Figure 2 shows a picture of this setup.

DBPXA255 development board (DBPXA255): For comparing JVM power and performance behavior differences between a high-performance platform and an embedded processor, we use the Intel DBPXA255 development board [26], which consists of Intel's PXA255 microcontroller running at 400MHz. The PXA255 microcontroller contains a single-issue in-order processor with a 32-way 32KB instruction cache, a 32-way 32KB data cache, and no L2 cache. The DBPXA255 development board contains many of the features commonly found in modern portable devices such as a high-resolution LCD display, a MultiMediaCard (MMC) slot, FLASH memory (64MB) and 64MB of SDRAM. For this board, the operating system is the Linux 2.4.19 kernel, which at the time was the only supported kernel of our development board.

C. Monitoring of JVM components

An important feature of our physical measurement methodology is the ability to monitor when a particular virtual machine component is running. Although we are not the first to divide the virtual machine into various unique components, we are the first to make such distinction within a physical power measurement infrastructure.

In simulation environment, tracking the execution of JVM components can easily be done by keeping track of the current instruction pointer (if the code layout of the JVM is known) or by allowing each component to set a global flag upon taking control of the processor. We employ the latter, except that rather than saving the component *ID* to a variable in main memory, we save it to a memory-mapped I/O register. The I/O implementations of the two platforms call for slight differences in our measurement setup. For the DBPXA255, we use general-purpose processor pins that are accessible to the user. The P6 platform, however, does not have these pins available, so instead we use the parallel port due to its simplicity of use and availability.

Hardware identification of software components requires modifications to our JVMs. For Kaffe, placing entry and exit system calls for each component serves this purpose. However, we have to be careful in covering cases of recurrent or overlapping component calls. Jikes RVM requires a different approach, as some of its services run on different threads, such as the optimizing compiler. For Jikes RVM, we place identification calls within the thread scheduler so that when the scheduler schedules the garbage collector thread, for example, its unique identification code is visible at the terminals of the parallel port.

D. Power Measurements

The power consumption of a device can be calculated using the equation $P = VI$, where V is the voltage of the device and I is the current drawn by the device. Thus, for our power measurements, we need to keep track of both voltage and current. The voltage at the V_{cc} pins of the processor and

Suite	Benchmark	Description
SpecJVM98	_201_compress	A modified Lempel-Ziv compression algorithm
	_202_jess	A Java Expert Shell System
	_209_db	Database application working on a memory-resident database
	_213_javac	A Java compiler based on SDK 1.02
	_222_mpegaudio	Audio decoder based on the ISO MPEG Layer-3 standard
	_227_mtrt	Raytracing application
	_228_jack	A Java Parser generator
DaCapo	antlr	A grammar parser generator
	fop	Application that generates a PDF file from an XSL-FO file
	jython	Python program interpreter
	pmd	An analyzer for Java classes
	ps	A Postscript file reader and interpreter
Java Grande Forum	euler	Benchmark on computational fluid dynamics
	moldyn	A molecular dynamic simulator
	raytracer	A 3D raytracer
	search	An Alpha-Beta prune search

Fig. 5. Our study uses a wide selection of applications taken from SpecJVM98, DaCapo and Java Grande Forum benchmark suites to study power and energy behavior of JVM software components.

memory is fairly straightforward to measure. Current, however, needs to be measured indirectly.

Current consumption of our P6 platform is measurable via two precision resistors placed in series between the voltage supply of the processor and its voltage pins (corresponding to the power lines that feed only the logic structures of the processor, not its I/O pins). These precision resistors allow us to measure the voltage drop across the resistors and thus indirectly measure the current being drawn by the processor. Once voltage and current consumption are known and sampled every 40μs (the fastest sampling rate of our digital acquisition system based on the number of sampling channels used), we multiply these values to obtain instantaneous power consumption. At each sampling point we examine the memory-mapped register and assign the measured power consumption to the corresponding component. This approach places a 40μs measurement window on all power measurements: transient changes inside the 40μs window are not captured by our system, nor do we keep track of when exactly a component switch happens. Since typical component duration is hundreds of micro-seconds on our P6 system and milliseconds on our PXA255 system, our sampling fidelity accurately captures all important behavior.

RAM Power consumption is measured using a similar sampling approach: we place precision resistors in series with the main voltage supply line, from which we measure a voltage drop to calculate the amount of current flowing through them. The idle processor and memory power consumption of our P6 platform is about 4.5W and 250mW respectively.

The DBPXA255 development board is specifically designed for testing and probing purposes. System voltages, including the processor's power lines, are exposed. Processor voltage can be taken by direct measurement. The measured idle processor power consumption of the Intel PXA255 is close to 70mW. Memory power consumption is near 5mW.

E. Performance Measurements

Our system performance measurements are obtained using the processor's hardware performance monitors (HPM). Our HPM application programming interface (API) is custom-made to identify Java virtual machine component execution. In our setup, the operating system's main timer is responsible for taking periodic samples (every 1ms in our P6 platform and 10ms in the DBPXA255) of anything that is running on the processor. We keep track of JVM component execution by placing a system call at the start of the JVM component that informs the OS what JVM component is currently executing.

Figure 4 shows a block diagram of our measurement infrastructure. To the left of the figure we find the system under test, which is running one of our Java virtual machines.

V. Applications

We perform our experiments on three different benchmark suites.

SpecJVM98: We perform measurements on seven SpecJVM98 [27] applications: `_201_compress`, `_202_jess`, `_209_db`, `_213_javac`, `_222_mpegaudio`, `_227_mtrt` and `_228_jack`. Benchmarks are executed using the full dataset (`--s100`).

DaCapo: The DaCapo benchmark suite [28] is a collection of memory-intensive applications typically used in the study of Java garbage collectors. We utilize version beta051009 and employ five applications from this suite: `antlr`, `fop`, `jython`, `pmd`, and `ps`. We use the default data set for all.

Java Grande Forum: Our third set of applications comes from the Java Grande Forum (JGF) Benchmark Suite [29], which is comprised of a series of benchmarks ranging from small kernels to full-blown applications. We utilize four sequential benchmarks from the suite, `euler`, `moldyn`, `raytracer`, and `search`, and utilize dataset A for all our tests.

All applications are run until completion. Power and performance measurements are performed in different runs. For both cases, a warm-up run of the benchmark is performed before taking power or performance measurements. Figure 5 summarizes our benchmark selection.

VI. Physical Measurements Results

As mentioned in Section IV, our methodology can distinguish basic JVM services (components), allowing us to accurately quantify the execution time and energy consumption of each monitored component. The following section presents

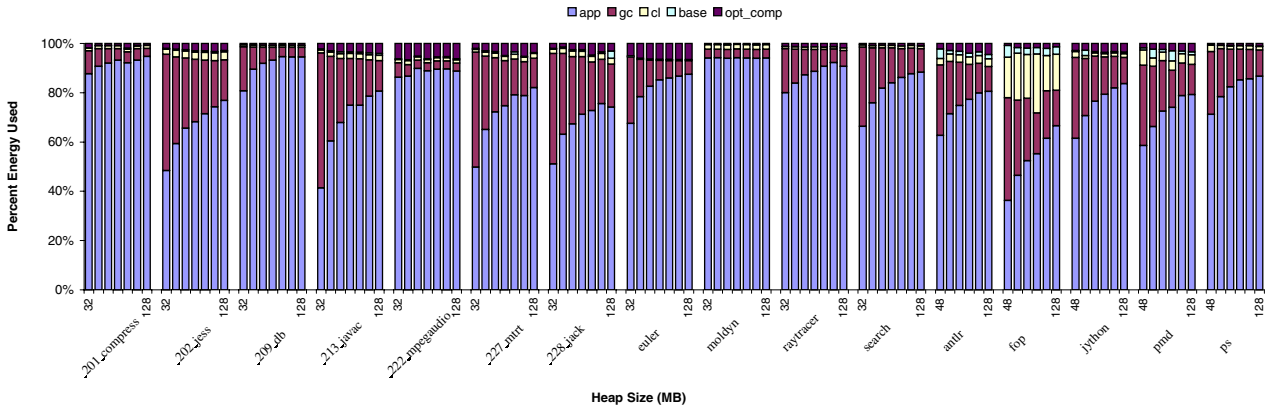

Fig. 6. Energy decomposition for SpecJVM98, DaCapo and JGF benchmarks under Jikes RVM using a SemiSpace collector. Overall, the garbage collector is one of the highest energy consumers.

results for four of the Jikes RVM components: garbage collection (GC), class loader (CL), Base Compiler (Base), and the optimizing compiler (Opt). The collective energy consumed by these components is classified as the energy consumed by the virtual machine (JVM energy). The rest of the energy consumed by the benchmark is classified as *application* energy. Clearly, the JVM energy is an underestimate of the real energy consumed by the JVM as there are other components within the JVM that we do not monitor, such as thread creation, scheduling, and the controller thread in the case of the Jikes RVM. For this last component, we monitored its execution time and energy requirements and determined that it was not a critical component in terms of these requirements. Its execution time accounted for less than 1% of the total benchmark execution time. For Kaffe we present results for three components: the garbage collector (GC), the class loader (CL) and the JIT compiler (JIT).

A. Energy Requirements of JVM components

Figure 6 shows the percent of total processor energy used for the optimizing compiler (opt_comp), the base compiler (base_comp), the class loader (CL), the garbage collector (GC), and the application (App) for the Jikes RVM using the SemiSpace garbage collection algorithm. A very high percentage of energy used (up to 60% for _213_javac using a 32MB heap) is dedicated to virtual machines components. The garbage collector in particular stands out from other JVM components as it can consume up to an average of 37% of the total energy for SpecJVM98 benchmarks when the heap size is 32MB. This value is reduced to 10% with a heap size of 128MB. Similarly, for DaCapo benchmarks, the garbage collector accounts for an average of 32% and 11% when the heap size is 48MB and 128MB respectively. For most cases, increasing the heap size has considerable energy benefits since the garbage collector is invoked less often. This reduction of GC overhead improves the overall energy-delay by effectively reducing both the execution time and the energy consumption of the system.

For Jikes RVM, the base compiler and the optimizing compiler consume very little energy in comparison with the rest of the measured components. For the SemiSpace collector, the base compiler has an average energy consumption of less than 1% across all tested applications. The optimizing compiler has an average energy consumption of 3% with a maximum of 7% for `_222_mpegaudio`. For the class loader the average energy consumption is 3% with a maximum of 24% corresponding to `fop`.

B. Energy-Efficiency

Figure 7 shows total benchmark energy-delay product as a function of heap size for all measured benchmark applications. The figure shows results for different garbage collectors and heap sizes under the Jikes RVM.

Jikes RVM configurations that use generational collectors perform better in terms of overall energy efficiency than configurations with non-generational collectors. For example, in the case of `_213_javac`, using a GenMS over a SemiSpace collector improves the EDP by as much as 70% when the heap size is fixed at 32MB. Note, however, that for some benchmarks, the efficiency of the non-generational collectors approach that of generational collector as the heap size is increased. An interesting exception is `_209_db`, where using a SemiSpace collector actually improves the energy-delay product by 5% over the best GenCopy collector when using a 128MB heap. This small decrease in the EDP comes in part from improved *mutator* performance. In other words, the improved data locality achieved by SemiSpace collector affects the application's performance in a positive way. GenCopy being a generational garbage collector also provides improved mutator locality. This improvement on performance, however, is undermined by a slight performance overhead of *write barriers* [16].

In many applications, the execution time of configurations that use non-generational collectors is greatly affected when varying the size of the heap: higher heap sizes produce shorter total garbage collector pause times. This drastic drop in execution time has a quadratic effect on EDP since a decrease in execution time is accompanied by a decrease in energy consumption as well. For example, when using a SemiSpace collector, `_213_javac`, `_227_mtrt`, and `euler` respectively see a 56%, 50% and 27% energy-delay product reduction when the heap size is increased from 32MB to 48MB. This in contrast to a 20%, 2% and 3% energy-delay product reduction on the same benchmarks when using a GenCopy collector. Consequently, we see that the garbage collector has significant more influence on the EDP metric than other JVM components.

The energy consumption of main memory follows a similar trend to the energy consumption of the processor: generational collectors, which tend to have less overall GC time, exhibit less memory energy consumption than their non-generational

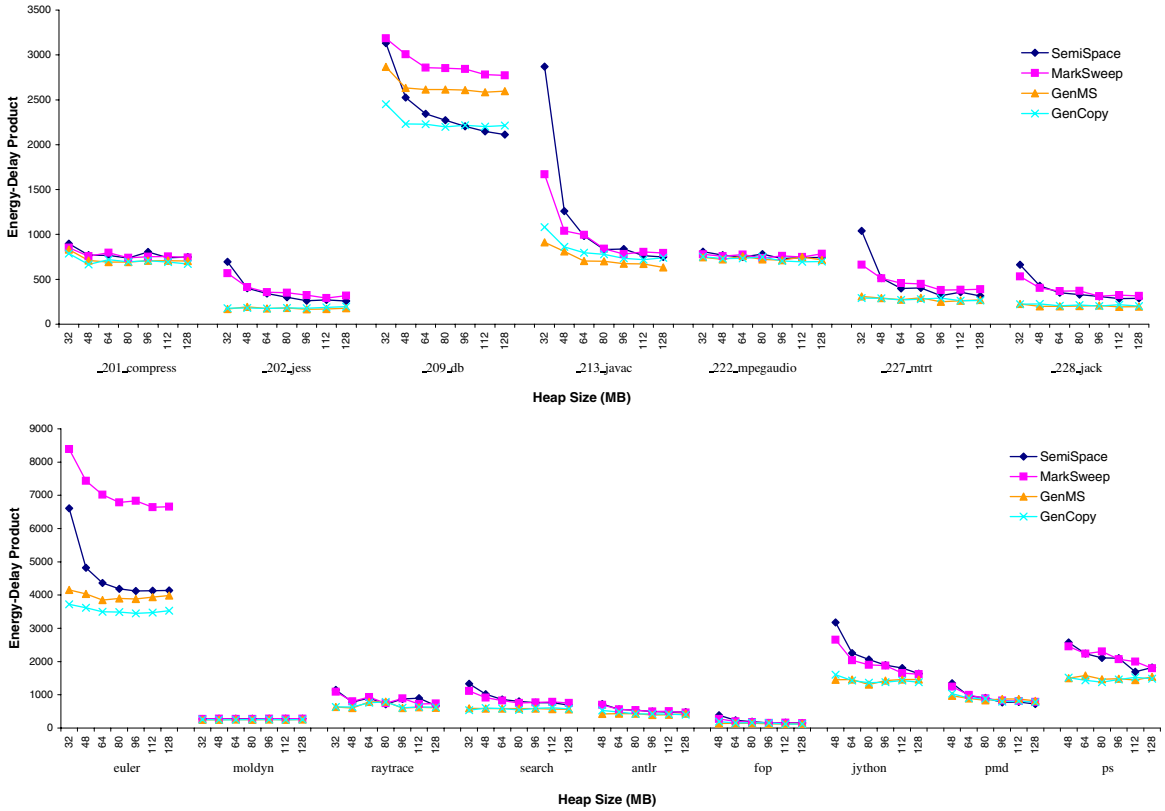

Fig. 7. Energy-Delay product for all benchmarks running on the Jikes RVM. A lower energy-delay product signifies a more energy-efficient virtual machine configuration for a particular benchmark. The reduction of the energy-delay product across garbage collectors is primarily attributed to a decrease in application execution time, t_0, rather than a reduction of overall power consumption, $P(t)$.

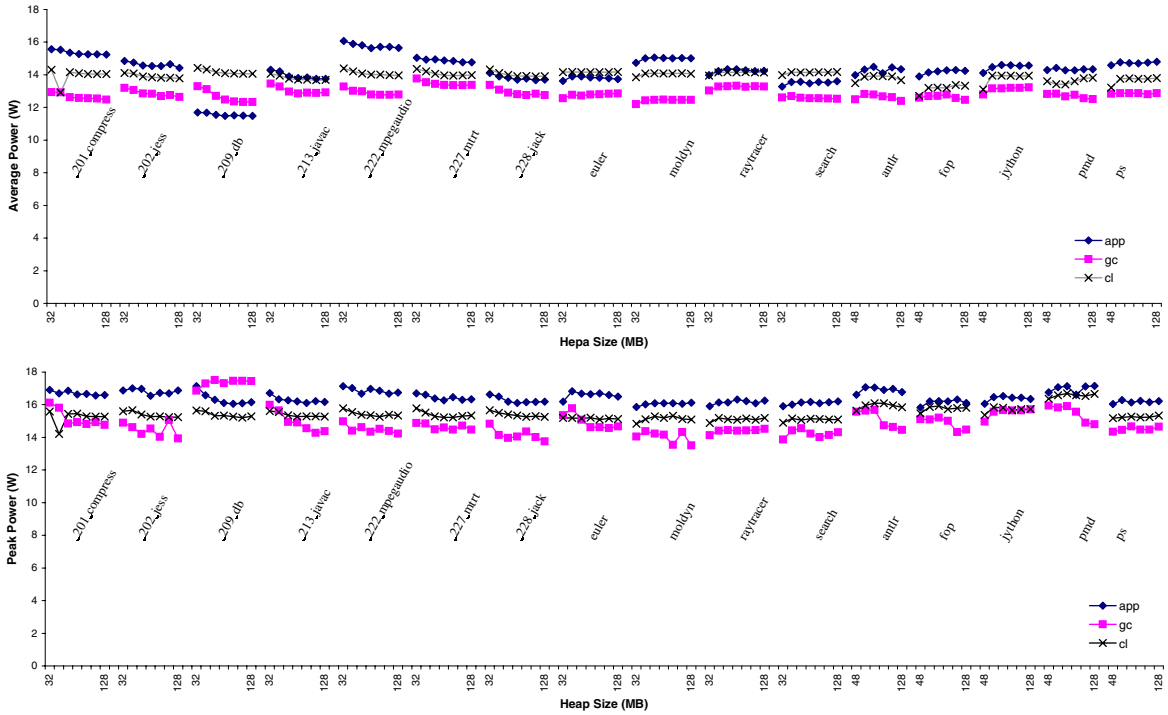

Fig. 8. Power consumption (top) and peak power consumption (bottom) for all measured benchmarks running on Jikes RVM with a generational copying garbage collector. JVM components exhibit little power variation from benchmark to benchmark. For most benchmarks, peak power consumption is dictated by the application and not by one of the studied JVM components.

counterparts. However, the energy consumed by main memory is small compared to that of the CPU; For SpecJVM98, the average memory energy is about 7%, 5% for DaCapo benchmarks, and 8% for Java Grande benchmarks.

C. Power Consumption

We now turn our attention to power, rather than energy. Figure 8 shows the average and peak power consumption of three components: The application, the garbage collector (GC), and the class loader (CL). These measurements correspond to the Jikes RVM using a generational copying collector (GenCopy) across multiple heap sizes. Our measurements indicate that the garbage collector is one of the least power-hungry components. While this is true for other garbage collectors measured in this study, different garbage collectors have different average power consumption. For GenCopy, the average power consumption is measured to be 12.8W; SemiSpace consumes an average of 12.3W. A GenMS collector under the Jikes RVM consumes an average of 12.7W while its non-generational counterpart consumes 11.7W. The common trend is that non-generational collectors consume less power on average, but have higher execution times (hence, in many cases, higher energy consumption). Non-generational collectors do not exploit object locality as efficiently as generational collectors, which increases processor stall time due to off-chip memory accesses. This power behavior can potentially have an important contribution in a thermal-aware Java virtual machine: by triggering garbage collection at points when the temperature of the processor has exceeded a safety threshold level, the processor executes a component with less power requirements, potentially giving it time to cool down to a safe level.

Power consumption is highly correlated with processor utilization [30][31][32]. This power-performance relationship can help us understand the power behavior of different JVM components. The generational garbage collector shown in Figure 8, for example, has an average L2 cache miss rate of 54% for SpecJVM and 56% for DaCapo. This in comparison to an average L2 miss rate of 12% for the class loader on SpecJVM98 and 21% for DaCapo, respectively. High L2 miss rates (along with a high number of L2 accesses) increase the amount of processor stall-time due to the high cycle cost of main memory access. This causes periods of low processor activity, which decreases power consumption at the expense of performance.

The average power consumption of other Jikes RVM components such as the class loader, the base compiler and the optimizing compiler, is generally higher than the average power consumed by the garbage collector. Moreover, these components have relative constant power consumption across heaps, and garbage collectors (performance counter values vary little across the various dimensions). Higher average power consumption of these components can be traced to higher processor resource utilization. For example, the component *application* has an average L2 miss rate of 11% and an IPC of about 0.8. The garbage collector shown in Figure 8 has an average IPC of 0.55.

The peak power consumed by our tested benchmarks are shown in Figure 8. An important observation here is that, for most benchmarks, the peak power comes within the application and not one of the studied JVM components. Consequently, it might be more cost-effective for future thermal-emergency avoidance efforts to focus on modulating application peak power, rather than optimizing JVM components. A visible exception is `_209_db`, where the garbage collector determines a peak power consumption of 17.5W.

D. The Kaffe Virtual Machine

The energy breakdown for the Kaffe virtual machine running on the P6 platform is shown in Figure 9. Kaffe's JVM component breakdown is very different from that of the Jikes RVM. The relative energy contribution of the JVM components (the garbage collector, the class loader and the JIT compiler) is much less visible than with the Jikes RVM. Kaffe's garbage collector has an average energy consumption of 7% across all measured applications. The class loader consumes 1% of the total energy while the JIT compiler is less than 1%. The low power consumption of JVM components in Kaffe is due to their short duration in comparison to the total running time of the benchmarks. It is also worth noting that these results are similar to the SpecJVM98 energy results shown in [9].

Kaffe's Mark-and-Sweep collector exhibits an average power consumption of 12.8W, very similar to the Jikes RVM Mark-and-Sweep collector. In the same way as the Jikes RVM, the power consumption of Kaffe's garbage collector is lower than the rest of the measured JVM components.

Kaffe JIT translates opcodes to native instructions without performing extensive code optimizations. This creates longer execution times for benchmarks causing it to consume larger amounts of energy. Figure 10 shows the EDP for Kaffe. Notably EDP changes little when increasing the heap size. This almost constant energy-delay product is a consequence of the very little performance improvements gained at higher heap sizes.

E. Kaffe on an embedded processor

So far we have been focusing on Java virtual machines executing on high-performance processors for consumer devices with large performance requirements. In addition to studying these platforms, we also want to consider devices targeted for hand-held platforms such as the Intel PXA255 processor. Since we expect different memory and workload requirements for a hand-held device than for a general-purpose processor, we reduce the test heap range to 12, 16, 20, 24, 28, and 32 MB. We also reduce the input set size of tested benchmarks from `s100` to `s10` for SpecJVM98 benchmarks. We perform our experiments on `_201_compress`, `_202_jess`, `_209_db`, `_213_javac`, and `_228_jack`. We use SpecJVM98 applications rather than more common embedded Java applications such as the EEMBC Java suite [33] for two reasons: (1) we want to close the application bridge between a high-performance JVM (Jikes) and an embedded JVM, and (2) SpecJVM98 applications stress the memory subsystem more effectively than smaller embedded applications (EEMBC applications are designed for very small heap size requirements, typically in the kilobyte range).

Figure 11 shows the energy distribution for Kaffe running on the PXA255 processor. The class loader is now the highest energy consumer with an average energy consumption of 18%. The garbage collector and the JIT compiler each have an average energy consumption of 5%. The high-energy consumption of the class loader is a consequence of its long execution time: Kaffe has a long initialization period characterized by a high number of calls to the class loader. Unlike Jikes, Kaffe does not merge system classes with the JVM binary, which reduces the size of the binary but generates more calls to the class

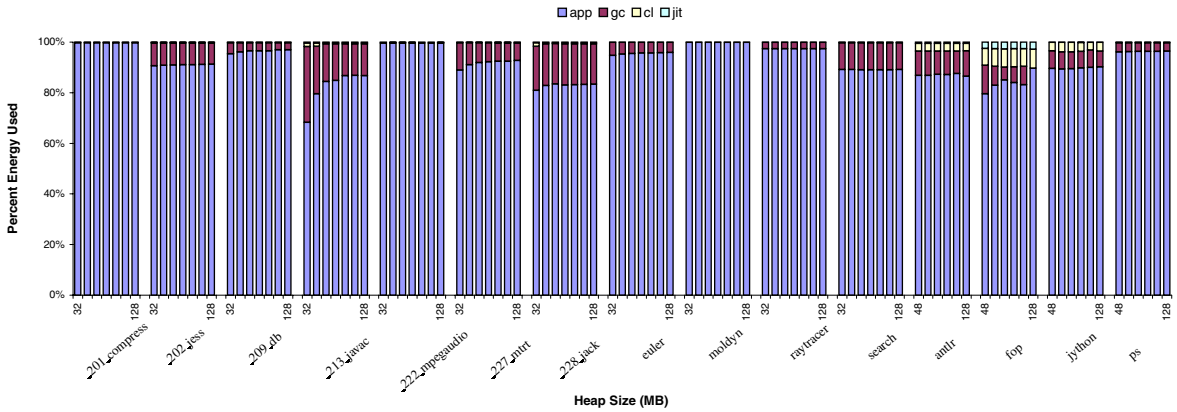

Fig. 9. Energy distribution for the Kaffe Java virtual machine running on a Pentium M 1.6Ghz processor.

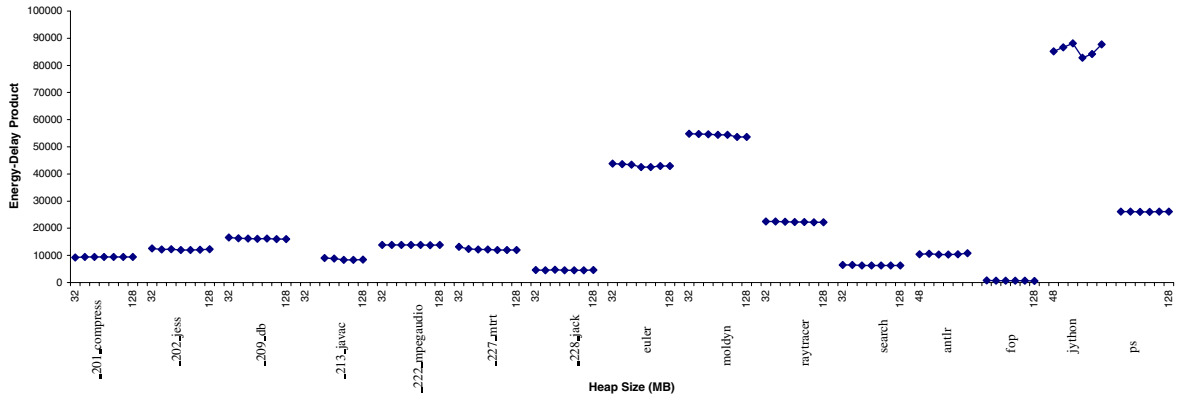

Fig. 10. Energy-delay product results for Kaffe running on a Pentium M 1.6Ghz processor.

loader. Consequently, for Java virtual machines running on embedded devices, improvements in class loading mechanism can lead to significant energy savings.

Another interesting difference is the fact that for Kaffe running on the Intel PXA255 processor, the garbage collector is the most power-hungry component with an average power consumption of 270mW, 7% higher average power consumption than the running application. This high average power consumption is caused by a high relative IPC when compared with other JVM components. In contrast, the class loader has the lower average power consumption, a consequence of lower CPU utilization. Closer examination of Kaffe's class loader performance reveals a large fraction of stall cycles, for which instruction fetch stalls and data dependencies represents a large fraction of the stall time.

VII. Future Work

Much future remains on these topics. Dynamic voltage and frequency scaling (DVFS) on real systems is a very effective tool in leveraging energy for performance [34][35][36]. In the future, we hope to investigate effective approaches that will reduce system energy consumption while minimally affecting performance. The present study has focused on client-based benchmarks; we hope to analyze server-type workloads in our effort to study thermal behavior of long-running applications. Power-aware scheduling algorithms through the use of dynamic processor and memory power estimation techniques using hardware performance counters [37] are also in future work.

VIII. Summary and Conclusions

We have presented a methodology for studying the energy consumption of the Java virtual machines and associated service components (garbage collection, class loading and dynamic compilation subsystem) *on real hardware*. Our study improves on previous art by using a live, running hardware platform rather than a simulation approach to study present and future factors that affect high-performance as well as embedded-class Java virtual machines. To our knowledge, we are the first to present such a methodology that covers both ends of the performance and energy consumption spectrum.

We have focused on two widely-used Java virtual machines: IBM Jikes Java Virtual Machine, a server-class virtual machine, and Kaffe, a clean-room, portable JVM suitable for high-end embedded devices. We perform a hardware-based power and performance analysis of the IBM Jikes RVM by studying different types of garbage collection algorithms with different heap sizes. Our results indicate that generational collectors offer the best energy-delay product. Under certain cases, a SemiSpace collector can improve this metric by as much as 12%. Generational collectors are also more power-hungry than their non-generational counterparts due to higher processor utilization. The class loading and dynamic compilation system tend to exhibit higher power consumption than the garbage collector, but lower than the Java application being run. When seeing the virtual machine as an extra layer between hardware and the running application, the fact that the application is responsible for peak power consumption and not some of the JVM services allows a more transparent power and thermal analysis of algorithms implemented in Java.

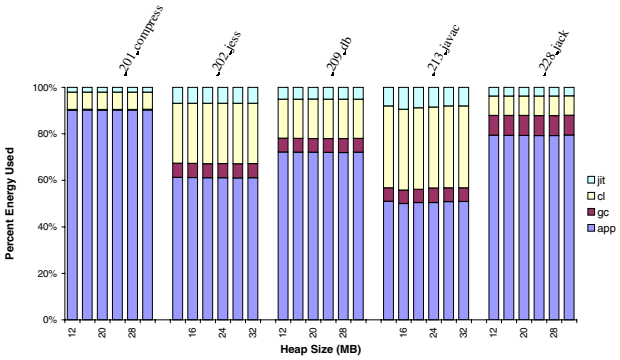

Fig. 11. Energy decomposition for Kaffe running on the Intel XScale PXA255 processor. The class loader accounts for an average energy consumption of 18% over five SpecJVM98 benchmarks. The garbage collector only consumes an average of 5% of the total energy.

REFERENCES

[1] T. Brecht, E. Arjomandi, C. Li, and H. Pham, "Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications," in *Technical Report CS-2000-04*, Department of Computer Science. York University, Ontario, Canada, 2004.

[2] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "The Garbage Collection Advantage: Improving Program Locality," in *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, October 2003.

[3] N. Sachindran, J. E. B. Moss, and E. D. Berger, "MC^2: High-Performance Garbage Collection for Memory-Constrained Environments," in *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, 2004.

[4] A. Heydari and V. Gektin, "Thermal and Electro-mechanical Challenges in Design and Operation of High Heat Flux Processors," in *The Ninth Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM '04).*, vol. 2, June 2004, pp. 694 – 696.

[5] Y. Hwang, R. Radermacher, S. Spinazzola, and Z. Menachery, "Performance Measurements of a Forced Convection Air-cooled Rack," in *The Ninth Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM '04).*, vol. 1, June 2004, pp. 194 – 198.

[6] Intel Corp., "Intel Pentium 4 Thermal Management," 2002, http://www.intel.com/support/processors/pentium4/thermal.htm.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Adaptive Garbage Collection for Battery-Operated Environments," in *USENIX JVM02 Symposium*, August 2002. [Online]. Available: http://www.gigascale.org/pubs/178.html

[8] G. Chen, R. Shetty, N. Vijaykrishnan, M. M. Irwin, , and M. Wolczko, "Tuning Garbage Collection in an Embedded Environment," in *Symposium on High Performance Computer Architecture (HPCA02)*, 2002.

[9] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin, "Energy Behavior of Java Applications from the Memory Perspective," in *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[10] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *Modeling and Computer Simulation*, vol. 7, no. 1, pp. 78–103, 1997.

[11] B. Alpern, C. R. Attanasio, et al., "The Jalapeno Virtual Machine," in *IBM Systems Journal*, vol. 39, no. 1, 2000.

[12] Kaffe Java Virtual Machine, http://www.kaffe.org.

[13] C. Hsieh, M. Conte, T. Johson, J. Gyllenhall, and W. Hwu, "A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms," in *CompCon '97*. IEEE, February 1997.

[14] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam, "Architectural Issues in Java Runtime Systems," in *Symposium on High Performance Computer Architecture (HPCA)*, 2000, pp. 387–398.

[15] A. S. Rajan, S. Hu, and J. Robio, "Cache Performance in Java Virtual Machines: A Study of Constituent Phases," in *The 5th Annual IEEE International Workshop on Workload Characterization*, 2002.

[16] S. Blackburn, P. Cheng, and K. McKinley, "Myths and Realities: The Performance Impact of Garbage Collection," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04)*, June 2004.

[17] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical Profiling: Understanding the Behavior of Object-Oriented Applications," in *19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, 2004.

[18] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind, "Using Hardware Performance Monitors to Understand the Behavior of Java Applications," in *USENIX 3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.

[19] J. Flinn and M. Satyanarayanan, "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications," in *Second IEEE Workshop on Mobile Computing Systems and Applications*, Feb. 1999, pp. 2–10.

[20] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J.-A. M. Anderson, "Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine," in *Measurement and Modeling of Computer Systems*, 2000, pp. 252–263.

[21] A. Diwan, H. Lee, and D. Grunwald, "Energy Consumption and Garbage Collection in Low-Powered Computing," in *Technical Report CU-CS-930-02*, University of Colorado, Boulder, 2002.

[22] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 9, pp. 1277–84, 1996.

[23] D. Brooks and M. Martonosi, "Dynamic Thermal Management for High-Performance Microprocessors," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA-7)*, January 2001.

[24] K. S. M. Stephen M. Blackburn, Perry Cheng, "A Garbage Collection Design and Bakeoff in JMTk: An Efficient Extensible Java Memory Management Toolkit," The Australian National University," Technical Report, 2003.

[25] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive Optimization In The Jalapeño JVM," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 47–65, 2000.

[26] *Intel DBPXA255 Development Platform for the Intel Personal Internet Client Architecture*, Intel Corporation, February 2003, order Number 278701-001.

[27] Standard Performance Evaluation Corporation, specJVM98 Documentation. release 1.03 edition, March 1999.

[28] J. E. B. Moss, K. S. McKinley, S. M. Blackburn, E. D. Berger, A. Diwan, A. Hosking, D. Stefanovic, and C. Weems., "The DaCapo Project Technical Report," 2004.

[29] The Java Grande Forum, http://www.epcc.ed.ac.uk/javagrande/.

[30] F. Bellosa, "The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems," in *Proceedings of 9th ACM SIGOPS European Workshop*, September 2000.

[31] C. Isci and M. Martonosi, "Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data," in *Proceedings of the 36th International Symp. on Microarchitecture*, Dec. 2003.

[32] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *International Symposium on Low Power Electronics and Design, pages 135–140*, 2001.

[33] Embedded Microprocessor Benchmark Consortium, "EEMBC GrinderBench for the Java 2 Micro Edition (J2ME) Platform," 2006, http://www.eembc.org.

[34] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004.

[35] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 38–48.

[36] A. Weissel and F. Bellosa, "Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France,*, Aug. 2002.

[37] G. Contreras and M. Martonosi, "Power Prediction for the Intel XScale Processor Using Hardware Performance Monitor Unit Events," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.