JUNE 2000

WRL Research Report 2000/5

Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine

Keith I. Farkas
Jason Flinn
Godmar Back
Dirk Grunwald
Jennifer M. Anderson

COMPAQ **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL), located in Palo Alto, California, is part of Compaq's Corporate Research group. Our focus is research on information technology that is relevant to the technical strategy of the Corporation and has the potential to open new business opportunities. Research at WRL ranges from Web search engines to tools to optimize binary codes, from hardware and software mechanisms to support scalable shared memory paradigms to graphics VLSI ICs. As part of WRL tradition, we test our ideas by extensive software or hardware prototyping.

We publish the results of our work in a variety of journals, conferences, research reports and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes, conference papers, or magazine articles. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

You can retrieve research reports and technical notes via the World Wide Web at:

http://www.research.compaq.com/wrl/

You can request research reports and technical notes from us by mailing your order to:

Technical Report Distribution
Compaq Western Research Laboratory
250 University Avenue
Palo Alto, CA 94301 U.S.A.

You can also request reports and notes via e-mail. For detailed instructions, put the word "Help" in the subject line of your message, and mail it to:

wrl-techreports@pa.dec.com

Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine

Keith I. Farkas Jason Flinn* Godmar Back* Dirk Grunwald* Jennifer M. Anderson*
Western Research Lab, Compaq Computer Corporation
250 University Ave., Palo Alto, CA. 94301 U.S.A.
keith.farkas@compaq.com

ABSTRACT

In this paper, we examine the energy consumption of a state-of-the-art pocket computer. Using a data acquisition system, we measure the energy consumption of the Itsy Pocket Computer, developed by Compaq Computer Corporation's Palo Alto Research Labs. We begin by showing that the energy usage characteristics of the Itsy differ markedly from that of a notebook computer. Then, since we expect that flexible software environments will become increasingly prevalent on pocket computers, we consider applications running in a Java environment. In particular, we explain some of the Java design tradeoffs applicable to pocket computers, and quantify their energy costs. For the design options we considered and the three workloads we studied, we find a maximum change in energy use of 25%.

1. INTRODUCTION

Advances in battery technology and low-power circuit design cannot, by themselves, meet the energy demands of mobile computers [1; 2]. Thus, it is critical that the software running on these devices be designed to minimize energy consumption. However, designing energy-efficient software requires that the software developer understand the energy-usage characteristics of the computer on which the software will be run and the energy impact of software design decisions. In this paper, we seek to further the software community's understanding of these two issues by characterizing the energy usage of a high-performance pocket computer and examining the energy impact of several options in the design of a Java run-time environment.

Developers must understand the energy-usage characteristics of the pocket computer for which they are developing software because significant energy costs may be incurred as a result of how the software makes use of system resources. For example, as the speed at which a system is run affects its energy consumption, developers may want to judiciously choose the system speed used to execute their applications. This decision must take into account the nature of the applications, the low power modes offered by the system, and the energy cost of the system when idle. In addition, the energy and power usage of pocket computers differ markedly from that of notebook computers. Thus, developers may need to re-evaluate decisions made for notebooks when porting applications to pocket computers.

Java, and similar programming environments, are essential enablers of mobile computing owing to their platform neutrality and flexibility. In the future, the personal (and mobile) computers that people will likely always carry with them will be used to run both a small set of core applications (e.g., web browser, e-mail client), and a potentially larger set of applications downloaded over a wireless connection for a specific purpose (e.g., a virtual tour guide). While the core applications could be stored in the native binary format of the computer, it is far simpler to distribute a single copy of a mobile application than to require that each distribution point maintain a copy of the application compiled for each platform in existence. Further, with platform neutrality, the task of migrating applications to new generations of platforms is much easier. For these reasons, we also discuss some of the tradeoffs in the design of a Java runtime environment, and quantify the energy costs of these tradeoffs. We also evaluate the design of the Kaffe [3] Java Virtual Machine (JVM) by quantitatively measuring the energy cost of several important design options.

The work discussed in this paper is based on energy measurements of the Itsy Pocket Computer [4], a state-of-the-art pocket computer developed by Compaq Computer Corporation's Palo Alto Research Labs. We begin the paper by reviewing important energy concepts, describing the Itsy Pocket Computer, and explaining our methodology. Then, we examine the energy-usage characteristics of the Itsy Pocket Computer in Section 3, and the energy impact of several Java design options in Section 4.

*Jason is a graduate student at Carnegie Mellon University (jflinn@cs.cmu.edu), Godmar is a graduate student at the University of Utah (gback@cs.utah.edu), Dirk was on sabbatical from the University of Colorado at Boulder (grunwald@cs.colorado.edu), Jennifer is now at VMware Inc. (jennifer@vmware.com).

This report was published in the proceedings of ACM SIGMETRICS 2000, the International Conference on Measurement and Modeling of Computer Systems.

Figure 1: The Itsy Pocket Computer Version 1.5

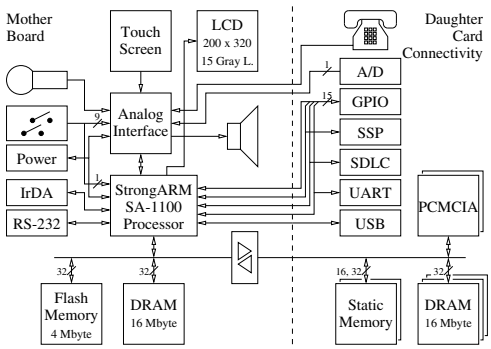

Figure 2: The Itsy system architecture (Version 1.5).

2. BACKGROUND

In this section we present background material and describe our methodology.

2.1 Energy Background

In normal usage, pocket computers run off batteries, which provide a finite amount of energy. The energy E, measured in Joules, consumed by a device over T seconds is equal to $\int^T p(t)$, where $p(t)$ is the instantaneous power measured in Watts. Given a sequence of n instantaneous power measurements, each taken δ seconds apart, the energy consumed may be estimated as $\sum_{i=1}^{n} p_i(t) \times \delta$. Finally, the average power P for such a sequence is approximately $\frac{1}{n}\sum_{i=1}^{n} p_i(t)$.

Considering again instantaneous power, we note that many of the components used in pocket computers today, such as memories and microprocessors, are implemented in CMOS. For such components, the instantaneous power they consume is proportional to $V^2 \times F$, where V is the voltage supplying the component, and F is the frequency of the clock driving the component.

2.2 The Itsy Pocket Computer

The Itsy Pocket Computer is a flexible research platform, developed to enable hardware and software research in pocket computing. It is a small, low-power, high-performance handheld device with a highly flexible interface, designed to encourage the development of innovative research projects, such as novel user interfaces, new applications, power management techniques, and hardware extensions.

Several versions of the Itsy Pocket Computer have been developed, with the differences between versions being small changes in the hardware. These changes have little impact on the results we report here. All versions are based on the low-power StrongARM SA-1100 microprocessor [5]. Further, all versions have a small, high-resolution display, which offers 320×200 pixels on a 0.18mm pixel pitch, and 15 levels of gray. All versions also include a touchscreen, a microphone, a speaker, and serial and IrDA communication ports, while the newest version (which was not used in this study) also has an integrated USB port. The Itsy architecture can support up to 128 Mbytes of DRAM and 128 Mbytes of flash memory. The flash memory provides persistent storage for the operating system, the root file system, and other file systems and data. Finally, the Itsy also provides a "daughter card" interface, which allows the base hardware to be easily extended.

The version 1.5 unit used as the basis for this work has 64 Mbytes of DRAM and 32 Mbytes of flash memory. It is powered by two size AAA batteries, which can supply the $\approx$2 Watts maximum demanded by the Itsy. Figure 1 shows a picture of a version 1.5 unit, while Figure 2 shows a block diagram of the Itsy architecture.

The system software includes a monitor and a port of Linux operating system (version 2.0.30). The monitor allows a user to adjust certain system parameters, to run applications directly on top of the hardware, and to download and boot the operating system. Linux provides support for networking, file systems and multi-user management. Applications can be developed using a number of programming environments, including C, X-Windows, SmallTalk and Java. Applications can also take advantage of available speech synthesis and speech recognition libraries.

Using a conventional pair of alkaline AAA batteries, the Itsy computer can run for $\frac{1}{2}$ hour in "high power" mode, such as when playing an MPEG video or the popular action game Doom. If the system is idle while the processor clock is set to 206 MHz (the fastest speed), the integrated power management modes stall the processor such that the same batteries last for 2 hours. This time increases to 18 hours if the processor is idle at 59 MHz (the slowest speed), and to one week if the system is in sleep mode.

The capabilities of the Itsy are in stark contrast to existing pocket computers such as the Palm Pilot. The processing capability of the Itsy supports speech synthesis, speech recognition, high quality MPEG video playback, audio codecs and other advanced audio processing. The large dynamic memory allows applications to be developed using rapid prototyping environments that also enable mobile applications and application reuse. The flash file systems allow complex systems to be developed and configured.

2.3 Measuring Power and Total Energy

To measure the instantaneous power consumed by the Itsy, we measure the voltage drop across a 0.02 Ω precision resistor that is located in the main power circuit. As power is drawn, current flows out of the battery or external voltage supply, through this sense resistor, and into the computer. This current induces a differential voltage V_{sense} across the resistor, from which the current I_{sense} may be calculated; $I_{sense} = \frac{V}{R} = \frac{1}{0.02} \times V_{sense}$. The power being consumed may then be calculated by multiplying this current by the voltage, V_{supply}, that is being supplied to the computer.

We measure these two voltages using two differential amplifiers and a data acquisition (DAQ) system while the Itsy is powered by an external voltage supply. A block diagram

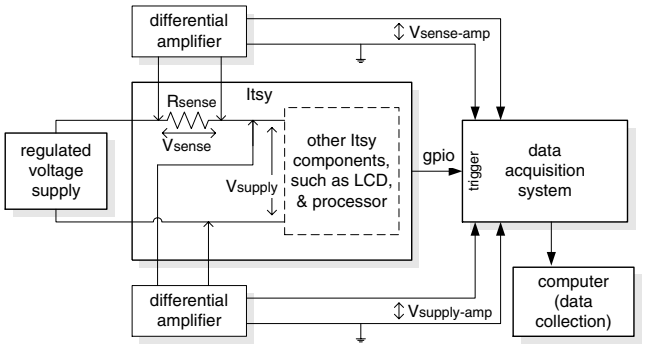

Figure 3: Block diagram showing the infrastructure used to measure the power consumed by the Itsy.

showing these components and the Itsy is given in Figure 3. The amplifiers minimize the error introduced into our measurements of V_{sense} and V_{supply} by electro-magnetic noise and the limited precision of the DAQ system; we call the output of these amplifiers $V_{sense-amp}$ and $V_{supply-amp}$, respectively. The output of each amplifier is connected to one of the analog inputs of the DAQ system.

When instructed to do so, the DAQ system reads the voltage that is present on the specified analog input, converts it to a 16-bit binary value, and forwards it to the host computer. The host computer stores these readings for subsequent analysis. By instructing the DAQ system to read the voltage on an input channel at regular time intervals, we determine how the voltage varies over time. For our experiments, we used a sampling rate of 5000 times per second. From these time profiles of the two voltages, we can compute a time profile of the power used by an application as it runs on the Itsy, and thus, the total energy consumed during this time.

In this paper, we examine the power use of a number of micro-benchmarks and several applications written in Java; these workloads are described in Sections 3 and 4 respectively. To determine the power-use profile of a workload, we measure the time required to execute the workload and then select the relevant set of voltage measurements from the data collected by the DAQ system. For the micro-benchmarks, we use the SA-1100's cycle counter to count the number of clock cycles required to execute the benchmark, while for the Java applications, we use the time function built in to the `csh` command shell. To synchronize the collection of the voltages with the start of execution of a workload, as the workload begins executing, it toggles one of the SA1100's general-purpose input-output pins. This pin is connected to the external trigger of the DAQ system, and when it is toggled, causes the DAQ system to begin recording measurements. This connection is labeled *gpio* in Figure 3.

The DAQ system we use can make only one voltage measurement at a time and measurement accuracy is decreased by quickly switching between inputs, so we used the following strategy. Before running a workload on the Itsy, we measure the supply voltage for several seconds and record its average value. Then, we configure the DAQ system to measure only the voltage drop across the sense resistor, $V_{sense-amp}$, and run the workload. Once the application completes, we use the predetermined average supply voltage to compute the instantaneous power at each time step during the execution of the workload.

This strategy increased the error in our measurements compared to what would have existed had we been able to measure both voltages simultaneously. To estimate the error in our supply voltage measurement, we measured the variation in the supply voltage during the replay of several of the workloads we considered. The resulting range of values varied less than $\pm$ 2 millivolts (mV) from the mean value. A second source of error corresponds to error induced by the measurement equipment and noise. To estimate the error in V_{sense}, we measured its variation for several seconds while the Itsy was executing a benchmark with a constant load. This measurement gave a $\pm$ 0.075 mV variation about the mean. The net effect of these errors is an error of $\pm$ 0.005 Watts, which in our experiments, yields an error of at most $\pm$ 1.1 Joules. These values represent the maximum error; we see much smaller variation in our measurements.

3. COMPONENT CHARACTERIZATION

In this section, we examine the energy-usage characteristics of the Itsy Pocket Computer, beginning with the system as a whole, and then for three important subsystems.

3.1 System Characterization

Using the methodology described in the previous section, we obtained power-use profiles such as the one shown in Figure 4. This profile shows the power consumption of the Itsy for a workload comprised of booting the operating system and a Java virtual machine several times and then running several Java applications within a single virtual machine. The first application lowered and then raised the clock speed of the processor and of the system. Note that although adjusting the clock speed produces a clear power reduction, running the calculator application or the drawing package produces no clear power usage patterns. Also, when the system is left idle for a period of time, a short power spike occurs, probably due to garbage collection. Finally, the profile for this workload shows considerable variability in the power demands of the Itsy, a characteristic shared by the large number of other profiles we have gathered. The Itsy's dynamic range of power demand is much larger than that reported in previous studies of notebook computers [6; 7]; we discuss the reasons for this difference in the next section.

3.2 Subsystem Characterization

To understand the power consumption of the processor, display, and memory subsystems of the Itsy Pocket Computer, we measured the power and energy consumption of a number of micro-benchmarks. Each micro-benchmark disabled (i.e., powered down) unused hardware components whenever possible. Further, we ran these directly on top of the hardware to eliminate any operating system effects. Due to space constraints, we discuss only a subset of the results; a more complete discussion can be found in [8].

The power and energy consumed by the benchmarks is given in Table 1. Column 2 lists the benchmarks, while columns 4 to 9 present the power and energy consumed by the benchmarks when run on the Itsy. For each benchmark, we report power and energy consumption for three clock speeds: the slowest possible speed, 59 MHz (columns 4 and 7), the mid-point speed, 133 MHz (columns 5 and 8), and the fastest possible speed, 206 MHz (columns 6 and 9). We report energy for only those benchmarks that perform a task

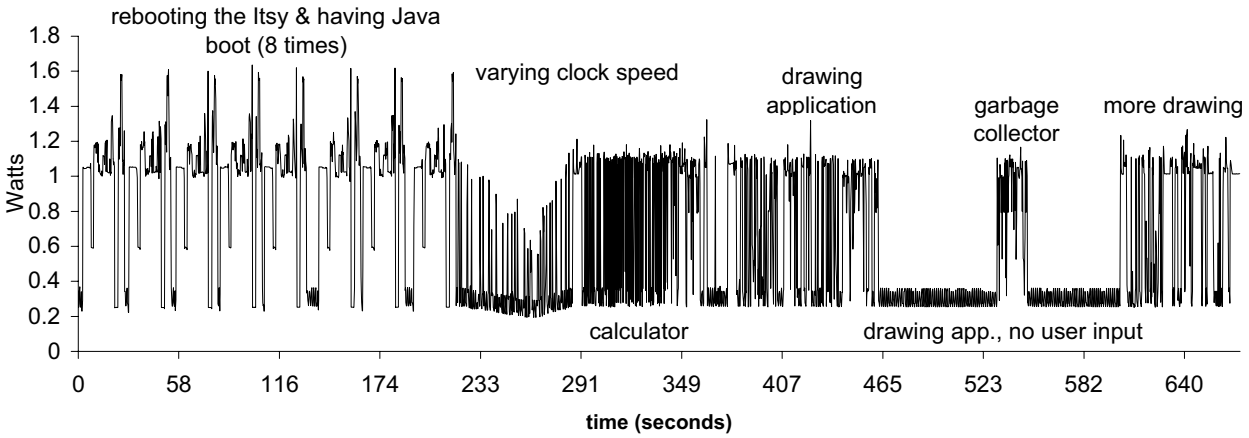

Figure 4: Power-use profile of an Itsy pocket computer as it boots a JVM and runs several applications.

with a specific completion condition, say executing a fixed number of additions. For applications with no specific completion condition, only power is meaningful.

The table also provides data corresponding to powering the processor from 1.5 Volts, the normal voltage, and 1.23 Volts, a reduced voltage; the voltage level is indicated in column 3. Note that we present the data for operation at the lower supply voltage only to illustrate the benefits of voltage scaling; the StrongARM processor is not rated for operation at the lower voltage level.

Finally, as a point of comparison, column 10 lists the power consumed a subset of the benchmarks when they were run on an IBM ThinkPad 560X notebook computer. This notebook had a 233 MHz Pentium Processor and 64 Mbytes of memory. We measured its power consumption using a technique similar to the one used for the Itsy.

3.2.1 Processor and Display Subsystems

Consider first the `sleep` micro-benchmark (row 1). This benchmark puts the system into a mode in which only the DRAM is refreshed at a constant rate; thus, the clock speed of the Itsy has no impact on the power used. In comparison, with the `idle mode` benchmark (row 2), less power is consumed by the Itsy at lower clock frequencies than at the higher ones. This relationship exists because, in idle mode the processor pipeline is stopped, but on-chip auxiliary components are still clocked. These components, in turn, consume power proportional to the clock frequency.

Next, consider the `busy wait` micro-benchmark, in which the processor executes a busy-wait loop for five seconds, and the `addition` benchmark, in which the processor executes 300 million iterations of a compute-intensive loop. Observe that there is a difference in the power consumption of each of these benchmarks (compare rows 4 and 8, 5 and 9). In both cases, this difference is due to differences in the instruction types executed by the benchmarks.

We expect the `busy wait` and `addition` benchmarks to be a "high power" test for the processor since during their execution, the Itsy and the ThinkPad processors never stall for memory. From the table, we note that the maximum power consumption of the ThinkPad is 8.3W (row 4), and between 314mW and 899mW for the Itsy (row 8) for these two benchmarks. Comparing these values to the idle power consumption (row 2), we see that the ThinkPad consumes 260% more power in the "high power" test, while the Itsy consumes between ≈340% to ≈550% more power in this test. The large percentages for the Itsy suggest that its processor exhibits a wider dynamic range of power consumption. This fact helps account for the Itsy's overall dynamic power demand being greater than that of a notebook.

Consider again the execution of the `busy wait` benchmark, but this time with the LCD of each computer turned on (row 6). For the ThinkPad computer, observe that turning on the LCD consumes between 2.1W and 4.8W, depending on the brightness of the backlight, while for the Itsy, the LCD consumes an average of 38mW. Thus, relative to the "high power" tests, turning on the LCD increases the power consumption of the ThinkPad by 25% to 57%, while doing so increases the Itsy's power consumption by 5% to 17%. However, if this version of the Itsy used a backlight similar to the one used in Version 2 of the Itsy Pocket Computer, we estimate the maximum power consumption of the LCD to be between 300mW and 600mW. Thus, the backlight would increase the "high power" test power by up to ≈380%. Nonetheless, we expect the backlight of a pocket computer to be used infrequently, while the backlight of a notebook is used almost continuously. Hence, the power consumption of the backlight is less of an issue for a pocket computer.

The `addition` micro-benchmark is the first for which we show both power and energy, because it performs a task with a specific completion condition. Note that at the higher clock rates, the Itsy consumes more power, but consumes less energy. This trend is also suggested by the other benchmarks for which energy is reported. Although lowering the clock frequency reduces power usage, the task takes longer to complete. Hence, the overall energy remains about the same or increases slightly at lower clock frequencies.

If the voltage is reduced along with the clock frequency, then the drop in power consumption dominates the corresponding increase in execution time. To illustrate this effect, consider again the `addition` benchmark. Note that when the processor's voltage is reduced from 1.5 V (row 8) to 1.23 V (row 9) and the frequency is kept constant, the energy consumption drops by ≈20% ($= 1 - \frac{4.181}{5.226}$). However, while this energy savings was achieved for this benchmark, in practice, to achieve reliable operation, the processor's clock speed must usually be reduced when its supply voltage is reduced. Thus, a more practical illustration of the energy savings brought about by voltage scaling is the energy sav-

		Itsy							ThinkPad
#	Micro-Benchmark	Processor Voltage	Power (Watts) at Specified MHz			Energy (Joules) at Specified MHz			Power (Watts)
			59.0	132.7	206.4	59.0	132.7	206.4	
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
1	Sleep mode	normal	0.010	0.010	0.011	––	––	––	0.312
2	Idle mode	normal	0.094	0.128	0.162	––	––	––	3.20
3	Idle mode, LCD enabled	normal	0.134	0.164	0.198	––	––	––	5.15-7.74
4	Busy wait	normal	0.225	0.412	0.596	––	––	––	8.30
5		reduced	0.177	0.324	0.469	––	––	––	
6	Busy wait, LCD enabled	normal	0.263	0.447	0.632	––	––	––	10.4-13.1
7		reduced	0.217	0.363	––	––	––	––	
8	Addition loop	normal	0.314	0.612	0.899	6.388	5.534	5.226	7.43
9		reduced	0.247	0.490	0.719	5.030	4.427	4.181	
Memory Test with instruction cache, MMU, write buffer and data cache enabled									
10	In-cache read test	normal	0.385	0.763	1.133	0.228	0.201	0.191	7.51
11	Out-of-cache read test	normal	0.458	0.719	0.777	8.439	6.419	6.880	6.74
12	In-cache write test	normal	0.383	0.762	1.120	0.227	0.200	0.189	7.43
13	Out-of-cache write test	normal	0.731	1.290	1.572	3.894	3.555	3.977	7.30
Memory Test with only instruction cache enabled									
14	In-cache read-test	normal	0.504	0.809	1.023	3.717	3.291	3.197	
15	Out-of-cache read-test	normal	0.523	0.840	1.063	3.853	3.415	3.320	
16	In-cache write test	normal	0.566	1.075	1.183	3.535	3.018	3.017	
17	Out-of-cache write test	normal	0.566	1.075	1.183	3.574	3.013	3.017	

Table 1: Average power and energy consumption of select micro-benchmarks running on an IBM ThinkPad 560X notebook computer and the Itsy Pocket Computer. The voltage supply of the Itsy's processor was either 1.5 V (normal) or 1.23 V (reduced).

ings obtained by operating at 1.23 Volts and 133 MHz rather than 1.5 Volts and 206 MHz. In this case, 16% ($= 1 - \frac{4.427}{5.226}$) less energy is consumed. Comparable energy savings occur with the `busy wait` benchmark (compare row 4 to 5, row 6 to 7), but in this case, the StrongARM failed to operate at the low voltage level and highest clock speed when the LCD was turned on.

3.2.2 Memory Subsystem

The next set of benchmarks measure memory system demands. They are divided into two subsets. The first subset (rows 10-13) enables all aspects of the memory system, and the second subset (rows 14-17) only enables the instruction cache. Other measurements we did on the Itsy show that disabling other components of the memory system (*e.g.* disabling the MMU) had little impact on the power demands. Each memory benchmark reads or writes 100 Mbytes of data. The tests were designed to use a working set that could either fit in the on-chip cache or explicitly *not* fit in that cache. For the Itsy, the 16-fold difference in energy use for the read test when data caching is enabled (row 10) and disabled (row 14) demonstrates the benefits of data caches – the additional energy is proportional to the additional time needed to run the benchmark. What is more interesting to note is that it takes less energy to conduct the out-of-cache read tests when the cache is disabled (compare row 15 to row 11). In these benchmarks, the cache provides no benefit, and by disabling it, the execution time is reduced significantly while the power consumption is increased by a small amount.

Finally, comparing the power consumption of the Itsy to that of the ThinkPad when executing the memory benchmarks, we note that unlike with the Itsy, the power consumed by the ThinkPad's memory system had little impact on its overall power consumption. Expressed as a fraction of the power consumed by the "high power" tests, the power variance across the memory benchmarks for the ThinkPad is 9% ($= \frac{7.51-6.74}{8.30}$), while for the Itsy, the variance is 110% ($= \frac{0.731-0.383}{0.314}$) to 88% ($= \frac{1.572-0.777}{0.899}$), depending on the clock frequency. The differences in the fractions for the computers is due to the ratio of the power consumed by the processor to that consumed by the memory system being much larger for the ThinkPad than for the Itsy. Thus, variations in the power consumed by the memory system have a greater impact on the overall power consumption of the Itsy. Consequently, because the memory behavior of applications varies during their execution, applications running on pocket computers, such as the Itsy, will exhibit a wider range of power demands than if they were run on a notebook computer, such as the ThinkPad.

3.3 Discussion

In summary, the Itsy Pocket Computer exhibits a much wider range of dynamic power demand than does the ThinkPad computer. This wider range is due to two effects: the processor used in the Itsy exhibits a wider range of power demand; and the power consumption of the memory system accounts for a greater fraction of the overall power consumption of the Itsy.

The best clock frequency at which to run an application depends on two main factors: the number of idle periods, and whether the system has the ability to scale down the voltage along with the clock frequency. For compute-intensive applications with little idle time running on systems with no voltage switching, applications should be run at as fast a clock frequency as possible until completed, and

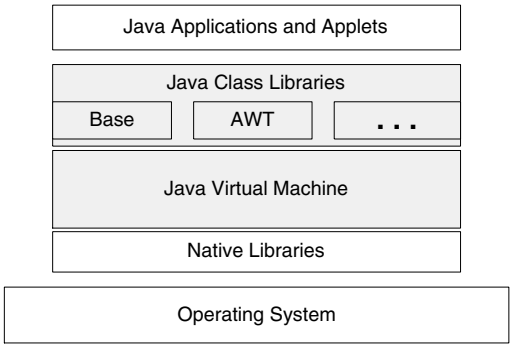

Figure 5: The Java Platform

then the system should be placed into a low-powered idle mode. However, if voltage switching is available, then the best clock frequency is the slowest speed at which the application can still meet its deadline. For applications with inherent idle time, the best clock frequency will depend on the power cost of idling, and the power saved when the clock frequency is reduced. Finally, we note that reducing the clock speed also extends the lifetime of the battery, as the capacity of batteries is reduced with increased power consumption. The importance of the battery capacity and clock speed tradeoff is a function of the type of battery used, and the peak and average power consumption [9].

4. JAVA AND ENERGY CONSUMPTION

In this section, we discuss the energy impact of decisions made in designing a Java virtual machine. We begin by presenting an overview of the Java runtime environment. Then, we enumerate the design tradeoffs we consider in Section 4.2, and evaluate their energy impact in Section 4.4 using the workloads described in Section 4.3.

4.1 Java Background

The Java [10] language provides many features of modern programming languages, including object orientation, strong typing, automatic memory management and multithreading. More importantly, it offers features that are highly desirable for mobile applications. In particular, Java source code is not directly compiled into native code, but into an architectural-neutral intermediate form called Java bytecode. This intermediate representation facilitates cross-platform development and deployment, and allows for easy migration of applications as system generations evolve. The Java language also offers a security model that helps guard against malicious applications.

Two key components are required to run a Java application on a system: a set of class libraries, and a Java Virtual Machine (JVM). These components are represented by the shaded boxes in Figure 5.

A class library comprises a number of classes, each of which is a collection of data and methods that operate on that data. Taken together, the contents of the class libraries offer software components that provide the standard functionality required by applications. One important set of class files is that provided by the Java Abstract Window Toolkit (AWT). The class files provided by this toolkit implement common graphical user interface components, such as those for displaying windows and receiving input from a user.

The JVM is a run-time system that executes Java bytecode instructions. Execution is done by either interpreting each instruction, or by compiling the bytecode "just in time" into native instructions, and executing these instructions directly. If during the execution of an application, a reference is made to a class that has not been previously accessed, the JVM suspends the application, and loads the needed or referenced class. This task involves several steps:

First, the JVM reads and parses the file containing the missing class. This file contains both symbolic type information and the actual bytecode instructions of the class. Because this class file may be stored in compressed format, the JVM may have to first decompress the file. Second, the JVM verifies that the bytecode does not violate the safety guarantees demanded by the Java language. Third, the JVM performs the necessary linking steps to resolve symbolic references in the class. This task may involve loading and linking referenced classes. Finally, the JVM resumes the execution of the application.

Interpreters usually execute bytecode one or two orders of magnitude slower than translated or compiled code [11]. However, just-in-time compilation takes time, and as such it may not be beneficial for infrequently invoked code. Ordinary ahead of time compilation, which is used for languages such as C or C++, is also available. However, ahead-of-time compilation not only sacrifices portability, but also the ability to verify the safety of untrusted code, such as mobile code contained in applets loaded over the network.

Since Java is not yet in wide-spread use on pocket computers, it is difficult to extrapolate how Java virtual machines for such systems will be used. In conventional systems, a virtual machine is used in one of two modes. In the "on line" mode, applications are downloaded and run for short periods of time; thus the initialization and translation time may be a significant fraction of the execution time. Similar behavior is exhibited by users of devices such as the Palm Pilot, where information is stored in and loaded from persistent memory whenever the user switches to another application. In contrast, in the "server" mode, applications such as Java Server Beans execute for a considerable period of time. Applications typically run in individual virtual machines, although they may share a virtual machine. At this time, it is unclear if Java application programmers would make the same design decisions if they wrote their applications for one mode or the other; thus, we consider both modes.

4.2 The Design Options

We consider two sets of JVM design options: startup options and runtime options. The startup options affect the costs of actually beginning the execution of an application or required method. As the startup time is typically much less than the execution time of an application, the energy cost of the startup will be much smaller than that of executing the application. However, depending on the usage pattern, startup costs may be incurred frequently. The runtime options affect the cost of running an application, once the startup process has completed. For this second set, the overall energy consumed is of interest. We consider four startup options, and two run-time options.

Startup:

- **Use of a single JVM versus multiple JVMs:** Tra-

ditionally, each Java application runs in its own virtual machine. However, more recently, solutions have emerged that allow multiple applications to be run in the same virtual machine.

- **Use of compressed versus uncompressed class files:** Persistent storage space on portable devices is at a premium. For this reason, the class files containing the bytecode for the supporting Java run-time libraries are stored in compressed form. We consider either uncompressing the class files when the JVM loads them, or when the Itsy Pocket Computer boots Linux. In the former case, we say that *compressed* class files are used, while in the latter, we say that *uncompressed* class files are used.

- **Class loading and just-in-time compilation:** Typically, classes are loaded on demand as they are needed, and methods are just-in-time compiled when they are first invoked. The startup delay is reduced by prefetching classes that are likely to be loaded later.

- **Cache flushing after code generation:** When code is just-in-time compiled, the data cache, and in certain cases the instruction cache, must be invalidated. We consider various options on how and when to flush or invalidate the caches.

Run-time:

- **Interpreting the byte code versus just-in-time compilation:** Our JVM can be configured to either interpret or just-in-time compile bytecode.

- **Polling frequency:** User input devices, such as the buttons or the touchscreen, are polled at regular intervals. Higher polling frequencies improve responsiveness, but also use more energy.

4.3 The Workload

To evaluate the impact on energy consumption of the design options mentioned in the previous section, we implemented each of them using the Java run-time environment that runs on the Itsy Pocket Computer. This environment is a port of the Kaffe JVM [3]. Then, we measured the power consumed by several benchmarks as they were executed by the Itsy Pocket Computer.

Our workload is designed to capture the two modes in which a pocket computer will likely be used. The first mode corresponds to using a single application for extended periods of time, such as a game or web browser. The second mode corresponds to using one or more applications for short periods of time, and either re-invoking the same application repeatedly, or switching between applications. An example of this mode is the use of an e-mail application to read e-mail messages with embedded attachments. To view each attachment, it may be necessary to start up helper applications, such as an MPEG player, and to switch repeatedly between such applications.

Our workload is comprised of the following activities: a human browsing web pages stored on the Itsy, (the **web** activity); a human playing a chess game against the Itsy Pocket Computer (the **chess** activity); and a human playing a chess game, then browsing a web page, then using a calculator, (the **composite** activity).

We modified the Java platform to allow user-generated events of button presses and touchscreen activations to be recorded into a file and then replayed at a later time. To ensure these subsequent replays mimicked as much as possible the original human's interaction with the Itsy, we also time stamped the events. We recorded three such trace files, one for each of the above activities in our workload. We then took power measurements while replaying the traces with each of the different JVM design options, with as close to the original inter-event timing as possible.

4.4 Discussion and Quantification

In this section, we describe the tradeoffs involved in each of the design decisions, the issues arising from their implementation, and their impact on energy consumption. We analyze each design option in relation to a baseline system that employs multiple JVMs, uncompressed class files loaded on demand, just-in-time compilation of class methods, and an AWT polling frequency of 0.03 seconds, which balances the energy cost of polling against response time.

We collected power profiles of the traces described in Section 4.3 for the baseline system and each variation. We repeated each experiment five times, and for each trial, computed the time required to execute the trace, the average power consumed, and the total energy consumed. In the following, when we report time, average power, and total energy, we are reporting the average of the five trials. Note that across all trials for all experiments, the standard deviation in the time, average power and total energy was less than 0.5 seconds, 0.002 Watts, and 0.6 Joules respectively.

We begin by discussing design decisions that impact the startup costs, and then follow in Sections 4.4.5 and 4.4.6 with those that impact the execution costs.

4.4.1 Single JVM versus Multiple JVMs

There are a number of tradeoffs regarding the use of a single JVM or multiple JVMs to run multiple applications. These tradeoffs affect both the (time and energy) startup costs as well as the overall consumption of resources, such as memory.

Running multiple applications in a single JVM reduces startup costs because these applications can share the Java classes contained in the Java run-time libraries. Therefore, these classes have to be processed only once for all applications. Other data structures are shared as well, such as the internal table of loaded classes and the symbol table. Due to Java's late binding, the JVM's run-time linker needs to maintain a large amount of symbolic information, which can be shared to avoid having to store multiple copies of a given symbol string. If applications need to communicate with each other, they can often do so by direct method invocation.

To evaluate the energy costs of using a single JVM versus multiple JVMs, we gathered power-use profiles of the execution of the composite trace. Average power and the total energy computed from the profiles are shown graphically in Figure 6. In this figure, there is a set of two bars for each Java design option, with one in each set giving the average power, and the second giving total energy. The x-axis indicates the design options, while the left-hand y-axis plots the average power in Watts, and the right-hand y-axis plots the total energy in Joules. In this graph and all that follow, the baseline configuration is noted by appending an asterisk

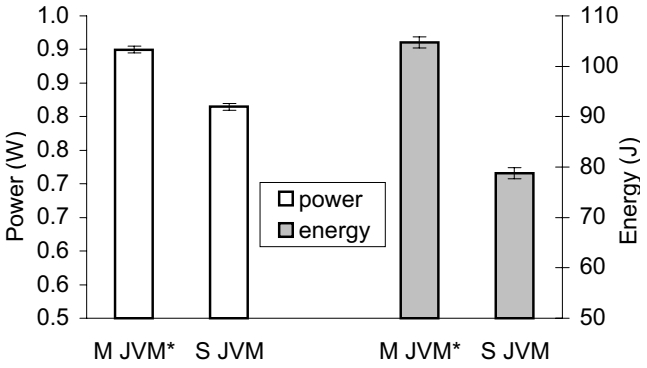

Figure 6: The energy consumption of the composite trace when a single JVM (SJVM) is used instead of multiple JVMs (MJVM).

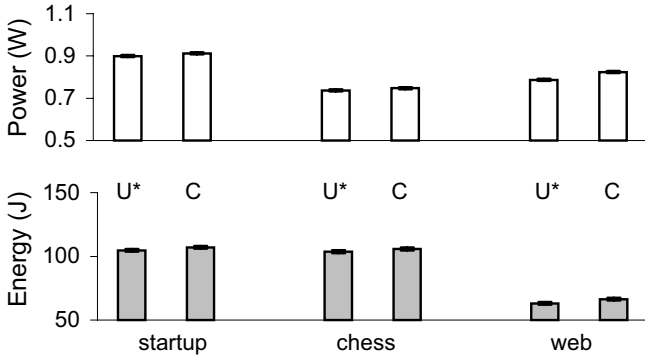

Figure 7: The energy consumption of all three traces when compressed JAR files (C) are used instead of uncompressed JAR files (U).

(*) to the x-axis label. The error bars represent an error of $\pm$ 0.005 Watts, and $\pm$ 1.1 Joules (see Section 2.3). Finally, the execution time, average power, and total energy data for this experiment and all others are given in Table 2 of the Appendix.

For the composite trace, the use of a single JVM resulted in a 10% reduction in the average power, and a 25% reduction in the total energy. At the same time, the execution time decreased by 17%, due to the time not spent repeatedly loading virtual machines. Clearly, the single JVM is more energy efficient for this trace. However, the sharing of resources required by a single JVM has drawbacks: because resources are shared, one application's use or misuse of a resource may affect other applications. For instance, since the memory allocator and garbage collector are shared, excessive allocations by one application will reduce the amount of memory available to all other applications. Also, applications may be delayed during garbage collection for other applications. If one application fails because of a bug in the run-time libraries or in the JVM, sometimes the whole JVM cannot continue to execute, resulting in all applications being aborted. Because we assume co-operating applications, our implementation did not need to address these issues.

Current research [12] tries to address the challenges of enabling a JVM to run multiple applications safely and without allowing one application to impact another—we believe that while such a JVM is more complex than the traditional single application per JVM model, the increase in resource efficiency outweighs the increased implementation efficiency.

4.4.2 Compressed versus Uncompressed Class Files

JAR (Java ARchive) files are used to store together some or all of the class files that are required to execute an application. Two approaches are used to form a JAR file: (1) compress all the class files individually and then combine them together to form the JAR file, or (2) combine all the class files together and then compress the resulting JAR file. These two approaches impact how the class files of an archive are accessed at run time, and the resulting size of the JAR file.

If the first approach is used, *compress-then-aggregate*, when an application requires a method at run time, the section of the JAR file that contains the associated class file is read into memory, this class file is then decompressed, and the method is interpreted or compiled. Because the JVM accesses the class files in their compressed format, we say *compressed* class files are used.

On the other hand, if the second approach is used, *aggregate-then-compress*, all class files are decompressed into RAM when the Itsy boots. Any JVMs that are started later can access these archives directly without having to uncompress them first. Thus, when an application requires a method, the class file has already been decompressed. Because the JVM accesses the class files in their uncompressed format, we say *uncompressed* class files are used.

The advantage of the aggregate-then-compress approach is that the resulting JAR file tends to be smaller, and thus, less persistent storage is required to store it. For example, the JAR file containing the basic class files for our standard Java libraries is 500 Kbytes under aggregate-then-compress and 800 Kbytes under compress-then-aggregate. However, the uncompressed size of the basic class files is 1500 Kbytes. Clearly, there is a run time memory cost of decompressing the entire JAR file should only a portion of it be required.

These two approaches not only differ in their static and dynamic memory footprints, but also result in different energy and time costs when a method is required. Figure 7 presents the energy and power costs when the JVM accessed uncompressed JAR files (aggregate-then-compress) and compressed JAR files (compress-then-aggregate); in this figure, we plot power and energy in different graphs. For the composite and chess traces, using uncompressed JAR files does not save any power or energy, while for the web trace, there is a 5% saving in average power and total energy.

The lack of energy saving in the first two traces is due to two competing effects. The use of uncompressed JAR files reduces the startup time of the applications, thus reducing both the energy and time required to perform the tasks specified by the traces. However, because the events in the traces are time stamped and thus are not replayed until the correct inter-event time spacing has occured, the execution times of the traces do not change (see Table 2 for the execution times). Since energy is consumed even while processes are idle, the overall energy saving from the use of uncompressed JAR files is less than that which would occur without the time stamping of events. The idle power cost is $\approx$0.3 Watts, as shown in Figure 4 during the period of time in which there was no user input to the drawing application. In comparison, the web trace keeps the system busy for a

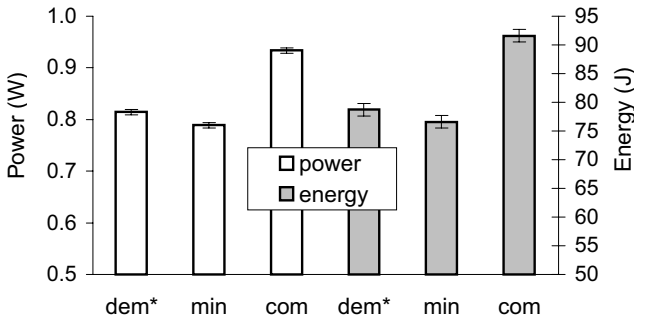

Figure 8: The energy consumption of the composite trace when two class preloading variants, minimal (min) and complete (com), are used instead of on-demand (dem) class loading.

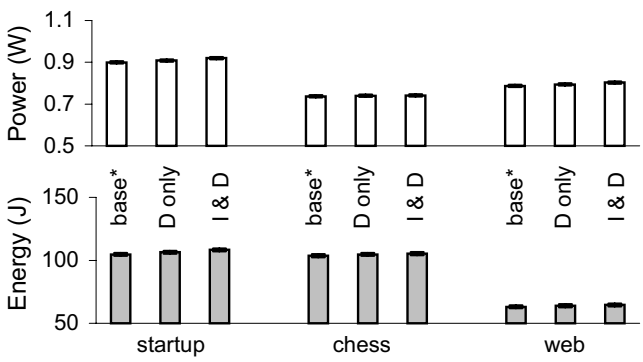

Figure 9: The energy consumption of the three traces when the baseline cache flushing algorithm (base) is replaced with two variants: complete data cache flush (D only), and complete data cache flush combined with instruction cache invalidate (I and D).

greater proportion of the time, and hence, idle time effects are less important.

4.4.3 Class Loading and Just-in-time Compilation

Java provides for very late dynamic linking. Instead of linking all required code for a given application statically into one big binary, Java classes are loaded into the virtual machine as they are needed. While dynamic class loading allows for programming techniques that require very late binding, such as the techniques used to implement mobile code, a great deal of work must be done at execution time instead of at compile time.

The just-in-time compilation or translation of Java bytecode into native instructions is done lazily as well. As a method is translated, references to other yet untranslated methods are filled with *trampolines*. Trampolines are small pieces of code that, when invoked, translate a method and invoke it. After the method is invoked the first time, the trampoline is destroyed. Subsequent invocations will directly reach the translated method.

The loading and linking process takes time. Aside from the energy costs, it also increases the time users have to wait for applications to start. We hid some of the startup latency through the use of a low-priority thread that runs when the JVM would be otherwise idle, and preloads a set of predetermined classes and translates the associated methods.

For preloading to be successful, the number of preloaded classes that turn out not to be needed must be minimized. Otherwise, unnecessary memory, time, and energy costs will be incurred. If no class mispredicts occur, the amount of work done and energy consumed will be roughly equal to that if preloading is not used. However, startup time will be reduced.

To illustrate the effects of choosing the preload set poorly, we assembled two preload sets: the *minimal set*, which includes only the classes required to load and begin executing the composite trace; and the *complete set*, which includes a large number of classes, many of which are not used by any of the applications that are exercised by the composite trace. Figure 8 shows the average power and the total energy consumed by the composite trace when on-demand loading is replaced with either of these two preloading variants. In all cases, a single JVM configuration was used. Observe that there is $\approx$ 18% difference in energy consumption between the two variants.

4.4.4 Cache Flushing after Code Generation

The translation of Java bytecode into native code may incur additional costs depending on the support provided by the computer system for memory coherence. In particular, once new native instructions are written to a sequence of memory locations, references to these locations must return the new instructions. For processors designed with separate instruction and data caches, such as the SA-1100 processor, this requirement can be met by flushing the data cache once the code has been generated and before the new sequence is referenced.

In addition, if the processor does not keep the instruction cache coherent with main memory, it may be necessary to flush the instruction cache as well. Flushing the instruction cache is only necessary if regions of memory are reused and the code stored in those regions is replaced. In most cases, a JVM can use bookkeeping information from its memory allocator to determine whether the instruction cache must be flushed or not.

The SA-1100 processor provides a mechanism for flushing only specific lines of the data cache, and thus, the entire data cache need not be flushed after code is generated. Our baseline design takes advantage of this feature of the SA-1100. Further, because our JVM keeps track of the memory regions that may contain stale code and does not reuse these, we never have to flush the instruction cache.

To obtain an estimate of how expensive these memory-system-induced penalties are for the startup process, we considered the energy impact of the worst cases: flushing the complete data cache at every method translation, or invalidating the complete instruction cache at every translation in addition to flushing the data cache. The energy consequences of these two worse-case scenarios on our three traces are given in Figure 9. Observe that for our traces, cache flushing has little energy impact, a likely result of the SA-1100 having fairly small caches (16 Kbyte instruction and 8 Kbyte data) and its ability to enter a low-power mode during the resolution of a cache miss.

4.4.5 Interpreting and Just-in-Time Compilation

We now turn to the run-time options. First, as noted in Section 4.1, just-in-time compilation reduces execution time and therefore energy consumption for a method if the

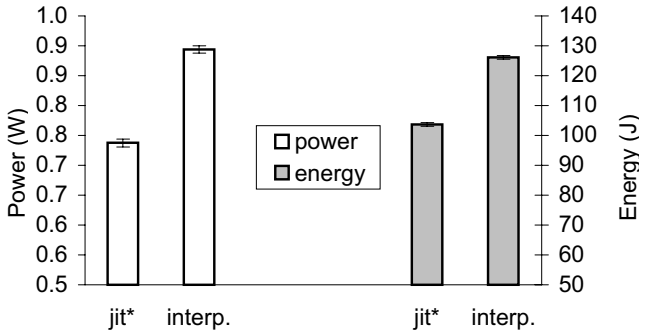

Figure 10: The energy consumption of the chess trace when the interpreter is used instead of just-in-time compilation.

time saved by translating the method exceeds the time spent translating it. For infrequently invoked methods, interpretation may be the better option. As the Kaffe JVM does not support choosing at run time whether to interpret or compile a method, for our experiments, we compared the cost of using only a just-in-time compiler to using only the interpreter.

Figure 10 presents the average power and the total energy consumed by the chess trace for both of these bytecode translation options. For this trace, interpretation results in an increase of 21% in the average power and 22% in the total energy. Further, while in both cases the total execution time was nearly the same, the amount of time that the processor was idle with the use of the just-in-time compiler was greater.

While our results show pronounced benefits from using a just-in-time compiler, we note that we do not expect such large differences to occur with the use of other JVMs and different interpreters. First, the Kaffe interpreter is very simple and does not include optimizations that are often found in production interpreters. These optimizations include the use of special routines written in assembly to interpret frequently used bytecode instructions. Second, more modern JVMs are capable of choosing whether to interpret or compile on a method-by-method basis at run time. Nonetheless, our comparison provides insight into the amount of energy that might be saved should a system be able to support the larger memory footprint of a just-in-time compiler.

4.4.6 AWT Polling Frequency

The AWT implementation for the Itsy relies on periodic polling of the attached input devices. The default polling interval is 0.03 seconds. If there is a sequence of related input events, such as a sequence of touchscreen `down`, `drag`, and `up` events resulting from a touchscreen stroke, this delay will only affect the initiating `down` events. The AWT is designed to poll for events that are expected (such as `drag` or `up` events following a `down` event) in a tight loop.

If the polling interval is short, more time and hence more energy is spent polling for events. If the polling interval is long, less time is spent polling: however, users may perceive the device as sluggish in its response, especially if the polling interval is chosen to be much larger than 0.25 seconds. Choosing an extremely large polling interval also has the effect that a task takes longer to execute overall; more time is spent idling between when an event becomes pending and when it is dispatched. Depending on the cost

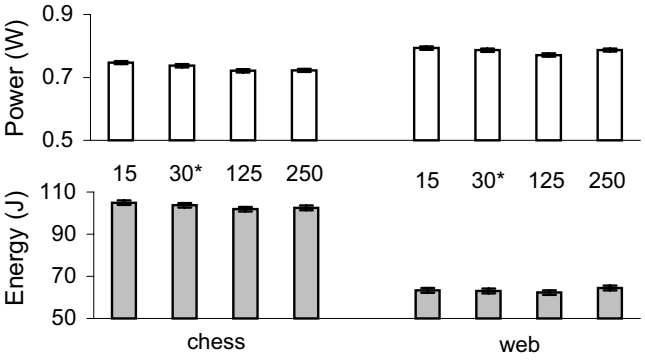

Figure 11: The energy consumption of the web and chess traces when the AWT polling interval is varied between 15 milliseconds and 250 milliseconds.

of idling, this delay may increase overall energy costs.

The effect of the AWT polling interval on average power and total energy consumption is shown in Figure 11. As the polling interval is increased from 0.015 seconds to 0.125 seconds, energy consumption decreases. The chess trace uses 3% less energy with a polling interval of 0.125 seconds than at 0.015 seconds. Even though the total execution times tend to increase as the polling interval is increased (see Table 2), the drop in average power – due to a larger fraction of idle time – accounts for the decrease in energy. However, when the polling interval is increased to 0.25 seconds, the energy usage increases slightly. We believe this increase is due to an artifact of how we replay events from the traces and is not a property of the JVM.

5. RELATED WORK

We believe this work is the first to characterize the energy usage of a high-performance pocket computer, and also the first to measure the energy impact of running Java.

Ellis [1] has characterized the power usage of a Palm Pilot, a pocket computer offering significantly lower performance that the Itsy, and has provided coarse estimates of the energy cost of various tasks. Martin [9] has also measured the effect on battery life of running a Itsy Pocket Computer at different clock frequencies.

A number of studies have profiled the energy consumption of notebook computers. Li et al. [13] report that of the power consumed by a notebook, the display consumes 68%, a disk consumes 20%, and the CPU and memory consume only 12%. Other studies have provided a more detailed characterization of power consumption for notebooks [14; 15; 16].

The PowerScope energy profiler maps energy usage to application structure by reporting the energy consumption of processes and procedures [17]. This tool has been used to provide detailed profiles of the energy usage of applications running on laptop computers [18]. We are currently exploring how the methodology described in this paper can be combined with PowerScope to produce similar profiles for applications running on the Itsy Pocket Computer.

In a broader scope, there has been considerable work in low-power or energy-conscious computer system design. This includes circuit-level design issues [19], CAD tools [20], optimizing the memory subsystem [21; 22], reducing the waste from speculative execution [23], instruction-level software

optimizations [24], and CPU scheduling with voltage scaling [7; 25; 26].

6. CONCLUSION

Pocket computers must operate on battery power for extended periods of time. Application design in this environment should therefore treat energy and power usage as primary concerns. In this paper, we have presented a two level characterization of the energy consumption of the Itsy Pocket Computer, with the aim of furthering the software community's understanding of designing software to be energy efficient. We began by characterizing the energy consumption of the Itsy hardware and three important subsystems. Then, we evaluated the energy and power impact of several tradeoffs in the design of a Java Virtual Machine for that environment.

There are four key messages of this study. First, the energy and power usage in pocket computers differ markedly from that of notebook computers. Compared to a ThinkPad notebook computer, the Itsy exhibited a wider dynamic range of power consumption, a result of two factors: the processor used in the Itsy exhibited a wider range of power demand; and the power consumption of the memory system accounted for a greater fraction of the overall power consumption of the Itsy. We believe these factors to be true in general of pocket and notebook computers, with the result that the variation in power usage during typical application scenarios is much higher than for notebook computers.

Second, using a single JVM to run multiple applications can have considerable energy benefit. Executing our composite trace with a single JVM reduced energy usage by 25%, demonstrating that current research efforts to safely execute multiple applications within a single JVM can be of significant benefit in extending the battery life of pocket computers.

Third, preloading Java classes can reduce startup time without impacting energy consumption, if one can accurately predict which classes will be used. Measures to reduce the delay a user observes when starting applications, can be undertaken without additional energy costs if additional work is avoided. For instance, preloading Java classes reduces startup time while not affecting energy costs, only if misprediction is averted.

Fourth, techniques that reduce overall execution time, primarily just-in-time compilation, are likely to provide significant energy savings for pocket computers. These savings are likely to outweigh the costs associated with them, which we believe to be mainly increased memory consumption and added implementation complexity.

Acknowledgments

We would like to thank the members of Compaq's Western Research Lab and Systems Research Center who created the Itsy Pocket Computer and built the infrastructure that made this work possible. We would also like to thank the anonymous referees for their comments.

APPENDIX

Table 2 presents the execution time, power and energy for each JVM design option presented in the paper.

A. REFERENCES

[1] C. S. Ellis. The case for higher-level power management. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems*, pages 162–167, March 1999.

[2] Board on Army Science and Technology, National Research Council, Washington DC. Energy-efficient technologies for dismounted soldier, 1997.

[3] Transvirtual Technologies Inc. Kaffe Java Virtual Machine. `http://www.transvirtual.com`.

[4] M. A. Viredaz. The Itsy Pocket Computer version 1.5: user's manual. Technical Report TN-54, Western Research Lab, Compaq Computer Corporation, July 1998.

[5] R. Stephany, K. Anne, J. Bell, G. Cheney, J. Eno, G. Hoeppner, G. Joe, R. Kaye, J. Lear, T. Litch, J. Meyer, J. Montanaro, K. Patton, T. Pham, R. Reis, M. Silla, J. Slaton, K. Snyder, and R. Witek. A 200MHz 32b 0.5W CMOS RISC Microprocessor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 15.5–15.5–9, February 1998.

[6] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.

[7] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of The First ACM International Conference on Mobile Computing and Networking*, pages 13–25, November 1995.

[8] J. Flinn, K. I. Farkas, and J. Anderson. Power and energy characterization of the Itsy Pocket Computer (version 1.5). Technical Report TN-56, Western Research Lab, Compaq Computer Corporation, February 2000.

[9] T. L. Martin and D. P. Siewiorek. The impact of battery capacity and memory bandwidth on cpu speed-setting: a case study. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 200–205, August 1999.

[10] J. Gosling, B. Joy, and G. L. Steele, Jr. *The Java Language Specification*. The Java Series. Addison-Wesley, September 1996.

[11] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, October 1996.

[12] G. Back and W. C. Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, March 1999.

[13] K. Li, R. Kumpf, P. Horton, and T. Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the 1994 Winter USENIX Conference*, pages 279–291, January 1994.

	Time (seconds)	Power (Watts)	Energy (Joules)
Composite Trace			
baseline	116.4	0.900	104.7
use a single JVM	96.6 (0.83)	0.815 (0.91)	78.7 (0.75)
use compressed JAR files	117.3 (1.01)	0.912 (1.01)	107.0 (1.02)
preload class files, using minimal set	97.1 (1.00)	0.789 (0.97)	76.6 (0.97)
preload class files, using complete set	98.1 (1.01)	0.934 (1.15)	91.6 (1.16)
data cache flush	117.1 (1.01)	0.909 (1.01)	106.5 (1.02)
data cache flush, instruction invalidate	117.8 (1.01)	0.920 (1.02)	108.3 (1.03)
Chess Trace			
baseline	140.7	0.737	103.7
use compressed JAR files	141.5 (1.01)	0.747 (1.01)	105.8 (1.02)
data cache flush	141.5 (1.01)	0.740 (1.00)	104.7 (1.01)
data cache flush, instruction invalidate	142.2 (1.01)	0.742 (1.01)	105.3 (1.02)
interpreted	141.1 (1.00)	0.894 (1.21)	126.1 (1.22)
AWT polling frequency = 15 ms	140.9 (1.00)	0.747 (1.01)	104.9 (1.01)
AWT polling frequency = 125 ms	141.3 (1.00)	0.721 (0.98)	101.9 (0.98)
AWT polling frequency = 250 ms	142.1 (1.01)	0.722 (0.98)	102.5 (0.99)
Web Trace			
baseline	80.2	0.787	63.0
use compressed JAR files	80.5 (1.00)	0.824 (1.05)	66.4 (1.05)
data cache flush	80.6 (1.01)	0.794 (1.01)	63.9 (1.01)
data cache flush, instruction invalidate	80.4 (1.00)	0.803 (1.02)	64.6 (1.02)
AWT polling frequency = 15 ms	79.8 (1.00)	0.793 (1.01)	63.3 (1.00)
AWT polling frequency = 125 ms	80.7 (1.01)	0.771 (0.98)	62.3 (0.98)
AWT polling frequency = 250 ms	81.9 (1.02)	0.787 (1.00)	64.4 (1.02)

Table 2: Execution time, power and energy for each JVM design option presented in the paper. The numbers in parenthesis are ratios over the baseline, with the exception of the preload experiments where the ratio is over the single JVM case. The values in this table are the average of five trials. The measurement error is $\pm$ 0.005 Watts for the power data and $\pm$ 1.1 Joules for the energy data.

[14] J. Lorch. A complete picture of energy consumption of a portable computer. Master's thesis, University of California, Berkeley, 1995.

[15] J. Lorch and A. J. Smith. Energy consumption of Apple Macintosh computers. *IEEE Micro*, 18(6):54–63, Nov/Dec 1998.

[16] T. Ikeda. Thinkpad low-power evolution. In *IEEE International Symposium on Low-Power Electronics*, pages 6–7, October 1995.

[17] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, February 1999.

[18] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems and Principles*, pages 48–63, December 1999.

[19] W. Nebel and J. Mermet (Eds.). *Low Power Design in Deep Submicron Electronics*. Kluwer, 1997.

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

[21] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick. The energy efficiency of IRAM architectures. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 327–337, June 1997.

[22] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 184–193, December 1997.

[23] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings 25th Annual International Symposium on Computer Architecture*, pages 132–141, June 1998.

[24] V. Tiwari, S. Malik, and A. Wolfe. Instruction-level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13:223–238, 1996.

[25] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of The ACM*, 36:74–83, July 1993.

[26] T. D. Burd and R. W. Brodersen. Processor design for portable systems. *Journal ov VLSI Signal Processing*, 13(2/3):203–222, August 1996.