

Energy-driven Statistical Profiling: Detecting Software Hotspots

Fay Chang, Keith Farkas, and Parthasarathy Ranganathan
Compaq Western Research Laboratories

{Fay.Chang, Keith.Farkas, Partha.Ranganathan}@Compaq.com

Abstract

Energy is a critical resource in many computing systems, motivating the need for energy-efficient software design. This work proposes a new approach, energy-driven statistical profiling, to help software developers reason about the energy impact of software design decisions. We describe a prototype implementation of this approach for the Itsy pocket computing platform. Our experimental results using the prototype with 13 benchmark programs show that (i) energy measurement tools that ignore system/kernel effects can give erroneous results about energy hotspots, and (ii) using execution time profiles to develop intuition about energy usage can often be misleading.

Key words: energy measurement tools, statistical profiling, energy-efficient software

1 Introduction

To develop energy-efficient software, programmers need to understand the energy impact of their design decisions. When reasoning about such decisions, programmers tend to rely on intuition. Unfortunately, intuition can be misleading because most programmers do not internalize an accurate model of the energy cost of different operations, or properly account for the relative frequency of the operations that take place. In this paper, we describe a tool that helps programmers develop energy-efficient software by helping them evaluate the energy impact of their design decisions. In particular, our tool enables programmers to quickly determine the energy required to execute some software and the energy hot spots within that software. With this information, programmers may then employ techniques, such as those described by Simunic et al. [9], to improve energy efficiency.

Our tool is based on *energy-driven statistical sampling*. While statistical sampling is not a new idea (e.g., [1, 5]), we introduce the idea of using energy consumption to drive sample collection. As illustrated in Figure 1, we interpose a simple *energy counter* between the energy supply and the system under study. This counter measures the energy consumed by the system and causes an interrupt to be generated on the system whenever a predefined amount of energy (i.e., an *energy quanta*) has been consumed. The system handles these interrupts by executing a particular interrupt service routine that will record *samples* identifying the program instructions that were interrupted. The recorded samples are processed, off-line, to generate energy consumption estimates for each application, procedure, and instruction.

Prior work has proposed several other tools for helping programmers evaluate the energy impact of their design decisions. Most of those tools are not based on statistical sampling. Unlike our tool, they rely on estimates of the energy cost of using various system-dependent hardware components, and generally require either binary modification or special compiler support [4, 8, 3]. Therefore, our tool is more portable and easier to use. In contrast, like our tool, PowerScope [6] is based on statistical sampling and requires neither binary modification nor special compiler support. However, it relies on time-driven statistical sampling. By relying on energy-driven statistical sampling, our tool delivers better accuracy for a given amount of overhead, and offers the potential for energy consumption estimates that are more precise.

The key contributions of this paper are three fold.

- First, we introduce a new approach – energy-driven statistical sampling – to evaluating the energy impact of software design decisions.

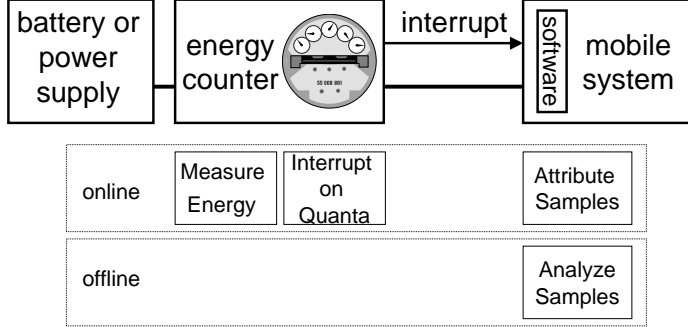


Figure 1. High-level overview of energy-driven statistical sampling.

- Second, we describe a prototype implementation of this approach for the Itsy pocket computing platform [7], and present experimental data that validates this implementation.
- Third, we present the results of an energy-consumption study of 13 benchmarks. Our results emphasize that that (i) energy measurement tools that ignore system/kernel effects can give erroneous results about energy hotspots, and (ii) using execution time profiles to develop intuition about energy usage can often be misleading.

The prototype that we describe in this paper attributes the energy consumed by all system activity to the software modules that are executing while this activity is on-going. Thus, for example, should a software module turn on the backlight for a minute, the energy consumed by the backlight during this minute will be attributed to all of the software modules that execute during this minute. While this approach to attributing energy consumption gives programmers insight into system-level energy consumption, it does not identify the modules that are actually responsible for such background activity. However, attributing the energy due to background activities is not relevant for this paper, since none of our benchmarks initiate such activities. We are currently exploring more sophisticated attribution approaches, one of which we discuss in the future work section of this paper.

The rest of the paper is organized as follows. In Section 2, we begin by discussing our prototype and an error analysis of this prototype. In Section 3, we discuss the results of our study of 13 benchmarks. Section 4 discusses the key differences between our approach and prior approaches. Finally, Section 5 concludes and discusses ongoing and future work.

2. Itsy-based Prototype

We have built a prototype implementation of energy-based statistical profiling on the Itsy version 2.0 pocket computing platform [7]. This section describes the hardware and software components of our prototype, and experimental results validating this prototype.

2.1 Hardware

As illustrated in Figure 2, we replaced the Itsy's battery with a power supply, and interposed an *energy counter* between the power supply and the Itsy electronics. We used a power supply rather than the battery to simplify our experimental procedure and to ensure that the Itsy's power efficiency stays constant during experiments.¹ The purpose of the energy counter is to generate an interrupt whenever a predetermined amount of energy has been consumed. The energy counter operates as follows.

As current i_s is drawn from the power supply by the Itsy, a current mirror comprising a resistor and a current-sense amplifier generates a current $i_m = \alpha \times i_s$ (where $\alpha = 1.5 \times 10^{-3}$ in our implementation). This current i_m

¹The Itsy includes a number of voltage regulators whose voltage conversion efficiency depends on the supply voltage. As the Itsy's battery discharges, the battery voltage drops, which causes the efficiency of the regulators to drop. Consequently, at lower operating voltages, more power is dissipated as heat. By using a power supply, our results are not affected by this effect.

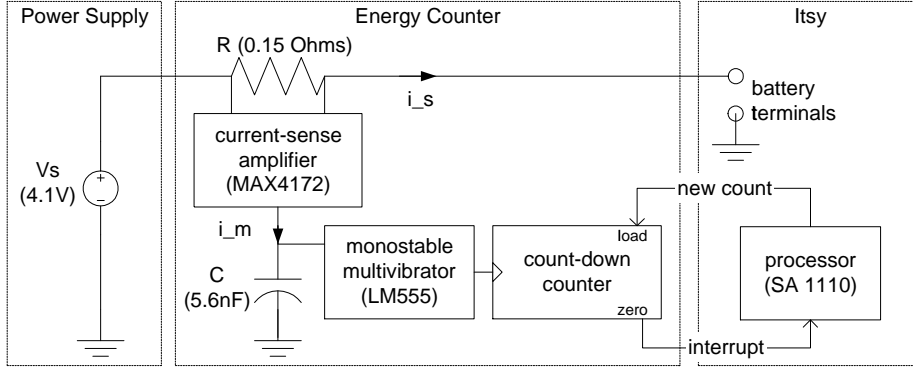


Figure 2. The Itsy energy-driven statistical sampling prototype. The energy counter is interposed between the power supply and the Itsy pocket computer.

deposits charge on the positive plate of a capacitor, which acts as a current integrator. When the voltage across the capacitor plates reaches $\frac{2}{3}$ of the supply voltage (V_s), a monostable multivibrator generates an output pulse P , and discharges the capacitor to a voltage of $\frac{V_s}{3}$. The capacitor then begins to accumulate charge again via i_m .

Thus, each pulse P indicates that the capacitor (with capacitance C Farads) has accumulated $Q_c = \frac{C \times V_s}{3}$ Coloumbs, during which time the Itsy has consumed $Q_i = \alpha \times Q_c$ Coloumbs. Since the voltage powering the Itsy stays constant at V_s , the energy consumed by the Itsy during this time is simply $V_s \times Q_i$. We refer to this quantity as the *energy quanta*. The energy quanta in our implementation is about $30 \mu\text{J}$.

Each pulse decrements the value of a count-down counter. When this counter's value reaches zero, the counter generates an interrupt request by asserting a signal that is connected to one of the Itsy's general purpose I/O lines (GPIOs). The value of the counter is reset by software, as described in the next section. Thus, the energy counter allows the amount of energy consumption that will generate an interrupt to be varied dynamically.

2.2 Software

The software for energy-driven statistical sampling is divided into three components – (i) a kernel-level pseudo device driver and interrupt service routine, (ii) a user-level data collection daemon, and (iii) user-level data processing tools. This software has been written to run on top of Linux (version 2.0.30), the operating system run by the Itsy. Calls to the pseudo device driver control profiling (e.g. turn it on or off). When profiling is turned on, the kernel-level interrupt service routine handles interrupts raised via the above noted GPIO line. On each interrupt, this routine performs two tasks. First, it records a sample in a circular buffer. A sample consists of the identity of the process, the software module (executable or dynamically linked library), and the instruction within that module that was interrupted. Second, before the routine ends, it resets the value of the count-down counter discussed above. (Calls to the pseudo device driver can designate a single reset value, or a small range of values from which the interrupt service routine can randomly select a value.)

The user-level data collection daemon issues calls to the pseudo device driver to obtain sample data, and writes one or more sample data files. At any subsequent time, these files can be processed using the user-level data processing tools to produce human-readable energy profiles.

2.3 Error Analysis and Validation

In this section, we discuss the sources of error inherent in our prototype. Broadly, the sources of error can be classified into two categories: (i) energy measurement related, and (ii) attribution and analysis related. The following two sections discusses each of these in detail.

2.3.1 Measurement-related Errors

Energy is given by $\int^t v(t)i(t)dt$, where $v(t)$ is the instantaneous voltage and $i(t)$ the instantaneous current. Yet, the prototype integrates only the current, thus introducing a source of measurement-related error. This source of error is not significant, however. First, the power supply we used provides a *regulated* output of $4.1V \pm 1mV$. Second, while the sense resistor between the power supply and the Itsy renders the voltage supplied to the Itsy dependent on the amount of current the Itsy draws, the Itsy supply voltage varied by less than 2% of the nominal value. If, on the other hand, we had used the Itsy battery, our analysis indicates that good accuracy could be obtained with periodic sampling of the voltage

A second source of measurement-related error is the inherent non-ideal characteristics of electrical components. However, such errors are negligible in our prototype owing to careful component selection.

Finally, a third source of error is the impact that the energy counter and the sampling software has on the resulting sample distribution. This error is dominated by the heat dissipated by the 0.15 Ohm resistor, which, for our benchmarks, was at most 17mW. To put this number into perspective, the maximum power consumed by our benchmarks is 1.4W.

In terms of the energy consumed by the profiling software, we measured the energy consumed by the pseudo device driver and by the user daemon. Figure 3 summarizes these measurements. For each of our benchmarks (see Figure 4), this table presents execution time, the number of samples collected, the average sampling frequency, and the approximate execution time overheads in the interrupt service routine and the user daemon as a fraction of total execution time. As we can see from Figure 3, the overheads of both the interrupt service routine and the user daemon are negligible for all our benchmarks.

2.3.2 Attribution and Analysis Error

The other major class of errors is attribution and analysis errors.

To accurately attribute a sample to an instruction, it is important to minimize the delay between the following two events: the monostable multivibrator generating a pulse P that will decrement the counter to zero (see Section 2.1), and the interruption of the program in execution so that the interrupt can be serviced. This delay is composed of the time required for the interrupt to be conveyed to the processor core, and the time required for the processor to begin servicing the interrupt. In our prototype, the former is negligible.

The latter, however, can noticeably affect attribution. The Itsy pocket computer uses a StrongARM SA1100 processor. One characteristic of this processor is that, before servicing a trap or exception, the processor first completes execution of the instructions in the *decode* and later pipeline stages. Thus, when the energy-counter interrupt is processed, the sample will be attributed to the instruction that was fetched *after* the instruction that was in execution at the time that the interrupt was delivered to the core. This property leads to poor sample registration and possible skews in the attribution.

To evaluate the extent of this problem, we performed a number of experiments with carefully designed micro-benchmarks comprising interleaved groups of high latency and low latency instructions and procedures. Using a time profiler, we determined that less than 2% of the samples were incorrectly attributed at the procedure level, but significant discrepancies occurred at the instruction level. Hence, if fine-grained attribution (at the instruction level) is required, it may be necessary to adjust the attribution of samples by analyzing the samples offline using a model that includes a detailed model of interrupt handling, the pipeline scheduling rules, and the instruction delay and energy parameters for the SA1100 processor.

A second source of error arises if phases of the application being profiled are synchronized exactly with the discharging of the capacitor. Although none of our benchmarks exposed this error, our prototype avoids this error by allowing for a small dynamic jitter in the number of quanta after which an interrupt will be generated (as described in Section 2.2).

Application	Time (sec)	Samples	Average frequency (Hz)	ISR overhead	daemon overhead
cpu-bound-1	193	85047	441	0.2%	0.25%
cpu-bound-2	444	125556	282	0.18%	0.20%
idle	299	38950	130	0.11%	0.01%
MPEG	171	190458	1113	1.89%	0.50%
synth	375	222207	592	1.23%	0.11%
synth2	377	300731	797	1.55%	0.13%
voice	41	12143	297	0.30%	0.15%
midi	342	140678	411	0.57%	0.02%
wave	141	42068	298	0.26%	0.01%
zip	30	23335	782	1.13%	0.29%
ftp	55	16166	291	0.35%	0.49%
xwin	71	17677	248	0.42%	0.13%
edemo	365	181695	498	0.86%	0.17%

Figure 3. Summary of overheads with energy profiling

Finally, a third source of error concerns sensitivity to the sampling rate. The accuracy with which a sample distribution reflects the true allocation of energy consumption for an application is proportional to the square root of the number of samples [1]. However, larger number of samples can lead to greater processing overhead and increased profiler intrusiveness. For our benchmarks, our default parameters resulted in the collection of 12,000 to 125,000 samples at 130Hz-1100Hz sampling frequencies. Sensitivity experiments with longer execution times and higher sampling frequencies indicated that this setup did not have appreciable errors. We also studied several carefully-designed micro-benchmarks with known energy profiles. For all of these benchmarks, the energy profiles reported by our tool matched very closely the expected energy profiles.

3 Application analysis

Figure 4 describes our thirteen benchmarks and their execution times. The benchmarks were chosen to cover a wide range of activities. They include two contrived CPU-intensive benchmarks, a variety of media processing benchmarks, file handling and file transfer benchmarks, and two more complex benchmarks (xwin and edemo) that include a combination of the functionality of several of the other benchmarks.

3.1 Application profiles

Figure 5 summarizes the profiles for our thirteen benchmarks. For each benchmark, the bar on the left indicates the (self-normalized) profile generated by time profiling and the bar on the right indicates the (self-normalized) profile generated by energy profiling. Each profile is divided into four components – the application modules, the kernel-idle loop, other kernel modules, and miscellaneous library modules.

As Figure 5 indicates, the system energy consumption is important on many of our benchmarks. While the simple CPU-intensive benchmarks do not show a significant kernel component, the non-idle kernel component is large in eight of the remaining benchmarks. These results indicate the importance of energy measurement tools accounting for system effects. Second, the results indicate that time and energy profiling can often give significantly different results. Seven of the thirteen benchmarks in our study show significant differences between time and energy profiling. These differences are particularly evident for benchmarks that cycle through multiple energy states. The most dominant example of this is the difference between the energy consumed by the kernel-idle loop versus other portions of the code. For our system, the power consumed by the idle loop is about 120mW compared to the 330mW consumed when the CPU is busy.

Figure 6 once again summarizes the profiles for our thirteen benchmarks, but this time with the kernel-idle component factored out. As before, for each benchmark, the bar on the left indicates the (self-normalized) profile generated by time profiling and the bar on the right indicates the (self-normalized) profile generated by energy profiling. Each profile is now divided in greater detail into the top five application modules (App-1 to App-5)

Application	Description	Time
cpu-bound-1	A simple CPU-intensive application that spends its time in two procedures dominated by loop-based computation (addition)	193 seconds
cpu-bound-2	Multiple runs of a simple CPU-intensive application with loop-based computation interspersed with sleep times	444 seconds
idle	5 minutes of Itsy in idle mode	299 seconds
MPEG	Six iterations of the MPEG benchmark playing the movies bigtoy.mpg and swars2.mpg	171 seconds
synth	Speech synthesis of the speech.txt file (samples scaled by 0.25)	377 seconds
synth2	Speech synthesis of the speech.txt file (samples scaled by 8)	375 seconds
voice	Recording dictation of a single paragraph	41 seconds
MIDI	MIDI benchmark (timidity) playing the m3-spain.mid file to completion	342 seconds
wave	Four iterations of playing the Itsy.wav sound file	141 seconds
zip	Three instances of tarring, zipping, and removing a 573KB directory	30 seconds
ftp	FTP'ing a 573KB file	55 seconds
xwin	Operations on the X-windows environment including minimizing, maximizing of windows, use of character recognition, and the xeyes benchmark	71 seconds
edemo	An e-mail demo distributed as part of the Itsy distribution. Our experiment included voice-activated turning on of the unit, several instances of voice-activated mail reading, one video attachment, one voice mail composition, and usage of buttons to navigate.	365 seconds

Figure 4. Applications studied

and the top two kernel modules (Kernel-1 and Kernel-2). As we can see from Figure 6, time and energy profiling continue to yield different results even with the idle time factored out. However, in this case, the differences are important only on four of the benchmarks – MPEG, voice, wave, and x-win. Once again, the differences can mainly be attributed to benchmarks that transition through multiple energy states, particularly through use of different hardware resources that can consume a substantial amount of power (e.g., display, audio, modem, etc.). In particular, they are likely to be further emphasized in multi-programming or other workloads that are more complex than the benchmarks we consider in this study.

3.2 Observations

Based on our results, we can make two key observations.

- System-level components contribute significantly to the energy consumption of the system. Even factoring out the kernel-idle time, several of our benchmarks still spend large amounts of energy in various kernel modules. This implies that energy monitoring tools that only focus on the application-level (similar to some that exist today) may not be presenting a complete picture of the energy consumption of the system.
- Time profiles do not accurately reflect energy consumption in many cases. Such inaccuracies are particularly pronounced when software cycles through multiple energy states (e.g., idle mode versus CPU-busy, use of hardware devices like audio, microphone, backlight, etc.), which is common in practice. This implies that energy-monitoring tools and optimizations that use time consumption as a first-order approximation of energy consumption are likely to lead to erroneous conclusions with many real-world workloads. As discussed in the next section, this result also has implications regarding the use of time-driven energy sampling approaches like PowerScope [6].

4 Comparison with Other Approaches

Two kinds of approaches have been proposed with the aim of providing programmers with insight into the energy impact of software design decisions.

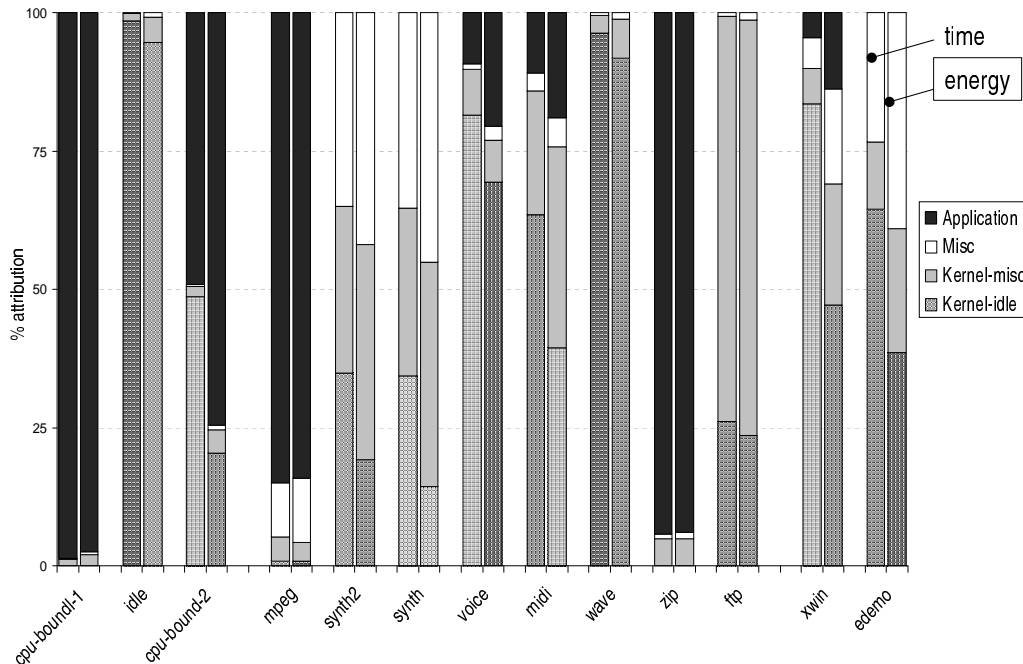


Figure 5. Comparison of profiles from time and energy profiling

4.1 Activation Models

Activation-model approaches require two components: (1) energy-use estimates for specific system activities, such as executing a specific type of instruction, idling the processor, accessing memory or a disk, and sending or receiving messages over a network; and (2) activation counts for each activity, which are obtained by counting the number of times each activity occurs when an application runs. Energy consumption for the application is then obtained by multiplying the activation counts by the energy cost per activation.

Activation-model approaches differ widely in how they obtain activation counts. They can be classified as counting-based, sampling-based, or simulation-based approaches. With counting-based approaches, activation counts are obtained by either running instrumented versions of applications (e.g., Millywatt [4]), or running (unmodified) applications on an operating system that is modified to leverage hardware event counters (e.g., Joule Watcher [2]). PowerMeasure/StateProfiler [8] is a sampling-based approach in that some activation counts are estimated by periodically sampling the system's state. Finally, with simulation-based approaches (e.g., SimplePower [10] and Wattach [3]), a specially-designed simulator is used to collect the activations of interest for an application.

One advantage of activation-model approaches is that they can leverage activation counts for background activities to more easily attribute energy consumed asynchronously. The main disadvantage of activation-model approaches is the difficulty of ensuring their accuracy. In particular, the accuracy of these approaches depends on tracking all important activation counts, and on obtaining accurate energy-use estimates for all important system states. Obtaining such estimates is non-trivial for a single platform, let alone many platforms. In contrast, our tool requires neither intuition about what activities may be of interest nor energy-use estimates for specific activities. While asynchronous energy attribution is currently a problem for our tool, we are working on addressing this limitation; Section 5 discusses one solution we are considering.

4.2 Statistical Sampling and PowerScope

A second approach, which was developed by Flinn and is the basis for his PowerScope tool [6], is to augment the information gathered by a time-driven statistical sampler to include power usage. While a number of implementation differences exist between our tool and PowerScope, this section focuses on the conceptual differences.

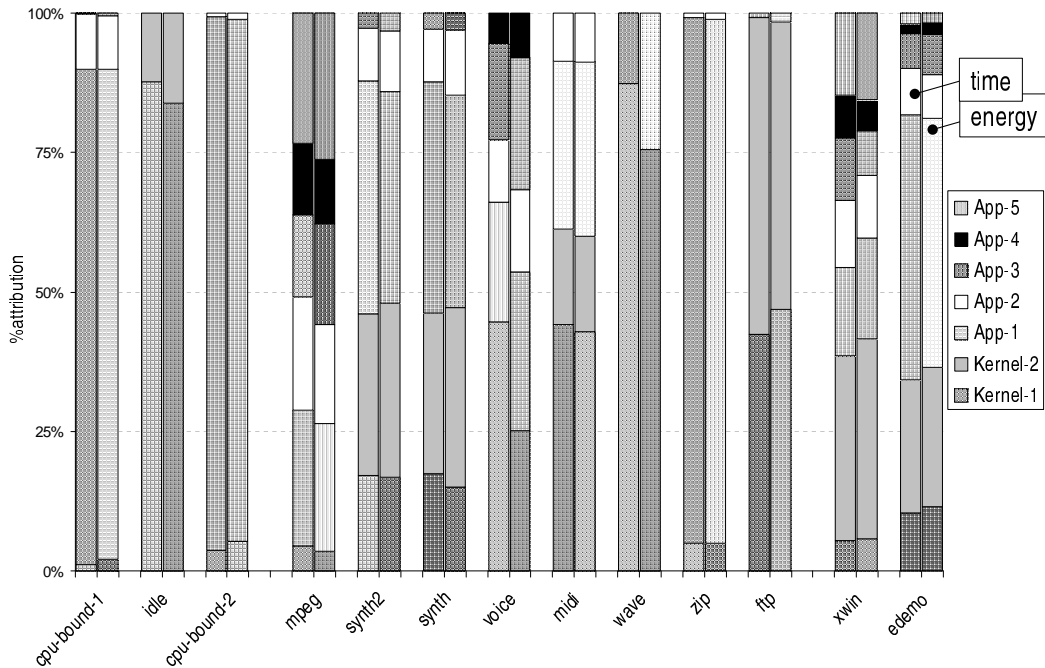


Figure 6. Comparison of profiles from time and energy profiling – idle time factored out

PowerScope uses a timer to generate a stream of interrupts, each separated in time by a fixed time interval. On each interrupt, in addition to recording information about the interrupted software module, PowerScope records an estimate of the amount of power the system was using immediately prior to the interrupt's occurrence. More exactly, PowerScope records the instantaneous current being drawn by the system. Later, during offline processing, PowerScope estimates the energy used by the system between the occurrence of the $(i - 1)^{th}$ and i^{th} interrupt by multiplying the current sample i , the (assumed constant) supply voltage, and the time between the two interrupts.

One noteworthy difference between our tool and PowerScope is that sample collection is driven by energy consumption not time. Consequently, for a given amount of system overhead, our tool will more quickly identify energy hot spots than will PowerScope. PowerScope can achieve similar accuracy if it is run longer or with a higher sampling frequency. Another point of comparison relates to the frequency of interrupts during periods of low energy usage. Under a time-driven approach (e.g., PowerScope), samples will be collected even when the system is consuming relatively little energy, such as when the processor is in a light sleep mode. Consequently, the act of collecting samples will significantly affect the energy profile of the system. In comparison, in such a situation, an energy-driven approach will generate much fewer interrupts, thereby perturbing the system much less.

A final point of comparison is that PowerScope samples the current while our tool uses a capacitor to integrate the current. To investigate the impact of this difference, we have examined the extent to which current sampling and current integration may give different results. We have also implemented the PowerScope approach for our test system and are in the process of evaluating the differences that occur in practice.

Current Sampling Assessment

The Itsy's current shows considerable variation. For example, Figure 7 shows the current-consumption profile of the Itsy while running two of our benchmarks, MPEG and MIDI. This profile was obtained by sampling the Itsy's current 10,000 times a second using a data acquisition system. As can be seen from the figure, the variation in the current is dominated by pronounced peaks.

To assess the information contained in such peaks, we used a 1 GHz digital oscilloscope to examine the fre-

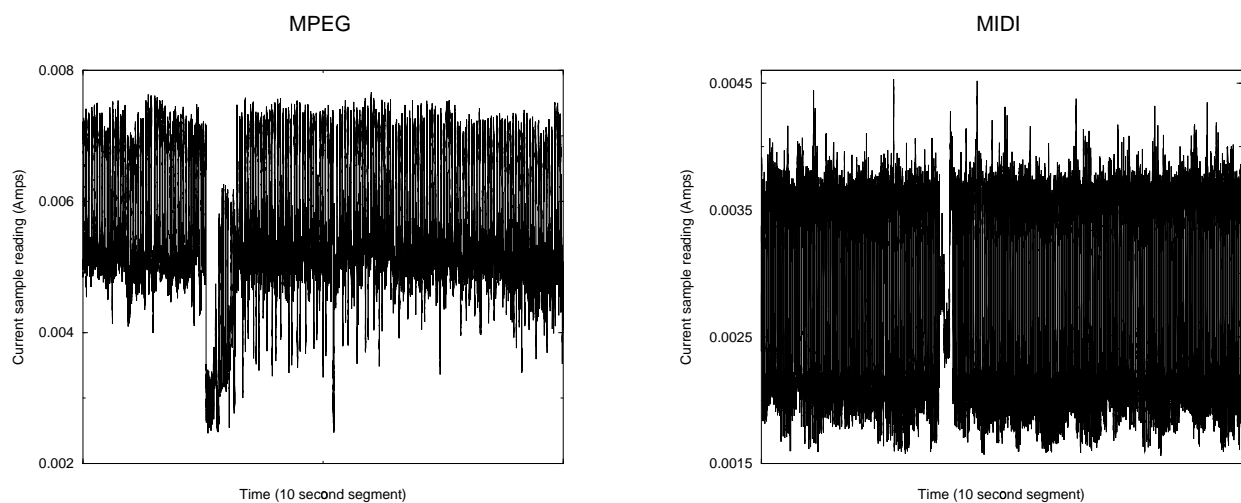


Figure 7. Variation in current samples across two benchmarks, MPEG and MIDI

MPEG			
Procedure	Time	Energy	Pscope
mpegVidRsrc	24.13%	22.73%	22.08%
mpeg-j-rev-dct	20.08%	17.65%	17.89%
GrayDitherImage	14.66%	17.89%	17.80%
ParseReconBlock	12.68%	11.48%	11.26%
-start	08.64%	09.94%	10.28%
mpegaudio-Misc	04.86%	04.55%	04.75%
Kernel-misc	04.47%	03.45%	04.19%
Kernel-idle	00.79%	00.76%	01.02%
libc	07.39%	09.04%	08.67%
Misc	2.30%	02.51%	02.06%

MIDI			
Procedure	Time	Energy	Pscope
Kernel-idle	63.49%	39.42%	51.48%
Kernel-emulate	16.11%	25.96%	21.89%
Kernel-misc	06.28%	10.35%	07.85%
Timidity	10.95%	18.97%	14.97%
Misc	03.17%	05.30%	03.81%

Figure 8. Profiles obtained with a time-only profiler, our energy profiler, and PowerScope for the MPEG and MIDI benchmarks.

quency content of the Itsy's supply current. In particular, we measured the current drawn from the Itsy's battery when the Itsy was running our two MPEG clips. For each, we captured five 20 msec traces of the battery current, and then used the oscilloscope's FFT function to compute the FFT of each trace with a 50 Hz resolution and 250 KHz bandwidth. Finally, we determined for each FFT curve the frequency at which 50% and 75% of the area under the FFT curves was obtained relative to 0 Hz. The results of this analysis were that, in order to capture 75% of the information in the battery current, the battery current must be sampled at the Nyquist rate of approximately 60 KHz for one clip and 22 KHz for the other. Further, to capture just 50% of the information would require 22 KHz and 4 KHz sampling rates respectively. The difference between the 50% and 75% rates indicates the presence of significant information at frequencies above 11 KHz and 2 KHz respectively. Thus, periodic sampling at low frequencies of the current, as is done by PowerScope, may be inaccurate compared to our approach of integrating the current in the analog domain if the same number of samples are collected in both cases.

Preliminary PowerScope Comparison

To evaluate the extent to which the high variance in battery current matters, we implemented a version of PowerScope on our base Itsy system that samples current 100 times a second. Our implementation is similar to Flinn's [6], with the key difference being that we used a data acquisition system rather than a digital meter. Figure 8 present a comparison of the profiles generated by a time profiler, our energy-driven profiler, and our version of PowerScope for the MIDI and MPEG benchmarks, respectively.

PowerScope addresses a lot of the inaccuracies associated with the time profiler, as demonstrated by the results for the MPEG benchmark. However, as seen from the results for the MIDI benchmark, PowerScope attributes too much energy to the Kernel idle loop. This discrepancy arises because the current drawn by the Itsy while idle is not constant. Rather, due to background system activity (e.g., real-time clock updates, daemon activations), there are occasional pronounced current peaks of short duration. Because our version of PowerScope samples at 100 Hz, sampling such a peak can have a significant effect.

Although these results are preliminary, we were surprised by the lack of a greater difference between energy-driven profiling and our version of PowerScope. One possible reason may be that the high-frequency information in the current waveform is not distributed evenly in time, so that repeated time-based sampling of the instantaneous power does not yield significant errors. We are currently examining this and other possibilities.

Finally, it should be noted that both our current tool and PowerScope cannot correctly attribute the energy used by background activities.

5 Summary and Future Work

While the issues with designing applications to reduce the execution time are fairly well understood, there does not exist a similar understanding about designing applications to reduce their energy consumption. This work presented a new approach, *energy-driven statistical profiling*. Tools developed with this approach can help the software designer both reason about the energy impact of software design decisions and identify application energy hot spots.

Energy-driven statistical profiling uses a small amount of hardware to trigger an interrupt at pre-defined quanta of energy consumption. The interrupt is used to collect information about the program currently executing, and the information thus collected over the course of the program is used offline to generate an energy profile of where energy is being spent in the workload. We have developed a prototype system on the Itsy mobile computing environment and our studies on the prototype indicate that energy-driven statistical profiling approaches can provide an accurate system-level energy profile of the software with very little extra overhead.

Our results using the energy profiler to study the behavior of 13 benchmarks programs show that a non-trivial amount of energy is spent by the operating system. Additionally, there are often significant differences between the profiles generated by time and energy profiling, especially in workloads that transition between multiple energy states. Our results have important implications for future studies that try to identify and optimize energy hotspots in software. Specifically, (i) energy measurement tools that do not study system/kernel effects can give highly erroneous results about energy hotspots, and (ii) using intuitions based on execution time profiles to approximate energy usage can be very inaccurate in many cases.

As part of our ongoing efforts, we plan to extend our work to account for background power usage. One solution we are exploring is to augment the energy-counting hardware of a system to include software-readable counters that track the amount of energy consumed by background energy consumers (e.g., the backlight, a wireless radio). These counters could then be used by our interrupt service routine to assign fractions of the sample (based on energy usage) to system energy-use categories (e.g., backlight on, wireless active). In this way, a programmer may identify major energy consumers, and with further investigation, the software modules responsible.

We are also evaluating the version of our profiler we describe here as well as our extended versions with the PocketPC environment on the iPAQ pocket computer. Finally, we are also using the insights from our studies to identify optimizations to make applications more energy-efficient.

References

- [1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: where have all the cycles gone. In *Proceedings of the 16th Symposium on Operating Systems Principles*, October 1997.
- [2] F. Bellosa. The benefits of event-driven energy accounting in power sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.
- [3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, June 2000.
- [4] T. L. Cignetti, K. Komarov, and C. Ellis. Energy estimation tools for the Palm. In *Proceedings of the ACM MSWiM'2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.
- [5] X. Zhang et al. Operating system support for automated profiling and optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.
- [6] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 2–10, February 1999.
- [7] W. R. Hamburgren, D. A. Wallach, M. A. Viredaz, L. S. Brakmo, C. A. Waldspurger, J. F. Bartlett, T. Mann, and K. I. Farkas. Itsy: Stretching the bounds of mobile computing. *IEEE Computer*, 34(4), April 2001.
- [8] J. Lorch and A. J. Smith. Energy consumption of Apple Macintosh computers. *IEEE Micro Magazine*, 18(6), November/December 1998.
- [9] T. Simunic, L. Benini, and G. De Micheli. Energy-efficient design of battery-powered embedded systems. In *Proceedings of the International Symposium on Low-Power Electronics and Design '98*, June 1998.
- [10] W. Ye, N. Vijaykrishan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proceedings of the Design Automation Conference*, June 2000.