

Energy Estimation Tools for the *Palm*TM

Todd L. Cignetti, Kirill Komarov, and Carla Schlatter Ellis

Department of Computer Science
Duke University
Durham, NC 27708 USA
{cignetti,kirill,carla}@cs.duke.edu

Abstract

*Reducing the energy consumed in the use of mobile and wireless devices is becoming a major design challenge. While the problem obviously must be addressed with improved low-level technology, we have advocated also considering a higher-level view in which energy management becomes an explicit design goal of the software developer who can be more aware of the needs of applications. In support of this objective, new programming models, measurement tools, and simulation environments must be developed to provide the developer with feedback on the energy implications of various design decisions. In this paper, we describe an energy model and an execution-driven simulator incorporating this model for the *PalmOS*TM family of devices.*

1. Introduction

Reducing the energy consumed in using mobile/ wireless devices, thereby extending the lifetime of the batteries that power them, is one of the major challenges in designing such systems. While this problem can be addressed at various levels (e.g., by improving battery technology, by engineering low-power electronics and components, and by designing more efficient computer architectures), we have advocated designing applications and system software with energy consumption as a primary measure of performance [2, 12]. However, currently any attempts by software developers to tailor the energy use to the specific needs of applications are frustrated by several factors: a lack of feedback on what effect various design decisions might have on energy use, an inadequate programming model of power consumption in the target platform, a gap between our vision of how the application should control its energy use and

the ability to express that to the system, and limitations on the flexibility of interactions between the operating system and the given architecture. Identifying and alleviating these drawbacks in the state-of-the-art of energy conservation are among the goals of our Milly Watt Project¹.

In this paper, we describe an energy model for the *PalmOS*TM family of devices and a suite of tools including an execution-driven simulator. We present a power measurement framework which is based on the model and used to set empirically-derived parameter values for the simulator. The development of a simulation environment that can provide useful information to the programmer faces a number of challenges: The first step that must be taken is the development of an abstract energy model. The model must provide a simplified but accurate picture of how energy consumption is related to the actions of an application. In this way, any causality between the application source code and the energy consumption of the device can be exploited. The next step is to design experiments that accurately characterize the power consumption of the device in terms of the model. These values are then used as parameters in a simulator incorporating the abstract energy model. Our simulator is implemented as an extension of the Palm OS Emulator (POSE) that is a major component of the PalmOS program development environment [9]. For good predictions of energy consumption, the timing in the emulator must also be calibrated to reflect the actual device. Finally, the results of executing applications on the simulator must be presented in a meaningful fashion to the user. We describe our approach to each of these issues.

The remainder of the paper is organized as follows: In the next section, we describe the target device and the existing POSE environment which serves as a starting point for our development. Section 3 presents the energy model.

¹The name was inspired by Reddy Kilowatt, a mascot for the electric power industry, whose heyday was in the 1950's.

Section 4 describes the enhanced simulator with the energy model embedded inside. Section 5 describes the measurement methodology and results used to parameterize the simulator. Then, we show the visualization tools for the simulation results. Finally, we present related work and conclude.

2. Background

The target platform for this work is the *PalmTM* family of organizers. In particular, the timing and power measurements have been performed on the Palm IIIe (IBM Workpad). This model is based on the Motorola Dragonball processor, with 2MB of memory and a backlit, monochrome LCD display. It also has infrared (IR) and a serial port for communicating with other devices (hotsync cradle to a desktop machine, add-on modems). It is powered by two AAA batteries and runs PalmOS version 3.1. The advertised estimate on battery life is approximately 2 months with typical use. The standard applications include a calendar, to do list, address book, email, memo pad, and various games.

The Palm OS Emulator (POSE) [9] was the starting point for the development of our energy simulator. It was developed from an earlier application called Copilot [4]. Source code for POSE is available. POSE is a hardware emulator that allows testing, profiling, and debugging of PalmOS applications on a desktop computer rather than the PDA device. It is execution driven, fetching and interpreting instructions from both applications code and a ROM image of the operating system. A virtual Palm device is shown within a window on the desktop display and its screen is updated as applications run. Stylus and button input is simulated via the mouse and keyboard. It provides testing features not available on the device itself, including logging of events and functions, support for external debuggers, and automated random input generation (gremlins). External debuggers work with POSE to provide source-level debugging support. POSE provides an extensibility mechanism for intercepting system calls and changing their functionality (without access to PalmOS source). These are essentially head and tail patches that do additional or alternative processing before or after the ROM function executes.

3. Energy Model

It is impossible to address higher-level energy management without an understanding of what different operations cost in terms of power consumption. Unfortunately, few programmers have much intuition of how energy is used in a computing system. Thus, the first necessary step is to build a model with the following desirable properties: (i) the model components that represent energy use must be traceable to constructs in the source code in order to have meaningful predictive value for users and (ii) the model must

facilitate measurements of energy consumption that correspond to detectable energy-related events or functions on the device.

With these considerations in mind we have constructed a model of energy consumption. We chose a level of abstraction that reduces the complexity of the hardware state sufficiently, but that encompasses enough information to allow useful high-level energy optimization. The abstract model manages the problem of scale that is present in complex systems by encapsulating low level hardware details.

The energy model identifies a set of device power states and transitions between the states that are based on the hardware subsystems of the device. It assumes that the relevant transitions between states occur as the result of system calls, and that, by keeping track of system calls, the system may monitor its own energy consumption (one of our eventual objectives). There are several hardware subsystems that do not fit into this level of abstraction precisely, and these must be addressed separately.

For each separate hardware subsystem, a set of device states is defined. The device states are differentiated by the power consumption of the hardware during steady state (e.g. backlight ON versus backlight OFF). Each state is assigned a power consumption cost by measuring the steady state power consumption in that state (using the methodology described in Section 5.2). Each transition between states is assigned an energy consumption cost by measuring the transitional energy consumption.

The total energy consumed by the system is determined by summing the power of each device state multiplied by the time spent in each state plus the total energy consumption for all transitions.

Our model is based on the following premise: Significant device state transitions occur as a result of system calls and events (particularly those posted by interrupt handlers, ultimately recognized by a system call). Our model will apply to systems at the levels for which this premise is true, and we believe it to hold for many modern operating systems. Generally, user level programs get access to hardware devices via system calls (traps). Accessing devices through system calls provides the mechanism for the operating system to eventually be able to monitor the energy consumption of its devices by tracking device power states and transitions.

The current PalmOS does not provide as much protection for user programs as other operating systems. User programs have free access to memory mapped hardware devices. This is an artifact of the fact that the Motorola 68000 processor did not support virtual memory. Despite these limitations, the importance of providing operating system compatibility requires that well designed programs use system calls for device access. The practice of accessing hardware directly is discouraged by the PalmOS documenta-

tion [1]. We believe it is reasonable to assume application programs will use the conventional system call interface to access device hardware.

We found that, in PalmOS, our premise that device state transitions only occur as a result of system calls is only strictly true if we liberally interpret the definition of the system call to include shared library calls. In PalmOS, shared libraries are dynamically loaded libraries that are used to implement functions that multiple applications may use. A shared library function call first traps to a common shared library dispatcher which determines which library and which function to invoke[6]. By patching the shared library handler, the operating system can keep track of which shared library calls are made, and thereby keep track of device state transitions.

These power states and energy transitions can be applied to most hardware subsystems on a PDA, but there are subsystems that do not fit this model. One instance of such an exception is the power consumption of the CPU. Another special case is any device for which the power consumption states are not discrete.

The power consumption of the CPU and memory system vary depending on the mix of instruction types executed and the load on the system. There is not a system call that tracks this explicitly. In our model, we assume the CPU can be in one of three states: *sleep*, *doze*, *busy* [1]. Since system calls are required for the transitions between these states, the operating system can be aware of the coarse-grained power state of the CPU. Of course, there is variation in the power consumption within the busy state (idle and sleep states have a constant power consumption, as described in Section 5.4). The power consumption in the busy state may vary depending on many factors [11]. At the present time, we average these into one power consumption estimate for the CPU busy state. By contrast, a more complex model can be used within the simulator to model the power consumption of the CPU since it emulates the activity of the CPU on an instruction-by-instruction basis as it interprets code. This is planned future work, both to build the CPU model and to perform an instruction-level measurement study as in [11] to provide parameter settings.

Because our model uses discrete device states to model power consumption, devices with power consumption values that fall into a continuous range may be problematic. One case where this occurs is with the serial link state which is part of almost all communication from the Palm device. The serial port may be in the following states: open, closed, sending, receiving, or sending and receiving. The power consumed while sending or receiving is highly dependent on the bit rate. In order to be useful, the device state must account for the bit rate. When the power consumption falls into a continuous range, we break the continuous range into convenient discrete steps.

4. The Simulator

We have developed this simulation environment with the goal of facilitating development of software which pays attention to issues of higher-level power management. The simulator enables prototyping of energy-efficient application programs with substantial feedback given to the programmer. It provides flexibility for conducting studies evaluating the benefits of using new policies, mechanisms, and/or architectural features that are not currently available on the actual devices. It provides a convenient framework in which to experiment with proposed new features of an energy-aware API. For example, it allows investigation of cooperation between the operating system and application programs through an exchange of power-related information across that boundary that is relatively easy to incorporate in the simulator but difficult to implement since it involves modifying the actual OS interface.

The simulation environment is implemented as an extension of the Palm OS Emulator (POSE). POSE is a Windows-based application that simulates functionality of the Palm device, emulating its operating system and instruction execution of the Motorola Dragonball processors used in the Palm. We concentrated primarily on incorporating the power state model for the Palm into this existing environment. The transition states are currently not explicitly captured in the simulator's version of the power state model. The information in the output logs is sufficient to apply the energy costs of state transitions in the post-processing stage. Since measured instruction-level power consumption data is not currently available, fine-grain power state monitoring of the busy CPU within the processor emulator is not supported and an average power consumption value is associated with the busy CPU state.

4.1. Timing Calibration

Since energy expenditure depends on both time and power consumption, timings gathered from the simulator must be representative of an execution on the actual device. However, the simulator emulates PalmOS and does not run on the same platform or processor. Thus, straightforward time measurements of calculations done by simulator are not accurate. It is necessary to calibrate simulator time units to time units on a Palm. Timing in the emulator is governed not by the actual time but rather by the number of cycles the processor spends executing instructions. Therefore, any cycle may be adjusted to have an arbitrary duration in terms of actual time. We performed experiments to determine the correspondence between execution time on the actual Palm device and the virtual time in POSE in order to adjust the time calculations reported by the simulator. Timing measurements were taken at the initial stage of de-

velopment and after the simulator was close to completion. Since different types of instructions and system calls may affect the timing differently, timing was performed on applications that executed a mix of arithmetic instructions, mix of memory-access instructions, and mix of system calls. Different classes produce different scalings: $\frac{T_{Palm}}{T_{POSE}}$ ranges from .625 to .69 in our experiments. Because of disparity in timings between the actual device and POSE, necessary adjustments have to be done to POSE in order to assure the validity of the measurements and correspondence between clocks in POSE and actual Palm device.

In the POSE simulator, the functionality of the keys is simulated using the mouse or the keyboard. Similarly, tasks that are usually done using the pen in an actual Palm are also performed using the mouse. Therefore, when accounting for power and energy consumption of the pen and key components in the emulator, we need to keep track of mouse events. These include clicking of the mouse left and right buttons, and movement of the mouse. The cursor position at the time of the click determines whether there was pen or key activity. The difference is that pen events correspond to clicks in the display area, and graffiti writing area of the Palm, while the key events correspond to clicks on the hard buttons of the device. The described mouse events closely correspond to the PalmOS events that are generated in response to pen or key activity. Unfortunately, the timing of mouse activity is not representative of real pen and key user interface behavior on the Palm. This is an unavoidable limitation in the simulator when applications are used interactively in the simulator.

4.2. Energy Accounting

In the simulator, we currently identify eight major power-consuming components. These include the CPU, LCD, Backlight, Pen on the digitizer, Hardware Keys, Serial port, IR, and Sound. We account for power and energy consumption of the first six. Each of the components can individually be in some state. Each particular state of each component has an associated value of power consumption expressed in milliwatts. Given the power measurements for each individual component, it is easy to compute the power consumption values of the whole Palm device in each system-wide power state. For each power state, its power consumption value is calculated to be the sum of the power value consumed by each individual component of the Palm in that power state. Some components may contribute 0(mW) to the overall power consumption of certain power states of the Palm. After the initial state has been determined, the rest of the states are acquired by monitoring power state transitions. The power state transition occurs when one of the components of the currently active power state changes its state. In the simulator, the transitions be-

tween power states are caused by a certain system event or system call aimed at changing state of the particular component. For instance, the application may use HwrLCDSleep call in order to turn off the LCD. This call will also change the current global power state of the simulator.

For every power state encountered, we maintain a structure that keeps information about it, such as power and active components. Power usage is computed by determining states of all components of the Palm, and summing up their associated power values. Also, for each power state, we maintain the list of activation periods. Each activation period specifies the time when its associated power state was in force. The active power state is always timed starting and ending when a power state transition causes a change. A timer is reset upon the change of the power state. Using the starting and ending times, we are able to compute the power state's energy usage. When a transition out of a power state occurs, information about the active period of that power state is logged. This information includes starting and ending times of the active period, total energy consumed during that active period, owning application, and other related information. Owing application refers to the application program that has been running during the active period of the power state. In this way, we are able to compute statistics for each application. This data, in conjunction with the execution logs in the original POSE, allow useful analysis (see Section 6).

5. Power Measurement Method and Results

This section describes the methodology that was developed to quantify the power consumption of a device. We designed the methodology to be general enough to be used to characterize the power consumption of many devices, so first we describe the design goals of the methodology in general terms. Next we describe the experimental design and the programs implemented to take measurements on a device. Finally, we present selected measurements taken on the Palm III/e or IBM Workpad device. Such measurements provide the parameter values for the simulator.

5.1. Design goals

We intend that the measurements taken with this methodology can be used with the energy model developed in Section 3 and provide the power measurements for the tools that will use the model (Section 4). Our goal is that the experiments must be able to capture transient energy consumption, as well as steady state power consumption. It is also of primary importance that the measurements be accurate and that the act of taking measurements have minimal effects on the power consumed by the device. Further, we require that the experimental apparatus be as inexpensive

and widely available as possible. We limit the equipment to readily available laboratory instrumentation: a power supply, voltmeter, and oscilloscope.

Since the model specifies that state transitions be assigned an energy cost, our measurement methodology is able to capture transient energy consumption. This is important because the energy consumed by changing the device state may, in some cases, be significant. We have found that transient effects were much more difficult to measure than steady state power consumption, and most of the discussion of measurement methodology addresses the question of how to measure the energy of state transitions.

5.2. Experimental design

We connected a Workpad device to a power supply with an oscilloscope measuring the voltage across a small resistor, R_{test} (as shown in Figure 1).

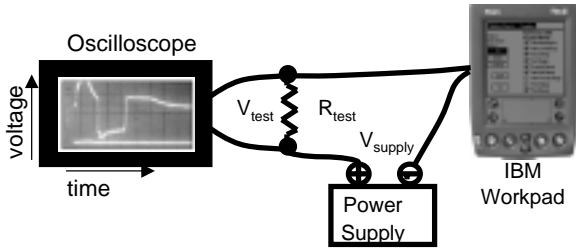


Figure 1. Experimental Setup.

The resistor R_{test} is necessary because our oscilloscope probes measure voltage and not current. Current probes that measure very small currents (milliamperes) are available for some oscilloscopes [10], but are relatively uncommon. The value of R_{test} must be small, so that the voltage drop across the resistor (V_{test}) does not make the voltage at the device terminals too low, but large enough that V_{test} is measurable. To make sure the voltage at the device is sufficient, we turn up the supply voltage so that the test which draws the most current ($max(I_{test})$) will not cause the voltage at the device terminals to be below the required minimum voltage ($V_{device_minimum}$).

$$V_{Supply} - max(I_{test})R_{test} > V_{device_minimum} \quad (1)$$

Since we did not know the maximum allowable device voltage, we conservatively chose the smallest R_{test} , such that equation 1 is true, and that produces a sufficiently measurable signal.

We compute the steady state power by multiplying the current times the voltage at the device terminals. This kind of measurement is possible with even the simplest instrumentation, substituting a voltmeter for the oscilloscope in Figure 1.

$$Power = (V_{Supply} - V_{test})I_{test} \quad (2)$$

We assume a constant supply voltage. One critical requirement of our experiments is a power supply that can provide sufficient current to the device over the course of the tests. If the power supply is current limited, the voltage level will drop when the device begins to draw too much current [13]. In our experiments we can assume that the voltage is constant because we use a power supply that can source 30 Watts at 3.5 volts (about 60 times more power than is used by the device). We have also verified that the voltage was constant using the second oscilloscope channel during the most power consuming tests.

Whereas the steady state power consumption is easy to measure with a voltmeter, the transient power consumption is hard to measure with only a voltmeter. It is not possible to determine the exact time of the system call or to differentiate the power consumed by starting the test from the test itself. Thus we use the oscilloscope to record the voltage as a function of time, starting when a voltage threshold level is exceeded. An in-band marker signal provides a clear reference point in the voltage versus time trace. The marker consists of a configurable delay before and after the test, and an in-band sync pulse before and after the test. The typical way of executing a test using in-band synchronization markers is the following:

1. *start test* – by pressing an on-screen button in the test application.
2. *delay* – configurable delay with processor in *doze* mode, using the SysTaskDelay() system call allows the transient effects of starting the test to settle.
3. *in-band synchronization pulse* – a subroutine with a known voltage versus time curve generates a large spike in the power consumption. This spike is used to trigger the scope to start recording (the scope is set to use a level sensitive trigger). The sync signal must be repeatable and, in order to reliably trigger the scope, should have a peak voltage level that is greater than the voltage levels before, during, and after the test.
4. *system call* – the system call that changes the device state.
5. *measurement period* – usually, a delay which allows the device state to settle. The processor is in *doze* mode. The exception is the tests of a *busy* CPU.
6. *system call* – the system call that restores the device state.
7. *end synchronization pulse* – a second time reference

8. *end delay*

There are several reasons that the experiments were designed this way. The in-band synchronization signal provides a convenient trigger for the oscilloscope, and a necessary time reference for the transient behavior of the system call. The delay allows the transient effects of starting the test to reach a steady state. There is a negligible amount of overhead when the delay timer expires, and immediately thereafter the system call is made. The system call causes a state change, and the subsequent delay allows the system to reach a steady state, while the oscilloscope is used to measure the transient effects of the state change. After the delay, another system call is made to restore the state, followed by another delay, followed by a synchronization pulse and a delay.

As we noted in Section 3, the power consumption of some devices is not directly influenced by system calls. These require exceptional tests. In the case of the CPU, it is sufficient to measure the power of the device in each of the three states: *sleep*, *doze*, *busy*. We compute the busy state power by putting the processor in a tight loop. The transition costs between CPU states are assumed to be negligible. In the case of the serial link, we parameterize the measurements using the bit rate and flow control as parameters.

External devices pose a difficult set of problems. They may require serial or other external communication rather than system calls to change states. For example, we are aware of several Global Positioning System (GPS) devices that are controlled over a serial link with serial command sequences. The energy consumption associated with these commands cannot be monitored without actually parsing the command string. Another complication external devices pose is they may have separate energy source, which the main device cannot access or monitor.

5.3. Measurement Programs

Several Palm applications have been built to help measure the power consumption of the device.

The *Power* and *Millywatt* applications provide a user interface to call some of the basic functions of the device for measurement intervals. *Power* puts the device into a steady state for a long (10 seconds) period of time, during which we can monitor the power consumption. Many of the tests originally performed with *Power* were subsumed with the development of *Millywatt*. The *Millywatt* application implements the configurable markers and delays described in Section 5.2 and is designed specifically for transient energy.

Another category of Palm applications are being used to validate the results of the simulator. The *Energy Monitor* program keeps track of the device state using the energy model described in Section 3 directly on the Palm. The program uses the *Hackmaster* [7] interface to patch system

calls. A patch routine is executed before the system call which records the time of the state transition. The current implementation can track the total running time, backlight on time, and LCD on time. While the current implementation does not monitor all of the device states, it serves to demonstrate the feasibility of a resident energy monitor running under the PalmOS, without the knowledge or cooperation of the operating system or application programs. In addition, several Palm applications have been instrumented to track the time spent in their own power states while they are running. When these programs are subsequently run on the simulator, results can be compared. This validation effort is on-going work.

5.4. Measurement Results

This section describes selected results of the power and energy measurements of a Workpad device (Palm III/e equivalent) running PalmOS 3.1. The *Power* and *MillyWatt* application programs were used to make these measurements. We used the following lab equipment: Power supply (HP 3611A - 0-20V, 0-1.5A), Multimeter (HP 34401A), and Oscilloscope (HP 54600B - 100 MHz, 2 channel + external trigger).

CPU, LCD, Backlight, Buttons, Pen

We measured the power consumption of the basic hardware subsystems of an IBM WorkPad device using our experimental setup. R_{test} was measured to be 7.451Ω , and V_{Supply} was 3.5 volts. Table 1 shows the results. The values are in milliwatts, and are relative to the default mode that is defined as: CPU doze, LCD ON, Serial port CLOSED, Backlight OFF, Buttons UP, Pen UP, Sound OFF. The system call in the last column is that which we associate with the device state. In most cases this is the system call used to change the device state. The CPU Busy state does not have a system call for reasons described previously. We associate button presses and pen taps with the `EvtGetEvent()` system call. These are handled at the lowest level by interrupts, the generation of events, and an intermediate level of system calls, but `EvtGetEvent` is the “visible” system call which notifies the application that an event occurred.

Serial link

The serial link is the hardware subsystem most critical for wireless or wireline communication since this is the standard way to connect a modem device to the Palm. Such modems generally have their own power source. The results do not reflect the power consumption of the attached external device, only the energy impact on the PDA itself. For each experiment, the user chooses the baud rate and flow control options.

Device	State	Power (mW)	System Call
CPU	Busy	104.502	none
	Idle	0.0	SysTaskIdle
	Sleep	-44.377	SysSleep
LCD	On	0.0	LCDWake
	Off	-20.961	LCDSleep
Backlight	On	94.262	HwrBacklight()
	Off	0.0	HwrBacklight()
Button	pushed	45.796	EvtGetEvent()
Pen	On screen	82.952	EvtGetEvent()
Pen	+Graffiti	86.029	EvtGetEvent()

Table 1. Steady state power of IBM WorkPad (relative to default mode defined in section 5.4), with associated system calls

System Call	Transient Energy (mJ)
CPU Sleep	2.025
CPUWake	11.170
HwrLCDWake	11.727
KeySleep	2.974
PenOpen	1.935

Table 2. Transient energy of IBM WorkPad for significant system calls relative to energy consumed in default mode defined in section 5.4 for the same amount of time.

The steady state power (in excess of the default mode) of sending on the serial link with the SerOpen, SerSetSettings(), and SerSend() system calls vary between 89.8 mW and 94.7 mW for various bit rates and flow control settings. The energy for a one Mbit send ranges from 308 J (300 bps, no flow control) to 1.6 J (56K, hardware flow control). The steady state power (in excess of the default mode) of receiving on the serial link ranges from 20.2 mW (300 bps, no flow control) to 109.3 mW (56K, hardware flow control) with the energy for a one Mbit receive ranging from 94.2 J (300 bps, hardware flow control) to 1.77 J (56K, no flow control).

Transition Energy

Only the following system calls were found to have non-negligible relative energy consumption during the transition period: *CPUSleep CPUWake HwrLCDWake KeySleep PenOpen*. Table 2 shows the results, with values in millijoules.

6. Programmer Feedback

A substantial part of this project included developing the graphical user interface that would convey information gathered using the simulator to the user. We have developed a package that allows creation, display, and dynamic runtime modification of various types of charts. This package is used to interpret information acquired using the simulator and display it graphically to the user. In addition, the chart package provides a convenient interface to construct, modify, and display charts. Currently, there are six types of charts supported in the package. Figure 2 shows a sample of data acquired from running of the simulator. This pie chart shows the total percentage of energy consumed by each power component state throughout an execution. Other charts include bar graphs representing each power

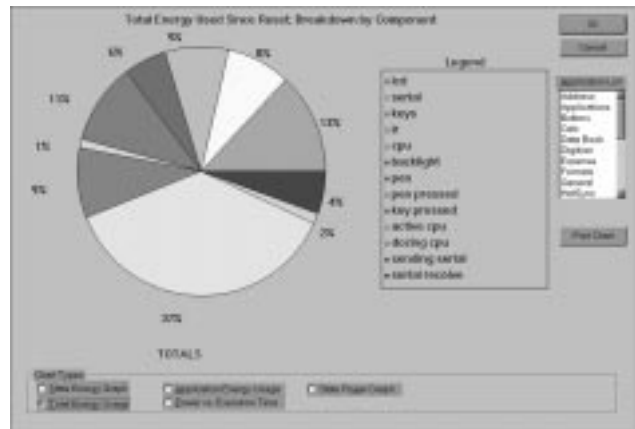


Figure 2. Pie Chart of Energy Use Breakdown.

state and its total consumed energy, broken down by each component, over all activation intervals of that state. Each of these components is listed in the chart legend. Data can be filtered and presented on a per-application basis. Another useful feature present in the simulator is the ability to log important execution events, producing traces. System calls and events are logged at varying granularity, producing traces of the system's and applications' execution.

7. Related work

The Advanced Configuration and Power Interface (ACPI) specification [5] is a lower-level OS/device interface, providing one framework for gross system-wide power states and per-component power states as a basis for the development of OS-directed power management. Our model is compatible with the ACPI framework.

Our model bears some resemblance to the instruction-level power estimation model of [11] (at a different level of abstraction, but for similar experimental reasons). We plan to pursue an effort to characterize the power consumption of the Dragonball at the instruction level in the near future.

The SimplePower simulator [15, 14] is a cycle-accurate, execution driven, simulator. This framework is an architectural level simulator that deals with the processor datapath, memory and bus energy. Integrating our device level model with such a finer grain, lower level model would yield an attractive hierarchical approach.

Our measurement work is related to the PowerScope toolkit [3] which combines hardware instrumentation with CPU statistical profiling techniques to map power consumption to program structure. In their method, the multi-meter interrupts the processor under test which seems intrusive. In addition, the approach of assuming a causal relationship between program counter and measured power may be problematic. Our approach is qualitatively different in the relationship between the multimeter and device (i.e. triggering method) and that the events traced represent power state transitions at a certain level of abstraction. This allows us to account for power consumption levels caused by an earlier, perhaps unrelated, activity but still in effect and observable (e.g. an earlier disk spinup for which the timeout has not yet expired).

This work is also related to the StateProfiler tool developed by [8]. The model is not fully generalized and the tools do not capture transient information.

8. Conclusion

In this paper, we have presented a model that captures the energy use characteristics of a popular mobile device, the Palm organizer. The model has served as the basis for both (1) a simulation framework for estimating the energy that might be used by Palm applications under development and (2) a measurement methodology that can supply the empirical power values for key components of the device. The model also lends itself to OS-level monitoring of energy consumption at power state transitions triggered by system calls. The contribution is to enhance a familiar program development tool for an existing mobile platform with the ability to evaluate the energy implications of higher level design decisions.

Future work includes incorporating a better CPU model with supporting measurements at the instruction level, providing a way to extend the model to capture the power states of external devices “hiding” behind the serial interface, such as wireless modems with their own power sources, and validating the results of the simulator against the results of the *Monitor* implementation. To be complete, the IR and sound devices need to be added to the simulator. The transient

state energy consumption also needs to be integrated into the simulator. Finally, more extensive analysis tools are needed to answer causality questions, linking the observed energy consumption to source code constructs.

References

- [1] C. Bey, E. Freeman, and J. Ostrem. Palm OS programmer's companion. <http://www.palmos.com/dev/tech/docs/palmos/>, 1999.
- [2] C. S. Ellis. The Case for Higher-Level Power Management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 1999.
- [3] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Workshop on Mobile Computing Systems and Applications (WM-CSA)*, pages 2–10, February 1999.
- [4] G. Hewgill. Copilot: Design and Development. *Handheld Systems*, 6(3), May/June 1998.
- [5] Intel Corporation, Microsoft Corporation, and Toshiba Corporation. Advanced Configuration and Power Interface Specification. <http://www.teleport.com/acpi>, December 1996.
- [6] J. Ishaq. Mastering shared libraries. <http://www.wwg.com/newsview/shared.shrml>, 1999.
- [7] E. Keyes. How to write a hack. <http://www.daggerware.com/hackapi.htm>.
- [8] J. Lorch. A complete picture of the energy consumption of a portable computer. Master's thesis, University of California, Berkeley, December 1995.
- [9] K. Rollin. The Palm OS Emulator. *Handheld Systems*, 6(3), May/June 1998.
- [10] Tektronix. Current measurements. http://www.tek.com/Masurement/Products/catalog/Accessories/probes/current_measurements/, 1999.
- [11] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, December 1994.
- [12] A. Vahdat, A. Lebeck, and C. Ellis. Every Joule is precious: The case for revisiting operating system design for energy efficiency. In *SIGOPS European Workshop*, September 2000. to appear.
- [13] J. Van der Spiegel. Basics of power supplies: Use of the HP e363a programmable power supply. <http://www.ee.upenn.edu/rca/instruments/HPpower/PS-3631A.html#SimplifiedView>, January 1997.
- [14] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. A unified energy framework with integrated hardware-software optimizations. In *International Symposium on Computer Architecture (ISCA)*, June 2000.
- [15] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. Irwin. The Design and Use of SimplePower: A cycle-accurate energy estimation tool. In *Design Automation Conference*, June 2000.