# Software Power Measurement

Dushyanth Narayanan
dnarayan@microsoft.com

April 26, 2005

Technical Report
MSR-TR-2005-51

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

## Abstract

Effective system-level power management requires cheap, accurate and fine-grained power measurement and accounting. Unfortunately current portable hardware does not provide this capability. We advocate *software power measurement*: estimation of power consumption by modelling it as a function of device state. The approach requires no additional hardware, and allows fine-grained, per-device and per-application power measurement. We describe a design and implementation of software power measurement, and a feasibility study showing significantly better accuracy than power profiling based on time averaging. We conclude with design recommendations for OS designers and portable hardware vendors to improve the ease and accuracy of power measurement.

## 1 Introduction

Energy is a critical resource for many computing systems. While battery life is especially relevant to portable and hand-held computers, peak power consumption affects fan noise on desktops and cooling costs for server farms. There is an increasingly recognised need to manage and account energy as a first-class resource within the operating system [13].

Energy management requires accurate measurement and accounting. Adaptive tuning of device parameters such as disk spin-down timeouts [3] requires accurate estimates of *per-device* power consumption. Per-device measurements at fine time granularity — when combined with existing OS accounting of devices such as CPU, disk, and network — also enable *per-application* accounting of energy consumption. This is of great value both for end-users ("Outlook is responsible for 80% of your battery drain, maybe you should kill it") and for application-level adaptation [5].

Unfortunately, current approaches to energy measurement have several drawbacks, especially when applied to laptop and hand-held computers. Accurate measurement with fine time granularity requires external hardware such as sampling digital multimeters, making the approach unwieldy and hard to deploy in the field. Unmodified laptop hardware typically offers nothing more than Smart-Battery measurements, which are only accurate at coarse time granularities and measure the power consumption of the entire system but not of individual devices.

We propose a novel technique known as *software power measurement* (SPM), which correlates infrequent, coarse-grained measurements of power with fine-grained observations of device state and activity. The result of the correlation is a *predictor* that estimates the energy consumption over arbitrarily short time interval from from the observed device state and activity.

The remainder of this paper is organised as follows. Section 2 describes current approaches to the problem and their drawbacks. Section 3 describes the design and prototype implementation of software power measurement on Windows XP. Section 4 presents a quantitative evaluation of the prototype, demonstrating both the feasibility of the approach and the limitations of the current implementation. Section 5 briefly describes related work, and Section 6 concludes the paper.

## 2 State of the Art

Current approaches to power measurement are either impractical, expensive, or coarse-grained. They fall into one of three broad categories:
**On-board hardware**: Many modern laptops and hand-helds can query battery drain information through the standard SmartBattery [11] interface. However, this does not provide per-device power consumption. Additionally, SmartBattery implementations filter the raw measurements before providing them to the BIOS, typically through averaging over some unspecified time period. This makes it impossible to get accurate measurements at fine time granularity.
**Multimeter-based sampling**: Power usage can be sampled at high frequency by instrumenting the power supply of a portable computer with a multimeter [6]. However, multimeters are bulky, expensive, and impractical in the field, and do not provide per-device power usage.
**Offline micro-benchmarks**: Carefully designed micro-benchmarks followed by differential analysis can tell us the power cost of each device in each power state [2, 7]; we can then estimate instantaneous power consumption from the current state of each device. However, benchmarking each possible hardware platform and device is a tedious task. The approach we describe in this section can will construct such "power profiles" automatically, online, and in the background.

Our aim is to design a power measurement strategy that is simultaneously

- *online*: low-overhead, automated, continuously running in the background on end-user platforms.
- *SmartBattery-based*: requiring no additional hardware on today's laptop computers.
- *fine-grained*: precise energy accounting to individual devices at fine time granularity.

## 3 Design and Implementation

How can we get accurate, high-frequency, per-device power measurements without additional hardware? *Software* power measurement is based on the observation that power consumption is a function of device state and device actions: CPU clock speed, disk spin-ups, etc. If we can identify all relevant states and actions, and correlate them to measured power at large time scales, then we can estimate power consumption at short time scales directly from the observed device states and actions. In other words, we can use device usage statistics as a *proxy* for power consumption, by

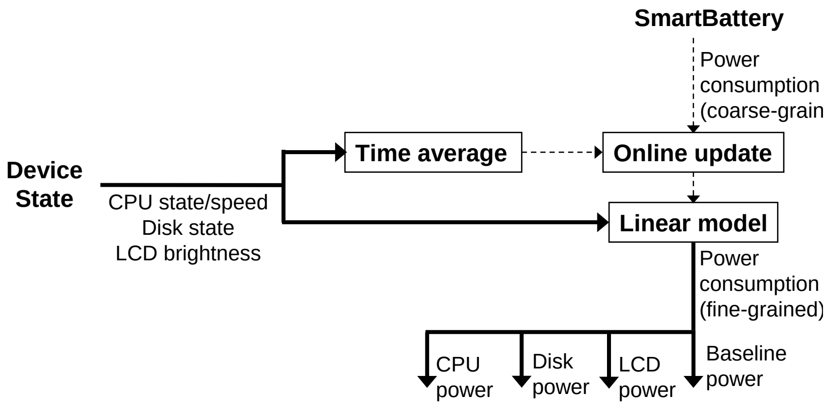

Figure 1: Power estimation from device state

- *tracking* device actions and state changes
- *measuring* SmartBattery statistics over relatively long time scales and matching the measurements to time-averaged device usage statistics.
- *modelling* power consumption as a function of device state and activity by computing the energy cost $E_a$ of each device action $a$, the power cost $P_s$ of each device state $s$, and the background power consumption $P_{bg}$.
- *estimating* per-device consumption over any time interval $t$ by applying the model to the observed device usage:

  $$E_{device} = \sum_{actions} E_a n_a + \sum_{states} P_s t_s$$

  Here $n_a$ is the number of times action $a$ was observed, and $t_s$ is the amount of time spent in state $s$. The total power consumption

  $$E_{total} = P_{bg} t + \sum_{devices} E_{device}$$

- *accounting* this power to processes by accounting the underlying device usage.

Figure 1 shows graphically the relationship between device parameters, power consumption, and the resulting predictive model. Our current prototype models three devices of interest for power management on portable/laptop computers: CPU, disk, and display. Table 1 shows the state variables and actions currently tracked for each device.

Our implementation platform is Windows, and we use Event Tracing for Windows (ETW) [9] to track device behaviour. ETW is a low-overhead mechanism for tracing kernel and application events with cycle-accurate timestamps.

| Device | State/action |
|---|---|
| CPU | Idle (C) state |
| CPU | Speed (P) state) |
| Disk | Sleep (D) state |
| Disk | Spin-up (action) |
| Disk | Spin-down (action) |
| Screen | Brightness level |

Table 1: Device states tracked for power measurement

Currently it can track context switches, disk accesses, and network send/receive events, and attribute them to the appropriate process. We have added additional events to the Windows kernel to track disk spin-ups and spin-downs, and changes in processor clock speed. For the LCD display, the main parameter of interest is the *backlight state*: whether it is on or off, and what the brightness level is. On our test platform, this information is not exposed to the OS from the vendor-specific binary video driver. In our feasibility study, we set the brightness level by hand, and provide the known brightness level as input to the model.

While tracing is done on the live system, analysis and modelling are currently done offline using scripts that parse the event trace logs to track the state and activity of each device. Given precise time-stamped events for each state change, we know the power state of each device at each point in time. This information is correlated with SmartBattery measurements to compute the power cost of each individual device state as well as the baseline power consumption.

The SmartBattery interface provides two ways to measure power consumption:

1. Spot readings of battery drain current

2. The current battery charge level

Thus, we could measure average power consumption by averaging a series of spot readings, or by measuring the change in battery charge over time. The first method has the disadvantage that certain power states can never be measured. For example, querying the SmartBattery interface cannot be done when the CPU is idle, thus we can never get spot readings when the CPU is in states $C1$, $C2$, or $C3$. A further disadvantage is that the spot readings are not really instantaneous, but are averaged over an unknown time duration specific to the proprietary vendor implementation of the SmartBattery interface. Without knowing the time period for which a power measurement is valid, it is impossible to know which device states it corresponds to.

For these reasons, we measure power by tracking changes in battery charge level over time. We divide time into *epochs*, each corresponding to a measurable change in battery charge level. In our prototype, an epoch corresponds to an average battery drain of 150 mWh. The energy drain for each epoch is chosen randomly from the range 100–200 mWh to avoid any correlation with periodic system events. Within each epoch, we measure the average amount of time spent by each device in each power state, as well as the average power consumption (the energy drain divided by the epoch length). This gives us one sample point for our model. By collecting a number of such samples, we derive the power/energy cost of each device state/action, and hence the energy consumed over any time interval for which all the device states and actions are known.

## 4 Feasibility study

Our evaluation of software power measurement was aimed at answering the following questions:

- How does device state impact battery drain?
- How accurate is software power measurement, compared to a scheme that ignores device state?
- How does the accuracy change with the number of training samples?

Our evaluation approach is to collect power samples with devices in different states and to derive the per-device state costs through linear regression. Ideally we would then validate these against the known power/energy costs of the device states and actions; however, these are not available on our hardware platform. Instead we validate our model by comparing its prediction of average epoch power with the measured value. In other words, given a series of epochs with known power consumption, we predict the average power consumption of a new epoch and compare that against the measured value.

In general, learning power consumption as a function of device state would require us to test a large number of device state combinations. In our linear model, however, the power consumption of each device is independent of the state of other devices. Thus, we can explore the state space by varying one state variable at a time and keeping the others constant.

Our evaluation platform was a Dell Latitude D600 with a 1.6 GHz Mobile Pentium processor, running Windows XP modified to record device state transition events as well as periodic measurements of battery charge level. The CPU power state was varied by modifying Windows power management to allow user-level control of the C- and P-states. This allowed us to set the P-state (CPU speed) to any one of 12 supported states ($P0 \ldots P11$). It also allowed us to force Windows to pick one particular C-state ($C1$, $C2$, or $C3$) when idle: the default scheme adaptively chooses the idle state depending on the recent idleness of the CPU. To reduce the number of experiments, our evaluation explores all four of the C-states but only two of the 12 P-states ($P0$ and $P5$, corresponding to 100% and 37% of the maximum processor speed). Of the 8 possible screen states, we explore three: blank, brightness 3, and brightness 7.

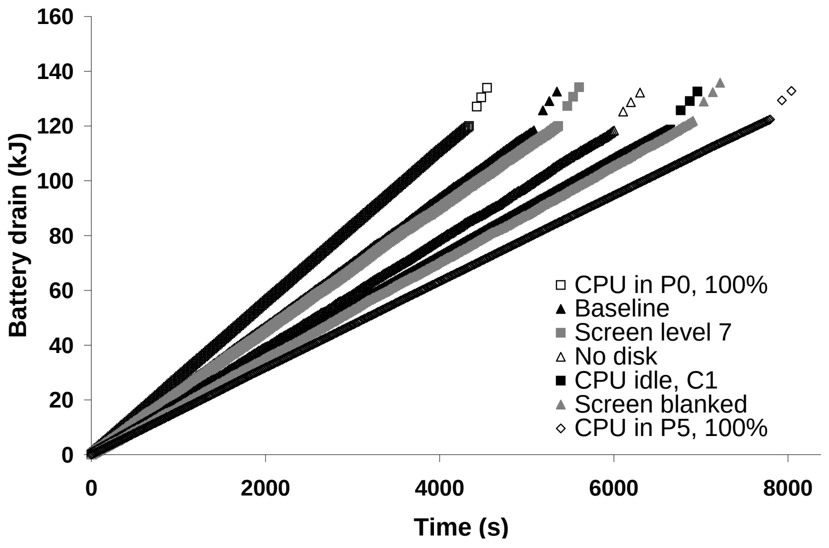

Figure 2: Differential power measurement

Unfortunately, disk state is much more difficult to control. The disk power management interface only allows us to set idle timeouts for various state transitions, and not to trigger state transitions directly. Further, the system disk on a Windows machine is accessed frequently, causing frequent transitions back to the high power state ($D0$). Thus, it is impossible to observe the disk in a low power state for any significant period of time. Given these constraints, we only consider two states for the disk:

- $D0$: the highest-power state, disk is spun up and ready to read/write
- No disk: we remove the disk and boot the machine from a network server

Also, given the difficulty of controlling the number of spin-ups and spin-downs in an epoch, we do not include these in the current model, ensuring instead that the disk always spun-up when present.

To explore the state space, we put the machine into one of 7 configurations, and sampled the power consumption for an entire battery charge (i.e. fully charged to fully empty). The baseline configuration had the CPU at speed level $P0$, the screen at level 3, and the disk in $D0$. The CPU was running a synthetic, randomly varying workload with a mean utilisation of 65%. When idle, the state transitions were determined by the default Windows strategy, which selects $C1$, $C2$, or $C3$ depending on recent idleness. All the other configurations are variations on the baseline:

- *Screen level 7*: Screen brightness 7, no CPU load

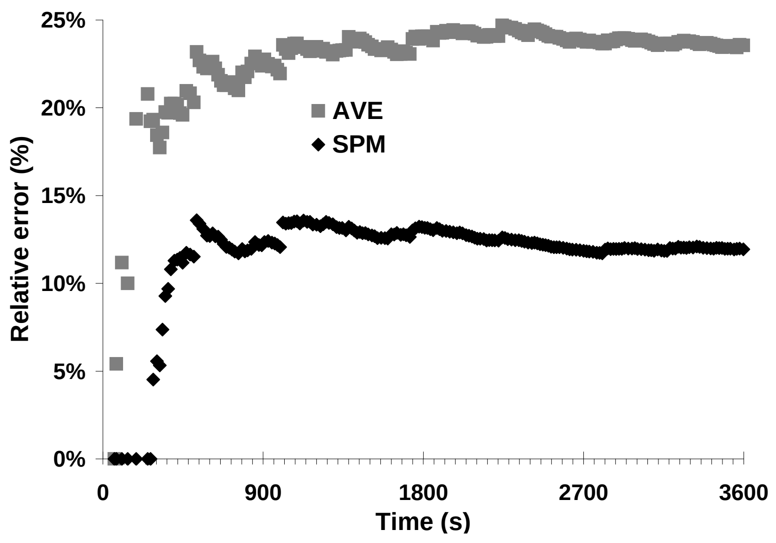

Figure 3: Accuracy of linear predictor

- *Screen blanked*: Screen blanked, CPU load
- *No disk*: Disk removed, no CPU load
- *CPU in P0, 100%*: CPU load of 100%
- *CPU in P5, 100%*: CPU load of 100%, speed 37%
- *CPU idle, $C1$*: No CPU load, states $C2$ and $C3$ disallowed

Figure 2 shows the energy drain in each of these configurations. We see that the rate of power consumption is constant for each configuration, but significantly different between configurations, indicating that power consumption is in fact highly correlated to device state.

In practice, we would not want to drain the entire battery once for each configuration, but train our predictor on a smaller number of samples, switching device states periodically to explore the state space. We measured the effect of doing this by interleaving the samples from the seven traces, progressively updating the linear model at each step, and measuring the prediction error with respect to the next sample.

Figure 3 shows the accuracy of the SPM predictor over time, compared to AVE: a predictor that simply averages past samples to predict future power consumption without regard to device state. We see that the SPM predictor requires 5–10 min of measurement to stabilise, and provides significantly better accuracy than AVE, with an RMS error that is 10–15% of the true value, compared to 20–25% for AVE.

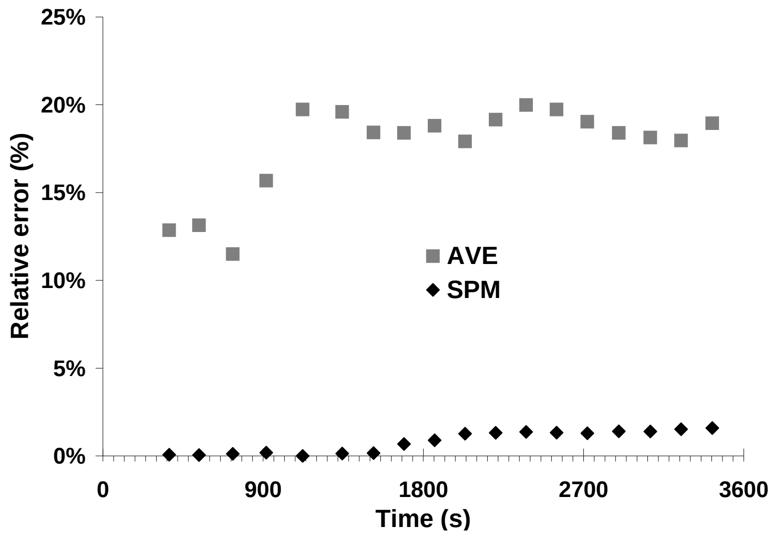

Figure 4: Accuracy of smoothed linear predictor

We surmised that much of SPM's error was due to noise in the underlying battery charge measurements. To test this hypothesis, we tested SPM against a smoothed version of the input data, by making the epochs 10 times longer: each epoch in the smoothed set consists of 10 consecutive epochs from the original analysis. Figure 4 show that this significantly reduces the error of SPM to under 2%, while AVE continues to suffer from ignoring device state. The disadvantage of smoothing is that it takes longer to acquire samples and hence longer for the predictor to converge: epoch length is a tuning parameter that trades agility for stability.

## 4.1 Limitations

The primary limitation on the SPM approach is the granularity of the underlying SmartBattery interface. The measurements available through this interface are coarse-grained both in time and in device-level scope. Further, the relationship between these measurements and the true power consumption (e.g. the period of time averaging) are vendor-specific and unknown. Some of these limitations can be alleviated by averaging power measurements over long time periods, at the cost of increasing the time to convergence.

Our current prototype is limited in the number of device states and actions that it tracks and models. Most importantly, it does not model the wireless interface, a significant consumer of energy when mobile. Internal states of the wireless interface are often hidden from the operating system, making it hard to accurately correlate these states with power consumption. We also ignore the energy consumption of memory subsystem activity, which is typically small compared to that of the entire system. Similarly, we do not include additional measurements such as Pentium event counters, which could provide more information about CPU power consumption than cycle counts [1], but would require a higher tracing overhead and a more complex model.

Our current approach is based on offline benchmarks which explore the space of device power states. The linear regression algorithm itself is cheap, fast, and easily used in an online update mode instead. However, online operation presents other challenges. We are no longer free to explore the state space arbitrarily, since this will impact the functionality and performance seen by the user. Instead we must learn from the state changes occurring naturally as a result of user behaviour and OS adaptation, which may be only a small subset of the entire state space. This disadvantage could be overcome by running the energy benchmarks only when idle and on AC power, or at boot time, or when a new device or OS is installed. The cost models could also be initialized from a database of known power parameters for each device or device type, providing approximate power estimates while the true costs are being learned.

Finally, our evaluation indicates the feasibility of the SPM approach, but does not validate the SPM predictions against real, fine-grained, per-device instantaneous power measurements such as those obtained from instrumented hardware. Such a comparison would let us calibrate the SPM approach on the lab bench before deploying it in the field.

## 5 Related Work

Section 2 we described current approaches to power measurement; here we briefly describe work in the broader area of adaptive power management.

Most work in adaptive power management focuses on dynamically setting a single resource to the lowest power state possible given some performance bound. Dynamic voltage scheduling for processors [4, 8, 12] reduces clock speed as much as possible without missing task deadlines. Similarly, adaptive disk spin-down [3] minimises disk spin-ups and time spent spun-up while bounding the impact on access latency. These approaches avoid explicit measurement or estimation of power consumption at runtime, and thus do not lend themselves to cross-resource adaptation decisions. For example, they cannot compare the energy benefit of slowing down the CPU to that of spinning down the disk.

Energy is treated as an explicit, first-class system resource in ECOsystem [13]. Each process's energy usage is computed from its usage of CPU, disk, and network by consulting a power profile. SPM could generate or refine such power profiles automatically without requiring manual benchmarking or analysis.

Previous work by the author and others [5, 10] has shown that the energy consumption of an adaptive application can be learned as a hardware-specific function of its adaptive parameters. SPM can simplify this learning task by separating out the hardware-specific portion: the mapping of resource consumption to energy consumption.

## 6 Conclusion

This paper proposes *software power measurement*, a novel technique for cheap, accurate, and fine-grained estimation of power consumption on mobile computers. Software power measurement is based on correlating device usage statistics to time-averaged power consumption, and using the resulting model to generate estimates of instantaneous, per-device power consumption. Our evaluation shows that the device power state information available in the OS is highly correlated with, and thus a good predictor of, power consumption. It also shows that a simple and cheap linear model can learn this correlation with good accuracy.

Based on our experience, we have the following recommendations for designers of laptop hardware such as "smart" batteries:

- to provide unfiltered, unsmoothed power measurements to the OS where possible
- if the samples are filtered, to provide detailed information on the filtering algorithm and the time constants involve
- to equip each major hardware component (CPU, disk, PCI cards, etc.) with its own power measurement hardware such as as a gas-gauge chip

We would also recommend that OS and device driver designers

- expose all device state transitions and actions that affect power/energy consumption
- provide interfaces for application-level control of device power state, allowing easy exploration of device power states and power management strategies.

## References

[1] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proc. 9th ACM SIGOPS European Workshop*, Sept. 2000.

[2] T. Cignetti, K. Komarov, and C. Ellis. Energy estimation tools for the Palm. In *Proc. Ninth ACM Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '00)*, Aug. 2000.

[3] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. Second Symposium on Mobile and Location-Independent Computing (MLICS '94)*, Apr. 1995.

[4] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for Linux. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002.

[5] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Dec. 1999.

[6] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, Feb. 1999.

[7] J. R. Lorch and A. J. Smith. Apple Macintosh's energy consumption. *IEEE Micro*, 18(6), Nov./Dec. 1998.

[8] J. R. Lorch and A. J. Smith. Operating system modifications for task-based speed and voltage scheduling. In *Proc. 1st International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, May 2003.

[9] Microsoft. Event Tracing for Windows (ETW). `http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp`, 2002.

[10] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applicatons (WMCSA '00)*, Dec. 2000.

[11] SBS Implementers Forum. Smart Battery Data Specification, Revision 1.1. `http://www.sbs-forum.org/`, Dec. 1998.

[12] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. First USENIX Symposium on Operating System Design and Implementation (OSDI '94)*, Nov. 1994.

[13] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proc. Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002.