Dynamic Power Management using Feedback*

Robert J. Minerick Vincent W. Freeh Peter M. Kogge†
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN 46556

Abstract

The increasing speed and complexity of the microprocessor has brought about a corresponding increase in power consumption. Coupled with relatively small gains in battery capacity over recent years, the importance of intelligent battery management has become paramount. This paper presents a mechanism that takes advantage of feedback about power consumption in order to use battery energy more effectively. This feedback mechanism allows the implementation of many different energy management policies. One such policy is presented here which allows us to direct power consumption by changing performance states in the scheduler to achieve predetermined energy goals. Furthermore, our implementation in Linux can synthesize any average power usage rate with little overhead.

1 Introduction

In many situations, performance is limited by the energy available. For example, in mobile devices, there is a finite amount of energy that must be conserved to provide a useful battery life. Similarly, it may not be possible during peak energy periods to supply the maximum power to all computers in a dense rack. Furthermore, their power consumption may be limited by the ability to dissipate heat.

Many new microprocessor architectures provide several voltage and clock settings that reduce the power required and the performance provided. In addition to lowering the power consumption, reducing processor voltage can reduce the energy expended because it can decrease the amount of energy used per work unit (i.e., instruction).

This paper describes a dynamic power management mechanism utilizing real-time feedback. Additionally, this paper describes a power management implementation that enables a finite amount of energy to last a particular amount of time. As an example, consider the user of a laptop who wants the battery to last for the duration of a transcontinental flight. In this scenario, the overarching goal is to ensure that the energy in the battery will last until some future time. Obviously, when there is ample energy, the goal is easily met. However, when there is insufficient energy to service all pending work, some work cannot be performed.

This paper describes a unique implementation in Linux. Our power management mechanism guarantees that the energy in the battery meets or exceeds the target energy at a specified time in the future. It continuously adjusts the power setting on the microprocessor to achieve an average power usage that meets the target energy goal. Rather than trying to predict future performance requirements, our mechanism uses real-time feedback about the current energy to adapt the proper power setting to reach the target. Consequently, it achieves the conservation goal in highly dynamic situations.

For this paper, we focus on an implementation for the AMD Mobile Athlon, which can scale both voltage and frequency. We show results in which battery life is extended by reducing power consumption. Power consumption is reduced by selecting a lower performance setting on the CPU, which generally reduces the power demand of other components in the system. This paper also shows that system power consumption can be regulated with CPU power set-

*This work was supported by DARPA FC 306020020525 and National Science Foundation CAREER Grant CCR-9876073.

†Email: {rmineric,vin,kogge}@cse.nd.edu

tings. Our mechanism exhausts the energy within 1% of the target time with little overhead.

The following section gives an overview of related work. Section 3 presents the model for our mechanism and introduces one possible policy. Section 4 shows how the mechanism, along with the policy mentioned above, is implemented in the Linux kernel. Section 5 presents our results. Finally, Section 6 discusses the conclusions we reached based upon this research.

2 Related Work

The Odyssey platform uses instantaneous power consumption information to direct program activity towards reaching an energy consumption goal [20]. It uses both an external device and a smart battery to monitor energy usage and report this energy usage to the kernel [8, 7] Odyssey assumes that applications can adapt themselves to the environment's energy needs, which requires either application specific knowledge or some type of proxy between the application and the OS. While this assumption enables Odyssey to control energy consumption, it greatly diminishes its applicability. On the other hand, our mechanism is totally transparent to applications, which makes it a more general solution.

There has been a wide variety of research geared toward estimating power in software through event counters [1, 14], system level events [2], or at the instruction level [30, 28]. Our implementation uses a laptop equipped with a battery containing a status monitor, which we access via the ACPI [3, 9] interface.

Allowing the operating system and related software greater control over power management is becoming more popular—ACPI is a direct result of this. Several papers have looked at the advantages of doing this at the software level [16, 5, 31, 34]. Some of this research is directed at particular components of the machine, such as disks or network devices [4, 18, 33, 17].

There has also been much research into modifying the OS scheduler to achieve power savings. While there are examples of scheduling optimizations that do not take advantage of voltage or frequency scaling [15, 19], developments in processor technology have directed most research towards scaling. Weiser et al [32] described and evaluated three scheduling algorithms to reduce power through the use of two voltages and clock speeds and offline analysis of system traces. The goals being to both reduce power consumption and to minimize delay by accurately predicting future load. Others have drawn on their work to provide improved algorithms [10, 12], but the goals remain the same.

This work developed into dynamic voltage scaling (DVS) [6, 11, 22, 24, 13], which has come to mean the simultaneous changing of clock speed and voltage to reduce power consumption. DVS takes advantage of the fact that peak processing power is not always necessary to adequately service the average system load.

DVS systems attempt to balance performance and consumption. Typically, they optimize the energy$\times$delay product. This creates a system that more efficiently uses energy, but is still powerful and responsive. In order to do this, DVS must accurately *predict* upcoming load. A poor prediction either consumes too much energy or causes too much delay. In contrast, our system has the singular goal of meeting a target energy. Consequently, it need not predict load. Instead, it must keep the remaining energy above some threshold necessary to meet the target energy.

Additional research ties the scheduler directly to DVS [21, 24, 23, 25]. Most of this work deals exclusively with meeting real-time scheduling deadlines while conserving energy. Our mechanism is complementary to such real-time schedulers in that it can exploit dynamic slack from the real-time tasks. Realtime tasks are always completed by their deadline, and the cycles left over are used appropriately.

3 Power Management

There are several power management scenarios. A system with no power restrictions operates at the maximum performance, without regard to power. On the other hand, a system designed solely to conserve energy operates only at the voltage and frequency setting with the lowest energy per instruction. This is essentially how SpeedStep is used on the Intel Mobile Pentium line of processors. Two per-

formance settings are available, a high power, high performance setting, and a low power, low performance setting. When the AC adapter is plugged in, the machine operates at maximum performance and maximum power. Without AC power, the processor switches to the lower performance setting to conserve power. The above extremes are trivial and static in nature. An energy conservation policy must balance the conflicting demands of performance and conservation dynamically. Because there are many different workloads and goals, there are also many possible balancing schemes.

This section models the problem of goal-oriented energy consumption, or energy conservation (EC). Our mechanism maintains a target average power for the system by selecting the appropriate operating point. The precise definition of operating point varies depending on implementation. For processors that feature voltage and frequency scaling, the operating point is a specific voltage/frequency combination. The energy conservation goal ensures that the finite energy in a battery will last a given amount of time. Formally stated the problem is: given the current energy (E at time 0) and desired energy (E_f) at a time from now (T), ensure that the energy in the battery is at least E_f at time T. Supposing a system has infinitely many operating points, one would merely select operating point P, where $P = (E - E_f)/T$. Of course, there are a rather small number of such points in real microprocessors.

Our power management mechanism selects the proper operating point to achieve an average power usage, $\hat{P}$, such that over time the goal is met: $\hat{P}T \leq E - E_f$. The closer that $\hat{P}T$ approaches $E - E_f$, the better performance will be, and, consequently, the closer the battery will be to depletion. If $\hat{P}$ is lower than the minimum operating point the goal cannot be met. Conversely, if $\hat{P}$ is greater than the maximum operating point, then the goal is trivially met (because it would be impossible even at the maximum operating point to expend enough energy to use the entire battery). Ignoring the two trivial extremes just stated, the desired average power always lies between two operating points. If operating points are labeled P_i from lowest to highest performance (and consumption), then the target point will lie between adjacent points, i.e., $P_i \leq \hat{P} \leq P_{i+1}$. This intermediate operating point, $\hat{P}$ can be synthesized over an interval by executing P_i and P_{i+1} for an appropriate proportion of time.

Many modern microprocessors have several operating points, which are typically voltage/frequency pairs. This directly effects the power consumption of the microprocessor, which in many systems consumes a significant percentage of the total system energy. Moreover, because the CPU is the primary producer of work, there tends to be a correlation between CPU performance and overall system power. In other words, when the CPU frequency is reduced, the load on other components tends to also be reduced. Obviously, there exist pathological cases in which the CPU has little or no influence on the power consumption of other components. Consider the case of the network device where the CPU has no control over the rate of incoming packets on the network. In general, though, CPU performance is important when considering system power consumption.

In a dynamic system, demand is varied and unpredictable. In particular, it can be zero—there are no ready processes to execute. At such a time, power consumption can be reduced, often significantly, by entering an *idle* or *doze* mode. (In X86 processors this mode is entered with the `HALT` instruction, which turns off functional units but still listens to signals.) Consequently, the desired operating point in a dynamic setting is:

$$P(t) = \begin{cases} P_{idle}, & \text{if no work} \\ (E(t) - E_f)/(T - t), & \text{otherwise} \end{cases}$$

When there is no work to do for a period of time, "excess" energy is accumulated. When demand on the processor returns, the operating point, $P(t)$, will be larger than the average $\hat{P}$. Therefore, idle periods afford the system a higher operating point later when it is needed.

This model can be extended to include recharging of the battery. Suppose that E_r energy is added during the period T, i.e., $E_r = P_r T$ for a constant recharge rate. Then the desired operating point is $P(t) = (E(t) - E_f + E_r)/(T - t)$. If the recharge rate is not constant, but one can estimate the recharge rate, then the operating point becomes $P(t) = (E(t) - E_f)/(T - t) + P_r(t)$. The estimate $P_r(t)$ must hold for the entire interval that this op-

erating point is used. Even if the confidence in this estimate is low, it is not an issue until very near the goal when there is insufficient time to adjust.

Additionally, the model can be used in conjunction with a real-time scheduler, where EC schedules non-real-time asks using excess power (beyond that needed for the real-time tasks). Supposing the worst-case energy needed for the real-time tasks is $E_{rt} = P_{rt}T$. Then the operating point is selected similar to that with recharge, except that E_{rt} is a debit not a credit. So the desired operating point is $P(t) = (E(t) - E_f - E_{rt})(T - t)$. (Of course, the operating point must never be set lower than that required to meet real-time performance demands.) A static real-time schedule must be conservative; therefore, in all but the worst-case situations, the energy at time T will exceed E_f (the goal). When a real-time task uses less energy (than the worst case), it creates "slack" energy which can be allocated to non-real-time tasks. A dynamic EC mechanism can use all available slack energy for non-real-time tasks.

This energy conservation policy discussed in this section is one of many that require a dynamic power management mechanism. For instance, instead of guaranteeing a certain battery lifetime, it can be used to control heat generated. In this case, the policy would monitor temperature and adjust the power setting appropriately.

Frequency	Voltage	Higher	Lower
1400	1.45	-	72
1350	1.45	72	529
1300	1.40	545	69
1250	1.40	66	489
1200	1.35	505	64
1150	1.35	63	449
1100	1.30	464	59
1050	1.30	58	408
1000	1.25	424	55
950	1.25	53	54
900	1.25	51	51
850	1.25	50	330
800	1.20	346	46
750	1.20	43	43
700	1.20	41	41
650	1.20	39	39
600	1.20	36	37
550	1.20	35	35
500	1.20	32	32
300	1.20	25	-

Table 1: Voltage/Frequency combinations and transition times to next higher and lower states (in microseconds)

4 Implementation

This section describes the implementation of the feedback mechanism and a goal-oriented policy we call *energy conservation* (EC). This implementation is done in Linux for the AMD Mobile Athlon, which utilizes voltage and frequency scaling. However, it does not require voltage or frequency scaling. An earlier implementation for the PowerPC 7400 places the processor in doze mode for a certain fraction of cycles to reduce power consumption.

The feedback mechanism is made up of four software components. First, a downloadable kernel module provides a policy (which selects the desired power setting) and several utility functions. The utility functions enable runtime specification of the policy, as well as policy-specific parameters (such as target lifetime). The module creates a file in the *proc* filesystem that provides all status information regarding the current state of the processor, battery, and policy. The second component is a kernel-level daemon that periodically inserts energy consumption and battery status data into the kernel. It can use an internal monitor (such as that implemented on a smart battery, accessed via ACPI), or an external monitor via the network (such as a DAQ running on a second machine). This daemon has direct access to kernel memory, which eliminates cross-boundary copies, reducing overhead. Third, another daemon logs data for postmortem analysis, and similarly supports an external monitor for logging via the network for instantaneous analysis. Finally, the feedback mechanism requires a modification to the kernel scheduler. In the scheduler, we provide a hook that calls a function in the downloaded kernel module when a scheduling decision is made.

The test machine has a 1.4 GHz AMD Mobile Athlon that supports twenty frequency and six volt-

age settings. Table 1 shows valid combinations that were derived empirically. For each frequency setting, there are various valid voltage settings. The combinations were chosen both to ensure system stability and to give a wide range of operating points. The table also shows the time to transition to the next higher and lower operating point, respectively, in microseconds. These values were found by using the Athlon's time stamp register, which continues to count during the voltage or frequency transition. As expected, transition times between states with constant voltage are considerately smaller than those with differing voltages. This can be an important consideration when implementing other policies, especially those which must react quickly to changing energy consumption. For the EC policy this is not particularly important because we change states only when new data is sampled, which occurs every few seconds—this is discussed further in the following section. Therefore, even when a state change takes 500 microseconds, the overall effect on performance is negligible.

The operation of the feedback mechanism is conceptually simple. Data is gathered from the energy monitor (in this case, ACPI) and written to a kernel data structure. When new data is inserted into the structure, a flag is set to make the module aware of the fact that new data has arrived. If new data has arrived, the scheduler makes a call to the module, which determines the proper setting via the current policy.

The energy conservation (EC) module, created for this paper, executes a proportional integral controller [27, 29]. The goal of the proportional integral controller is to minimize the error term

$$e(t) = e(t-1) + (\hat{P} - P_m(t)), \quad (1)$$

where $\hat{P}$ is the average power consumption that must be maintained (as described above) and $P_m(t)$ is the average power measured by the kernel daemon at time t. On the Mobile Athlon implementation, ACPI allows us to measure average power over a period of 1 minute. In another implementation on the PowerPC, we use a data acquisition board running on a second machine to measure average power consumption at a smaller granularity. The smaller granularity (for example, power averaged over 1 second) allows

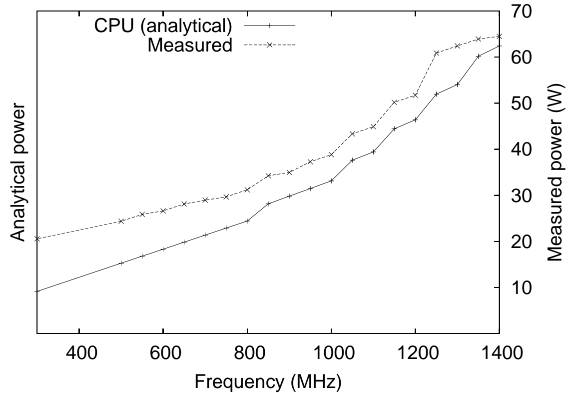

Figure 1: Power vs. performance curve for AMD Mobile Athlon @ 1400 MHz.

the EC policy to adjust more quickly, which means it can more precisely track the target power consumption. However, we have found that given an appropriate amount of time, ACPI provides an adequate granularity to allow the EC policy to meet it's goals successfully.

The user informs the EC module of the target battery energy, E_f, and lifetime, T, via either the *proc* filesystem or an *ioctl* call. The remaining battery energy $E(t)$ is read at a specific sampling interval by the daemon and copied into kernel space. Each time the EC kernel module is invoked by the scheduler, it reads this structure to determine if new data is available. It then calculates the error $e(t)$ from (1). $\hat{P} = (E(0) - E_f)/T$ is calculated only once, when the user supplies the desired battery lifetime. $\hat{P}$ is also referred to as the bias value, and is used by EC to maintain a steady state when $e(t)$ is 0.

Finally, the bias value and the error are used (along with K, a dimensionless gain constant, which is tunable), to attain our new set point, $P_s(t)$:

$$P_s(t) = \hat{P} + Ke(t)$$

$P_s(t)$, which is continuous, must be converted to a discrete setting. The value of the gain constant is interesting in itself, as it determines how quickly the EC module responds to changes in power consumption. This is discussed in the following section.

When the EC module calculates a new value for $P_s(t)$, it decides whether to transition to a new state based on a table of experimentally calculated

state/energy pairs. This table is shown graphically in Figure 1. It shows the power consumed while running the processor using a program called burnK7, part of the cpuburn [26] suite of programs. This program is meant to stress the CPU to find problems in a machines thermal design (it's ability to move heat away from the processor). Therefore, it is a good approximation of the maximum power the CPU will consume at a given frequency and voltage. Not shown on the graph is the voltage used at each frequency setting, these can be found by referring back to Table 1. It is interesting to observe that even though we are measuring the power consumption of the entire laptop, overall power still follows the analytical model, $E \alpha f V^2$, for the microprocessor alone.

If $P_s(t)$ is within a certain threshold of $\hat{P}$, then the CPU maintains its current setting. The threshold determines how aggressive the EC policy is. For example, consider the case where the desired average power is 33 watts, but the measured power $P_m(t)$ is 37 watts. Assuming a K value of 1 (as an example only, realistically this value is much too large), $P_s(t)$ is calculated to be 29 watts (assuming there was no prior error). Using a table of calculated state/energy pairs, the scheduler will find that 29 watts lies between two operating points. In the operating point above, frequency is 900 MHz, voltage is 1.25 volts with a corresponding approximate power consumption of 32 watts. In the operating point below, frequency is 850 MHz, voltage is 1.2 volts with a corresponding approximate power consumption of 28 watts. EC will transition to the higher operating point depending on the difference in power consumption between the two operating points multiplied by the threshold. If the threshold is set to be within, for example, 75% of the difference between the two operating points (which is 4 watts), then EC will transition if $P_s(t)$ is within $.75 * 4 = 3$ watts of the higher of the two operating points (which is 32 watts). In this case, it would transition to the higher operating point. If the threshold value is held lower, for example at 25%, then the policy would be considerably less aggressive, and in this case it would transition to the lower operating point.

As can be seen from the above example, the calculated state/energy pairs are used only as a guideline of the relative power consumption between states.

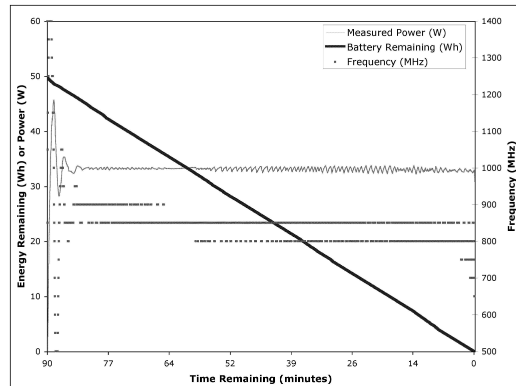

Figure 2: Trace of energy for fully loaded processor.

The most important factors in determining the next state are the accumulation of error in the controller $e(t)$ and the gain constant K As we show in the following section, $P_s(t)$ adjusts itself via these two terms to fulfill the requirements of the EC policy.

5 Results

This section shows results obtained with our implementation of EC in Linux. Our experimental machine is a Compaq Presario 700 laptop with a 1.4 GHz AMD Mobile Athlon processor, running the Linux 2.4.18 kernel. The battery on this machine has a built-in battery monitor, which reports back to the OS critical battery information such as the remaining battery energy and discharge rate. We access this information via the ACPI interface. The monitor can be configured to read the battery status information at varying rates. The minimum time interval between battery status updates is slightly less than 3 seconds. Fortunately, we found this to be more than adequate to implement the EC policy. For all tests in this paper, a sampling frequency of 3 seconds is used. For the sake of consistency, we choose a 90 minute goal for all related tests, although this number holds no particular significance.

Figure 2 shows the trace data for a 90 minute battery goal, where the processor is under a full, constant load. This trace shows that average power is tracked very tightly and the resulting curve is nearly a straight line. This particular case is handled easily by the proportional integral controller. Due to the

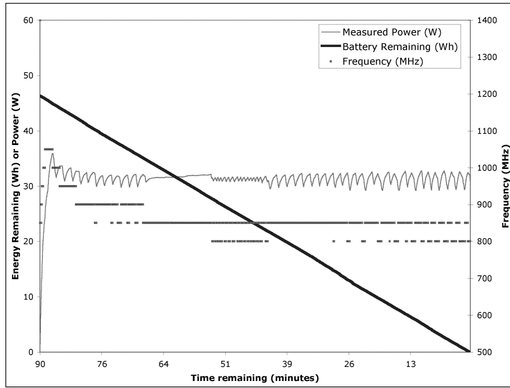

Figure 3: Trace of energy for a variable load, K = .02.

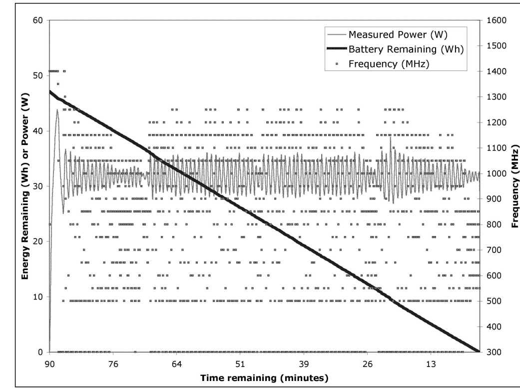

Figure 5: Trace of energy for a variable load, K = 2.

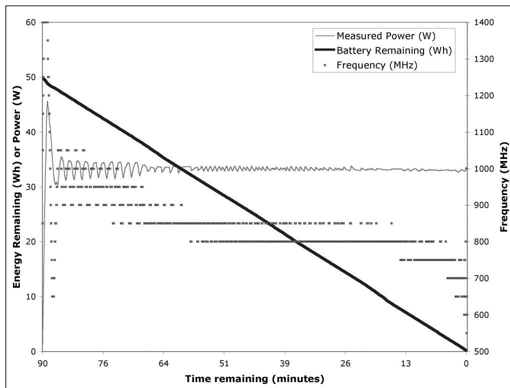

Figure 4: Trace of energy for a variable load, K = .2.

constant load, once the proper power consumption is determined, the controller needs at worst to choose between two operating points.

Figure 4 shows how EC responds when the processor is experiencing a variable load, again for a 90 minute battery goal. This test consists of a script which both loaded a random web page and encodes an mp3 periodically, while compiling a very large project in the background. This resulted in highly variable load, ranging from periods of full load to periods of mostly I/O. Again, the trace shows that average power is tracked tightly. It is interesting to note here that because of the highly variable load, the EC policy takes longer to steady the average power consumption.

The speed and magnitude of the adjustment made by the EC policy is determined both by the previous error and the damping factor *K*. Choosing an appropriate K that is satisfactory for all loads is critical. We found that a K value of .2 generally yields satisfactory results. It allows the EC policy to respond quickly to varying loads, while also maintaining a consistent average power consumption.

Figures 3, 4, and 5 show a variable load with K values of .02, .2, and 2, respectively. Notice that the average power consumption maintained with a K value of .02 or 2 is not as consistent as that maintained using a K value of .2. For the smaller value of .02, EC is not able to respond as quickly to a varying load. Instead, it often chooses to maintain the current operating point. This can be seen in the last part of figure 3 as increases in power consumption as EC maintains the current operating point, followed by sharp decreases when EC finally responds. While the operating point may yield an average power consumption close to that desired, this slow response can be dangerous. For example, the operating point may be too high near the end of the battery. Since EC with a low damping factor responds slowly, it may miss the battery goal. Using a large K value of 2 as in figure 5 causes other problems. The performance of both interactive and non-interactive tasks are degraded due to frequent modification of the operating point. In interactive tasks, the user experiences frequent pauses in feedback. For non-interactive tasks, because each operating point change takes some amount of time, the time for each task to run to completion will be increased. While choosing a large K is not necessarily problematic in

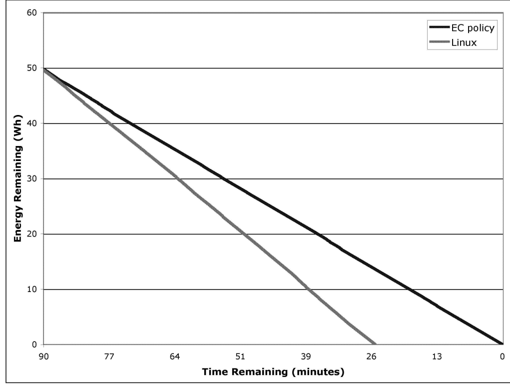

Figure 6: High and low priority processes.

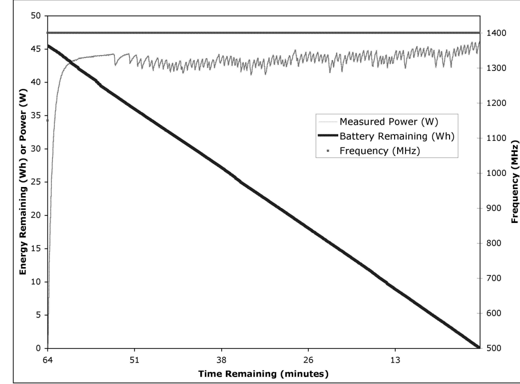

Figure 7: EC over a 64 minute interval.

Policy	Work performed		
	High	Low	Total
Linux	.338	.338	.676
EC	.500	.500	1.00

Table 2: Normalized work performed by high and low priority processes.

terms of hitting the target, it is generally undesirable when used on a real system.

Also of interest is the frequency changes in the last few minutes of Figure 2 and the last 15 minutes of Figure 4. Although barely noticeable in the figures, there occurred a significant drop in battery level coinciding with both events. Since the smart battery gives us only the energy consumption of the entire machine, it is not possible to deduce which component(s) led to the sharp drop in battery level. However, this is simply a limitation of our measurement device, as the EC policy responded appropriately by reducing the operating point of the processor.

Figure 6 shows energy traces for a system with periodic high and low priority processes. Here, a high-priority task with a period of 2 seconds and a low-priority task with a period of 1 second are executed. For EC, an energy goal of 90 minutes is used. Both processes execute the same amount of work per second.

Table 2 lists the normalized work (in terms of application instructions executed) performed by each policy. Since EC maintains the battery for a full 90 minutes, each task is executed more times than would be using the standard Linux scheduler.

Figure 7 shows EC running with a goal time identical to the length of time standard Linux depletes the battery (see Figure 6, approximately 64 minutes. EC ran the processor at maximum speed and power over the entire interval, doing an identical amount of work. This is to be expected, since the type of work is periodic.

As mentioned earlier, 90 minutes is not meant to represent the maximum lifetime that EC can achieve on this particular machine. In our experiments, we have found that under many conditions, EC will outlast unmodified Linux by up to 2.3 times. Of course, the relative power of the processor to other components on the machine plays a major part in determining the power savings one can achieve by using EC. However, on machines where the processor is the top consumer of battery power, EC can a provide significant boost to battery lifetime.

6 Conclusions

This paper presents the foundation for a feedback directed power management system implemented in Linux. It also describes the implementation of one policy, which we call energy conservation. Several tests are presented that show the implementation is able to guarantee that the energy in the battery meets or exceeds the target energy at a specified time in the future. Furthermore, we show that EC responds correctly under a variety of different loads. We show that a practical power management system can be

implemented without the need to predict the load of the system. Moreover, our mechanism is able to conserve energy for use by processes which execute periodically. This paper discusses how the model can be implemented as a complement to a real-time scheduler. Finally, we show that on a laptop where the CPU consumes a significant percentage of system power, battery consumption goals can be met simply by regulating CPU frequency and voltage.

References

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

[2] L. Benini, A. Bogliolo, and G. De Micheli. Monitoring system activity of OS-directed dynamic power management. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '98*, 1998.

[3] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced configuration and power interface specification, revision 2.0. July 2000.

[4] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proceedings of the 1994 Winter USENIX Conference*, pages 293–306, 1994.

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

[6] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM '01*, July 2001.

[7] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, 1999.

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

[9] Linux Kernel Foundry. *ACPI*. July 2002.

[10] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Mobile Computing and Networking*, pages 13–25, 1995.

[11] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '01*, August 2001.

[12] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of 4th Symposium on Operating System Design and Implementation*, October 2000.

[13] C. Im, H. Kim, and S. Ha. Dynamic voltage scheduling technique for low-power multimedia applications using buffers. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '01*, August 2001.

[14] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '01*, August 2001.

[15] J. Lorch and A. Smith. Scheduling techniques for reducing processor energy use in macos, October 1997.

[16] J. Lorch and A. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications Magazine*, 5(3):60–73, June 1998.

[17] Y. Lu and L. Benini. Power-aware operating systems for interactive systems. *IEEE Transactions on Very Large Scale Integration Systems*, 10(2), April 2002.

[18] Y. Lu, T. Simunic, and G. De Micheli. Software controlled power management. *ACM CODES 1999*, 1999.

[19] R. Minerick, V. Freeh, and P. Kogge. Feedback-directed, adaptive energy conservation, February 2002.

[20] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems and Principles*, pages 276–287, October 1997.

[21] T. Pering and R. Brodersen. Energy efficient voltage scheduling for real-time operating systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS '98*, June 1998.

[22] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '98*, pages 76–81, August 1998.

[23] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[24] J. Pouwelse, K. LangenDoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '01*, August 2001.

[25] Gang Quan and Xiaobo Hu. Energy efficient fixed priority scheduling for real-time systems on variable voltage processors. In *ACM/IEEE Design Automation Conference*, pages 828–833, June 2001.

[26] R. Redelmeier. *cpuburn*. http://users.ev1.net/ redelm/, June 2001.

[27] D. E. Sebord, T. F. Edgar, and D. A. Mellichamp. *Process Dynamics and Control*. John Wiley & Sons, 1989.

[28] P. Stanley-Marbell and M. S. Hsiao. Fast, flexible, cycle-accurate energy estimation. *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '01*, August 2001.

[29] G. Stephanopoulos. *Chemical Process Control*. PTR Prentice Hall, 1984.

[30] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[31] A. Vahdat, A. Lebeck, and C. Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. *SIGOPS European Workshop*, 2000.

[32] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 13–25, November 1994.

[33] Giovanni De Micheli Yung-Hsiang Lu, Luca Benini. Operating-system directed power reduction. In *International Symposium on Low Power Electronics and Design*, pages 37–42. Stanford University, July 2000.

[34] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource.