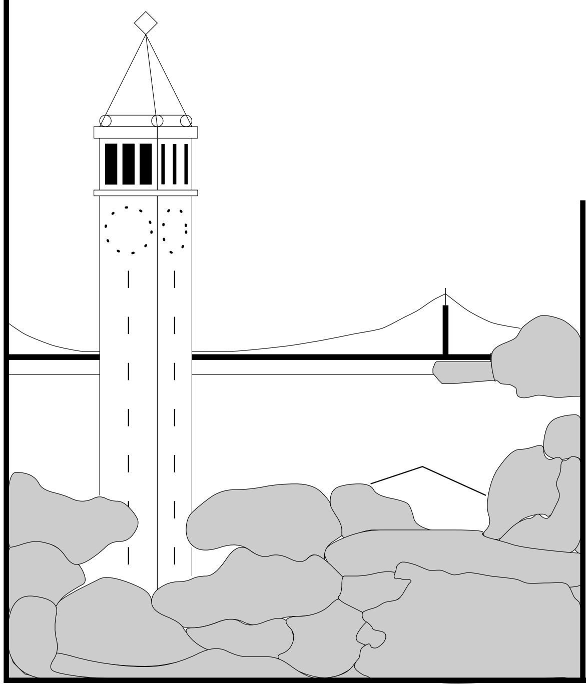
Reducing Processor Power Consumption by Improving Processor Time Management in a Single-User Operating System

Jacob R. Lorch and Alan Jay Smith



Report No. UCB/CSD-96-914

September 1996

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Reducing Processor Power Consumption by Improving Processor Time Management in a Single-User Operating System

Jacob R. Lorch and Alan Jay Smith*
Computer Science Division, EECS Department, UC Berkeley
Berkeley, CA 94720-1776

Abstract

The CPU is one of the major power consumers in a portable computer, and considerable power can be saved by turning off the CPU when it is not doing useful work. In Apple's MacOS, however, idle time is often converted to busy waiting, and generally it is very hard to tell when no useful computation is occurring. In this paper, we suggest several heuristic techniques for identifying this condition, and for temporarily putting the CPU in a low-power state. These techniques include turning off the processor when all processes are blocked, turning off the processor when processes appear to be busy waiting, and extending real time process sleep periods. We use trace-driven simulation, using processor run interval traces, to evaluate the potential energy savings and performance impact. We find that these techniques save considerable amounts of processor energy (as much as 66%), while having very little performance impact (less than 2% increase in run time). Implementing the proposed strategies should increase battery lifetime by approximately 20% relative to Apple's current CPU power management strategy, since the CPU and associated logic are responsible for about 32% of power use; similar techniques should be applicable to operating systems with similar behavior.

1 Introduction

Power consumption is becoming an increasingly important feature of personal computers. High power consumption in desktop computers is undesirable as it leads to fan noise, heat, and expense. High power consumption in portable computers is even more undesirable as users of such machines want them to last as long as possible on a single battery charge. For these reasons, much work has been done in reducing the power consumption of computers.

*This material is based upon work supported by a National Science Foundation Graduate Research Fellowship, by Apple Computer, and also in part by the National Science Foundation under grants MIP-9116578 and CCR-9117028, by the State of California under the MICRO program, and by Intel Corporation and Sun Microsystems. Although the work presented here has benefited from support from and cooperation by Apple, the relationship between this work and Apple's product development plans has yet to be determined.

In [7] and [8], we analyzed the power consumption of various Macintosh PowerBook computers, in typical use by a number of engineering users. We found that, depending on the machine and user, up to 18–34% of total power was attributable to components whose power consumption could be reduced by power management of the processor, i.e. the CPU and logic that could be turned off when the CPU was inactive. This high percentage, combined with our intuition that software power management could be significantly improved for the processor, led us to conclude that the most important target for further research in software power management was the processor.

Many modern microprocessors have low-power states, in which they consume little or no power. To take advantage of such low-power states, the operating system needs to direct the processor to turn off (or down) when it is predicted that the consequent savings in power will be worth the time and energy overhead of turning off and restarting. In this way, the goal of processor power management strategies is similar to that of hard disks [3, 5]. Some strategies for making these predictions are described by Srivastava et al. [13] Unlike disks, however, the delay and energy cost for a modern microprocessor to enter and return from a low-power mode are typically low. For instance, the AT&T Hobbit and certain versions of the MC68030 and MC68040 use static logic so that most of their state can be retained when the clock is shut down [13]. Also, the PowerPC 603 can exit the low-power Doze mode in about ten system clocks [4].

Because of the short delay and low energy cost for entering and leaving a low-power state, the optimal CPU power management strategy is trivial: turn off the CPU whenever there is no useful work to do. An opportunity for such a strategy is described by Srivastava et al. [13], who point out that the process scheduling of modern window-based operating systems is event-driven, i.e. that the responsibility of processes in such systems is to process events such as mouse clicks when they occur and then to block until another such event is ready. In this type of environment, the most appropriate strategy is to shut off the processor when all processes are blocked, and to turn the processor back on when an external event occurs. An essentially equivalent version of this strategy, namely to establish a virtual lowest-priority process whose job is to turn off the processor when it runs, is recommended by Suessmith and Paap [14] for the PowerPC 603, and by Suzuki and Uno [15] in a 1993 patent. Such a virtual lowest-priority process has in the past been called the "idle loop," and in mainframes typically lighted a bulb on the console.

1.1 Why it isn't trivial

We refer to the strategy of turning off the processor when no process is available to run the *basic strategy*. Unfortunately, in Apple's MacOS, processes can run or be scheduled to run even when they have no useful work to do. This feature is partially by design, since in a single-user system there is less need for the operating system to act as an arbiter of resource use [10]. Partially, it is because the OS was not written with power management in mind. Partially, it is because the MacOS, like other personal computer operating systems (e.g. those from Microsoft), is based on code originally developed for 8- and 16-bit non-portable machines, for which development time and code compactness were far more important goals than clean design or faithfulness to OS design principles as described in textbooks.

There are two main problems with the current management of processor time, one having to do with the system and one having to do with applications. The first problem is that the operating system will sometimes schedule a process even though it has no work to do. We were first made aware of this phenomenon when we studied traces of MacOS process scheduling calls, and found that often a process would be scheduled to run before the conditions the process had established as necessary for it to be ready were fulfilled. It seems that often, when there are no ready processes, the OS picks one to run anyway, usually the process associated with the active window. The second problem is that programmers writing applications generally assume that when their application is running in the foreground, it is justified in taking as much processing time as it wants. First, a process will often request processor time even when it has nothing to do. We discovered this problem in MacOS when we discovered periods of as long as ten minutes during which a process never did anything, yet never blocked; we describe later what we mean by "never did anything." Second, when a process decides to block, it often requests a shorter sleep period than necessary. Solutions to both these problems seem to be necessary to obtain the most savings from the basic strategy.

For this reason, we have developed additional techniques for process management. Our technique for dealing with the first problem is to simply make the operating system never schedule a process when it has requested to be blocked; we call this the *simple scheduling technique*. Dealing with the second problem is more difficult, since the determination of when a process is actually doing something useful is difficult. One technique we suggest is to use a heuristic to decide when a process is making unnecessary requests for processor time and to forcibly block any such process. Another technique we suggest is that all sleep times requested by processes be multiplied by a constant factor, chosen by the user or operating system, to ensure that a reasonable trade-off between energy savings and performance is obtained. We call these latter two techniques the *greediness technique* and *sleep extension technique*, respectively. We will show how using these techniques can improve the effectiveness of the basic strategy, allowing it to far surpass the effectiveness of the current MacOS inactivity-timer based strategy. Each of these is described in more detail below.

In this paper, we evaluate these different strategies, over a variety of parameter values, using trace-driven simulation. These simulations enable us to compare these algorithms to the current MacOS strategy, and to optimize their parameters. A comparison between two strategies is based on two consequences of each strategy: how much processor power it saves and how much it decreases observed performance.

The paper is structured as follows. In Section 2, we describe in detail the strategies we will be comparing, including the current strategy used by MacOS, the basic strategy, and our suggested process management techniques for improving the basic strategy. In Section 3, we describe the methodology we used to evaluate these strategies: the evaluation criteria, the tools we used for the trace-driven simulation, and the nature of the traces we collected. In Section 4, we present the results of our simulations. In Section 5, we discuss the meaning and consequences of these results, and point the way to future work.

2 Strategies

2.1 Current strategy

The currently used processor power management strategy in MacOS is based on an *inactivity timer*. The operating system will initiate processor power reduction whenever no activity has occurred in the last two seconds and no I/O activity has occurred in the last 15 seconds. Power reduction is halted whenever activity is once again detected. *Activity* is defined here and in later contexts as any user input, any I/O device read or write, any change in the appearance of the cursor, or any time spent with the cursor as a watch. The reason for the classification of these latter two as activity is that MacOS human interface guidelines specify that a process that is actively computing must indicate this to the user by having the cursor appear as a watch or by frequently changing the appearance of the cursor, e.g. by making a "color wheel" spin.

2.2 The basic strategy

The basic strategy is to turn off the processor whenever all processes are blocked. Unfortunately, under MacOS, this is not often the case, since MacOS frequently schedules some process whether or not the event for which the process was waiting has actually occurred.

One might wonder why MacOS uses this inactivity timer based strategy instead of the basic strategy. One reason is that all but the most recent Macintosh computers have high overhead associated with turning off and on the processor [6], making the basic strategy less applicable. In older processors, for example, the contents of on-chip caches were lost when the processor was powered down. Another reason is that, as we have described before and will see later, the effectiveness of the basic strategy is not very different from that of the inactivity timer based strategy, given the current MacOS method of process time management.

2.3 The simple scheduling technique

The simple scheduling technique is to not schedule a process until the condition under which it has indicated it will be ready to run has been met. In MacOS, this condition is always explicitly indicated by the process, and is always of the form, "any of the event types $e_1, e_2, \ldots$ has occurred, or a period of time t has passed since the process last yielded control of the processor." The period of time for which the process is willing to wait in the absence of events before being scheduled is referred to as the *sleep period*.

Note that, in some other operating systems, such as UNIX or Microsoft Windows, the simple scheduling technique is not needed, since it is the default behavior of the operating system.

2.4 The sleep extension technique

Using only the simple scheduling technique described above means that a process is given control of the processor whenever it wants it (unless the CPU is otherwise busy). For example, if it asks to be unblocked every 1 second, it is unblocked every 1 second, even if all it wants to do is blink the cursor, a common occurrence. Since MacOS is not a real time system, a real time sleep period does not actually have to be honored. In fact, in the current MacOS power management strategy, with power management enabled, the cursor may blink much more slowly than it would without power management. If this kind of behavior is acceptable, it is possible to increase the effectiveness of the simple scheduling technique by using what we call the *sleep extension technique*. This technique specifies a *sleep multiplier*, a number greater than one by which all sleep periods are multiplied, thus eliminating some fraction of the process run intervals. We envision that the sleep multiplier can be set, either by the user or by the operating system, so as to maximize energy savings, given a certain level of performance desired. We note that sleep extension may negatively impact performance, or even functionality, since not all delays will be as inconsequential as a cursor which blinks less frequently.

2.5 The greediness technique

The *greediness technique* is, in overview, to identify and block processes that are not doing useful work. First, we will describe the technique in general terms, and then we will indicate the details of its implementation for the MacOS.

The technique is based on the following model of the appropriate way a process should operate in an event-driven environment. A process, upon receiving an event, should process that event, blocking when and only when it has finished that processing. Once blocked, it should be scheduled again when and only when another event is ready to be processed; an exception is that the process may want to be scheduled periodically to perform periodic tasks, such as blinking the cursor or checking whether it is time to do backups. We say that a process is acting greedily when it fails to block even after it has finished processing an event. This can occur when a process busy waits in some manner, e.g. it loops on "check for event." When we determine a process is acting greedily, we will forcibly block that process for a set period of time.

MacOS uses cooperative multitasking, meaning that once a process gets control of the processor, it retains that control until it chooses to yield control. For this reason, application writers are strongly encouraged to have their processes yield control periodically, even when they still have work to do. Processes indicate that they still have work to do by specifying a sleep period of zero, thereby failing to block. We call the period of time between when a process gains control of the processor and when it yields control a quantum.

Part of our technique is a heuristic to determine when a process is acting greedily. We say that a process is acting greedily when it specifies a sleep period of zero even though it seems not to be actively computing. We consider a process to start actively computing when it receives an event or shows some signs of "activity," as defined below. We estimate that a process is no longer actively computing if it explicitly blocks, or if it yields control several times in a row without receiving an event or showing signs of activity. The exact number of control-yield times, which we call the *greediness threshold*, is a parameter of the technique; we expect it to be set so as to maximize energy savings, given a desired level of performance. We say that a process shows no sign of activity if it performs no I/O device read or write, does not have the sound chip on, does not change the appearance of the cursor, and does not have the cursor appear as a watch. The absence of activity as we have so defined it implies that either the CPU is idle, the process running is busy waiting in some manner, or the process running is violating the MacOS human interface guidelines that we mentioned earlier.

The greediness technique works as follows. When the OS determines that a process is acting greedily as defined above, it blocks it for a fixed period called the *forced sleep period*. The forced sleep period is a parameter to be optimized, with the following tradeoff: a short sleep period saves insufficient power, while a long sleep period may, in the case that our heuristic fails, block a process that is actually doing something useful.

3 Methodology

3.1 Evaluation of strategies

Evaluation of a strategy requires measuring two consequences of that strategy: processor energy savings and performance impact. *Processor energy savings* is easy to deduce from a simulation, since it is essentially the percent decrease in the time the processor spends in the high-power state. In contrast, *performance impact*, by which we mean the percent increase in workload runtime as a result of using a power-saving strategy, is difficult to measure. This performance penalty stems from the fact that a power saving strategy will sometimes cause the processor not to run when it would otherwise be performing useful work. Such work will wind up having to be scheduled later, making the workload take longer to complete. Without detailed knowledge of the purpose of instruction sequences, it is difficult for a tracer to accurately determine what work is useful and what is not, so our measure will necessarily be inexact.

We have decided to use the same heuristic used in the greediness technique to determine when the processor is doing useful work. In other words, we will call a quantum *useful* if, during that quantum, there is any I/O device read or write, the sound chip is on, there is any change to the cursor, or the cursor appears as a watch. It might be objected that using the same heuristic in the evaluation of a strategy as is used by that strategy is invalid. However, remember that a strategy does not have prior knowledge of when a quantum will be useful, whereas the evaluation system does. Thus, we are evaluating the accuracy of our guess that a quantum will be useful or useless.

We must also account for the time not spent inside application code in the original trace. We divide this time into time spent switching processes in and out, time spent context switching, time the OS spent doing useful work, and OS idle time. The OS is considered to be doing useful work whenever it shows signs of activity that would cause a process quantum to be labeled useful. Such useful work is scheduled in the simulations immediately after the quantum that it originally followed is scheduled, on the assumption that most significant OS work is necessitated by the actions of the process that just ran. Idle time is identified whenever no process is running and the operating system is not doing useful work for a continuous period over 16 ms. We chose this threshold for two reasons. First, it is the smallest time unit used for process scheduling by MacOS, so we expect any decision to idle to result in at least this much idleness. Second, 16 ms is much greater than the modal (most common) value of interprocess time, indicating that it is far longer than should ever be needed to merely switch between processes. Finally, context switch time is assumed to occur any time a process switches in that is different from the one that last switched out; context switches are considered to take 0.681 ms, the observed difference between the average interprocess time when no context switch occurs and the average interprocess time when a context switch occurs.

3.2 Tools

There are three main tools we used to perform our simulations. The first tool, *IdleTracer*, collects traces of events needed to simulate the different strategies, and is discussed in more detail in [6]. Specifically, it records the time of occurrence and other details about the following events: tracing begins or ends, the machine goes to or wakes from sleep, a process begins or ends, the sound chip is turned on or off, the cursor changes, the mouse starts or stops moving, an I/O device is read or written, a process obtains or yields control of the processor, or an event is placed on the event queue. *IdleTracer* only collects data while the machine it is tracing is running on battery power, since that is when processor energy savings is most important, and we want our analysis to reflect the appropriate workload. Also, *IdleTracer* shuts off processor power management while it is tracing, so that the traces it uses are not confounded by the current strategy and thus can be used to simulate any strategy. *IdleTracer* makes use of the *SETC* [12] module, a set of routines for tracing and counting system events.

The second tool, *ItmSim*, simulates power management methods using the current MacOS inactivity-threshold strategy, and provides a basis for comparison. In other words, it simulates the strategy that turns off the processor when there has been no activity (as defined earlier) in the last two seconds and no I/O activity in the last fifteen seconds. In actuality, during periods that the processor is supposed to be off, MacOS will occasionally turn the processor on for long enough to schedule a process quantum. This is done to give processes a chance to demonstrate some activity and put an end to processor power management, in case the processor was shut off too soon. The details of how process quanta are scheduled while the processor is supposed to be off is proprietary and thus is not described here; however, *ItmSim* does attempt to simulate this aspect of the strategy. To give an idea of the consequences of this proprietary modification, our simulations showed that for the aggregate workload we studied, it decreased the performance impact measure from 1.93% to 1.84%, at the expense of decreasing processor off time from 29.77% to 28.79%. This particular proprietary modification, therefore, has only a trivial effect on the power savings.

When, in the simulation, the processor comes back on due to some activity, any quanta in the original trace that preceded that activity but have not yet been scheduled are divided into two categories, useful and non-useful. Useful quanta are immediately scheduled, delaying the rest of the trace execution and thus contributing to the performance impact measure. Non-useful quanta are discarded and never

User number	1	2	3	4	5	6
Machine	Duo 280c	Duo 230	Duo 280c	Duo 280c	Duo 280c	Duo 280c
MacOS version	7.5	7.5.1	7.5	7.5	7.5.1	7.5
RAM size	12 MB	12 MB	12 MB	12 MB	12 MB	12 MB
Hard disk size	320 MB	160 MB	320 MB	320 MB	320 MB	240 MB
Trace length (hr:min:sec)	2:48:34	3:01:21	9:09:00	5:26:41	4:52:55	4:14:52

Table 1: Information about the six users traced.

scheduled. Any useful OS time associated with these quanta is also immediately scheduled, contributing to the performance impact measure.

The third tool, *AsmSim*, simulates the basic strategy with the simple scheduling technique, along with zero or more of our two other suggested techniques: sleep extension and greediness. The parameters for these techniques may be varied at will in the simulations. When, in the simulation, an event becomes ready for a process, all quanta of that process preceding the receipt of the ready event that have not yet been scheduled will be treated as described above, i.e. all useful quanta will be run immediately (before the power-up event), all useless quanta will be discarded, and any useful OS time associated with such quanta will also be run immediately. Even for periodic processes, we schedule quanta in the order in which they occurred. For example, if after its quantum i a process originally slept for 1 second but is actually awoken after 4 seconds, then at that point we schedule quantum $i + 1$, not some later quantum. Note that this approach may cause inaccuracies in the simulation, since the process might in reality check how long it has been since it last went to sleep, and act differently seeing that 4 seconds have passed than it did when only 1 second had passed. We expect and hope that such dependence of process action on time is rare enough that this does not introduce significant errors into the results of our simulations.

3.3 Traces

The traces were collected from six users, each an engineer at Apple Computer, Inc. Table 1 indicates data about the traces obtained from each user and the machines on which those traces were collected. Much more detailed discussion of the traces and their collection appears in [6]. Most results we present will concern the aggregate workload, i.e. the trace composed of the concatenation of all six of these traces.

4 Results

In this section, we refer to the *C*urrent MacOS strategy as strategy C and the *B*asic strategy as strategy B. We append the letter I to indicate use of the s*I*mple schedule technique, append the letter G to indicate use of the *G*reediness technique, and append the letter S for the *S*leep extension technique. Note that we never simulate the greediness technique or sleep extension technique without the simple scheduling technique, since they are designed as supplements to the simple scheduling technique.

4.1 Per-strategy results

The first thing we shall do is determine the *optimal* energy savings attainable. An optimum strategy would schedule only time that was spent doing useful work, and would entirely omit non-useful time; its performance impact would be zero, since it would have foreknowledge of when useful work would occur and arrange to have the processor on when it happens. Simulation indicates that such a strategy would yield an energy savings of 82.33%; thus, this is an absolute ceiling on what can be obtained by any realizable strategy. This is a remarkably high figure—what it says is that the processor is doing useful computation during only 17.67% of the 29.56 hours of the trace; the rest of the time is busy waiting by a user process or idling.

The second simulation results concern strategy C. We find from simulation that strategy C yields an energy savings of 28.79% along with a performance impact measure of 1.84%. In other words, it causes the processor to consume only 71.21% of the energy it would without a power-saving strategy, but increases overall workload completion time by 1.84%. The strategy increases processor energy consumption by 303% compared with the optimal strategy, since it only recovers 35% of the real idle time. Note also that since only 17.67% of the CPU time is actually useful, the performance impact of 1.84% means that we have misclassified 10% of the useful CPU time, and have had to run that work in a delayed manner. Thus, the actual real time delay perceived by the user may not be 1.84%, but may be closer to 10%, since the user waits for a reply only during periods of real, useful, work.

The next simulation results concern strategy B, which turns off the process when and only when there was idling in the original trace. Strategy B has an energy savings of 31.98% and a performance impact of 0%. Thus, we see that the basic strategy without any new process management techniques saves slightly more energy than the current technique, and has no impact on performance. However, it causes the processor to consume 285% more energy than under the optimal strategy, since it only recovers 39% of real idle time.

The next simulation results concern strategy BI. Strategy

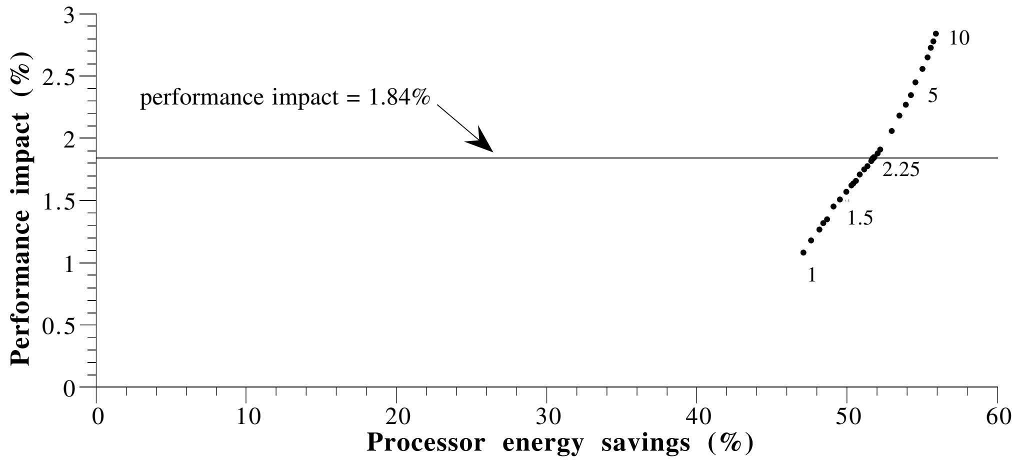


Figure 1: Performance impact measure versus processor energy savings for strategy BIS with various sleep multipliers. Certain points are labeled with the sleep multipliers to which they correspond.

BI has an energy savings of 47.10% and a performance impact of 1.08%. Thus, we see that strategy BI decreases processor energy consumption by 26% and decreases workload completion time by 0.7% compared to strategy C. Compared to the optimal strategy, it causes the processor to consume 199% more energy, since it only recovers 57% of real idle time.

The next simulation results concern strategy BIS. Figure 1 shows the performance versus energy savings graph for variations of this strategy using sleep multipliers between 1 and 10. We see that the point at which this strategy has performance impact of 1.84%, equal to that of strategy C, corresponds to a sleep multiplier of 2.25 and a processor energy savings of 51.72%. Thus, we see that, comparing strategies BIS and C on equal performance grounds, strategy BIS decreases processor energy consumption by 32%. Increasing the sleep multiplier to 10 saves 55.93% of the CPU energy, with a performance impact of 2.84%. Note, however, that the performance impact measure does not tell the whole story in this case. Generally, a real time delay is used by some process that wakes up, checks something, and if certain conditions are met, does something. A very large real time delay in the wakeup period may mean that certain checks are not made in a timely manner; we have ignored that issue here. In practice, sleep extension factors over some level, perhaps 3 to 5, may not be desirable.

The next simulation results concern strategy BIG. Figure 2 shows the performance versus energy savings graph for variations of this strategy using greediness thresholds between 20 and 80 and forced sleep periods between 0.025 seconds and 10 seconds. We find, through extensive exploration of the parameter space, that the parameter settings giving the best energy savings at the 1.84% performance impact level are a greediness threshold of 61 and a forced sleep period of 0.52 seconds. These parameters yield an energy savings of 66.18%. Thus, we see that, comparing strategies BIG and C on equal performance grounds, strategy BIG reduces processor energy consumption by 53%. Compared to the optimal strategy, it increases processor energy consumption by 91%, since it only saves 80% of real idle time.

The next results we present concern strategy BIGS. Figure 3 shows that, in the realm we are interested in, a performance impact of 1.84%, increasing the sleep multiplier always produces worse results than changing the greediness threshold and forced sleep period. The energy savings attainable by increasing the sleep multiplier can be attained at a lower performance cost by instead decreasing the greediness threshold or by increasing the forced sleep period. Thus, the best BIGS strategy is the BIG strategy, which does not make any use of the sleep extension technique. The figure suggests that if we could tolerate a greater performance impact, such as 2.7%, this would no longer be the case, and the best energy savings for BIGS would be attained at a sleep multiplier above one. We conclude that for some values of performance impact, it is useful to combine the greediness technique and sleep extension technique, but for a performance impact of 1.84% it is useless to use the sleep extension technique if the greediness technique is in use.

A summary of all the findings about the above strategies can be seen in Table 2, as well as the columns of Figure 5 corresponding to users 1–6.

4.2 Sensitivity to parameter values

An important issue is the extent to which the parameters we chose are specific to the workload studied, and whether they would be optimal or equally effective for some other workload. Furthermore, it is unclear how effective the user or operating system could be at dynamically tuning these parameters in the best way to achieve optimal energy savings at a given level of performance. Thus, it is important to observe the sensitivity of the results we obtained to the particular val-

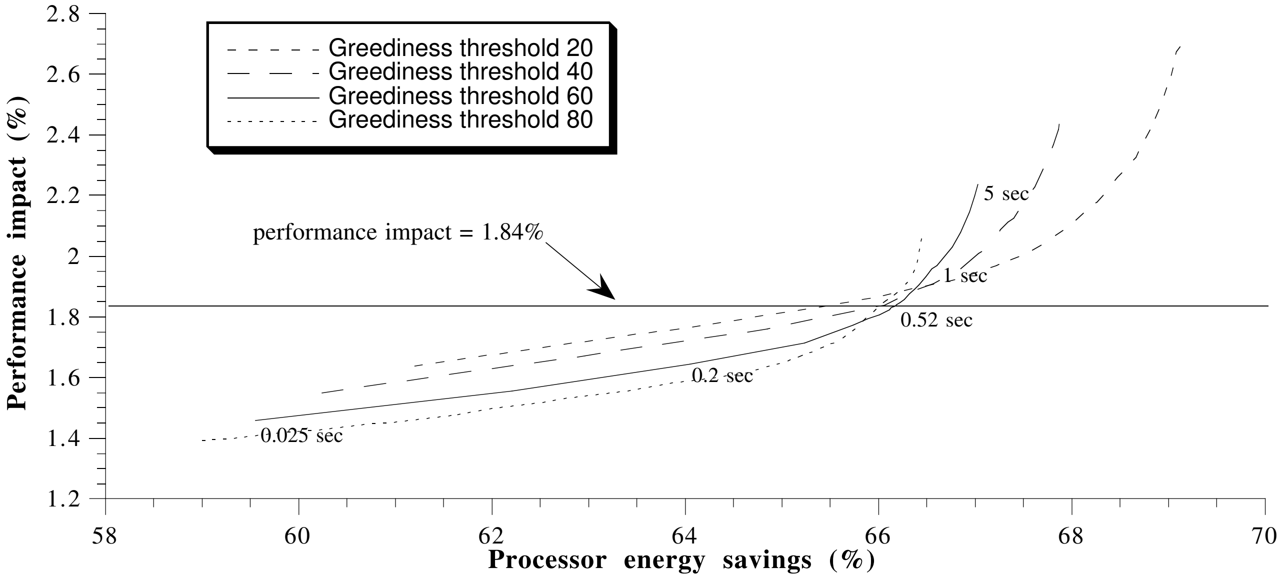


Figure 2: Performance impact measure versus processor energy savings for strategy BIG with various greediness thresholds and forced sleep periods. Points on the greediness threshold 60 curve are labeled with the forced sleep periods to which they correspond. The reader is cautioned that nonzero origins are used in this figure to save space and yet have sufficient resolution to enable its key features to be discerned.

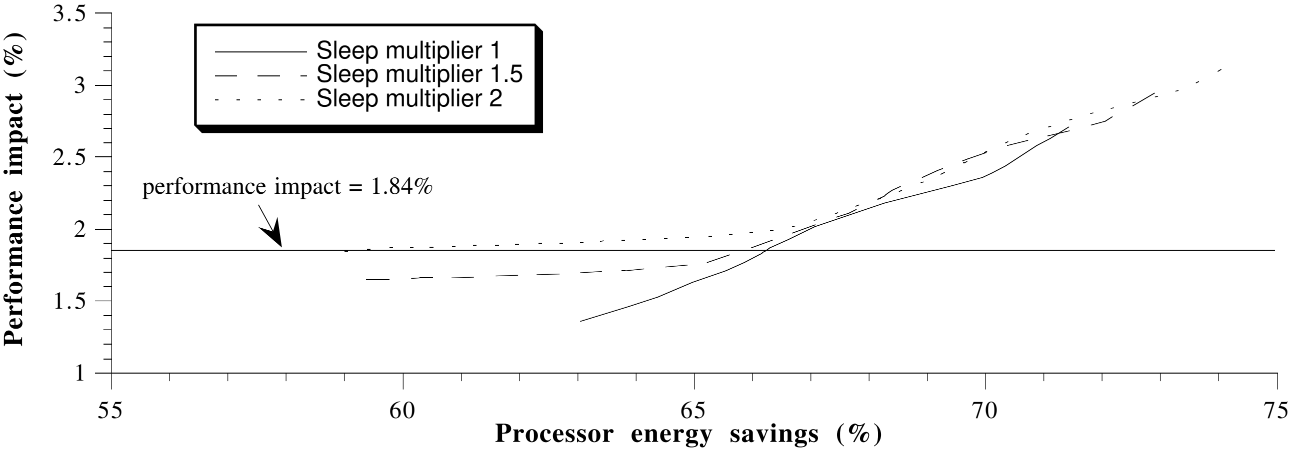


Figure 3: Performance impact measure versus processor energy savings for strategy BIGS with various sleep multipliers, various greediness thresholds, and a forced sleep period of 0.52 sec. The reader is cautioned that nonzero origins are used in this figure to save space and yet have sufficient resolution to enable its key features to be discerned.

Strategy	Processor power savings	Performance impact
Optimal	82.33%	0.00%
C	28.79%	1.84%
B	31.89%	0.00%
BI	47.10%	1.08%
BIS	51.72%	1.84%
BIG	66.18%	1.84%

Table 2: Simulation results for each strategy on the aggregate workload. Strategy BIS achieves the same performance impact as strategy C by using a sleep multiplier of 2.25; strategy BIG achieves this performance impact by using a greediness threshold of 61 and a forced sleep period of 0.52 sec.

ues of the parameters we chose.

The graphs we showed that demonstrate the relationship between performance, energy savings, and parameter values also demonstrate the reasonably low sensitivity of the results to the parameter values. For instance, varying the forced sleep period threshold in Figure 2 across a wide range of values only causes the consequent energy savings to vary between 59–67%. Varying the greediness threshold in Figure 4 across another wide range of values only causes the consequent energy savings to vary in the range 63–71%. Finally, varying the sleep multiplier across a wide range, as in Figure 1, only causes the consequent energy savings to vary in the range 47–56%.

Another way to gauge the sensitivity of the results to the parameters is to evaluate the effectiveness of the techniques on each of the six workloads corresponding to the users studied. To show the effect of using parameters tuned to an aggregate workload on individual users, Figure 5 shows the processor energy savings that would have been attained by each of the users given the strategies we have discussed. We see from this figure that strategy BIG is always superior to strategy C, and that strategy BIS is superior to strategy C for all users except user 2. And, even in this case, the fault seems to lie with the basic strategy and simple scheduling technique rather than the sleep multiplier parameter, since user 2 is also the only user for which the savings from C are much greater than those from strategies B and BI. These figures suggest that even parameters not tuned for a specific workload still yield strategies that in general save more processor energy than the current strategy. It is also interesting to note that there is a clear ordering between strategies BI, BIS, and BIG: for each user, strategy BIG saved more energy than strategy BIS, which saved more energy than strategy BI.

We were curious why strategy C is so much superior to strategy B for user 2, so we inspected the simulation results for that user carefully. We found that the reason strategy C does so much better than strategy B is that in that trace, the application Finder (discussed further later) frequently yields control requesting a sleep time of zero but then performs no activity when it gets control again; indeed, there is a contiguous section of the trace lasting over an hour (a third of the trace) during which Finder has this behavior. When this happens, the basic strategy, strictly obeying the request of Finder to never block, never gets a chance to cycle the processor, while on the other hand the current strategy notices that no activity is occurring and turns off the processor anyway. The sleep extension technique does not alleviate this problem, since multiplying the sleep request of zero by any factor still makes it zero. However, the greediness technique is able to overcome this problem, since this is exactly the problem for which it was designed. Consequently, strategy BIG beats strategy C for user 2, even though strategies B, BI, and BIS do not.

Yet another way to see that the basic strategy with the new techniques is effective even without tuning the parameters is to pick somewhat arbitrary parameters and note that the energy savings are still superior to that of strategy C. For example, the parameter settings we envisioned before running any of these simulations, a greediness threshold of 5, a forced sleep period of 0.25 seconds, and a sleep multiplier of 1.5, would yield a respectable energy savings of 71.70% and a performance impact of 2.61%, which, compared to MacOS, trades off a 60% decrease in processor energy consumption for a 0.8% increase in workload completion time. Even a conservative set of parameters, namely a greediness threshold of 100, a forced sleep period of 0.10 seconds, and a sleep multiplier of 1, yields a processor energy savings of 62.87% with a performance impact of only 1.48%, decreasing processor energy consumption by 48% and reducing workload completion time by 0.4% compared to strategy C.

4.3 Additional explanation of results

We have seen that the greediness technique by itself can be quite effective even within a broad range of parameter values. This suggests that processes often act greedily. In fact, using the ideal parameters of strategy BIG, 49 of 63 applications were found to act greedily at some point. The percent of an application's quanta during which it was determined to be acting greedily varied widely from one application to another, from 0.0009% for Express Modem to 93.7% for eWorld. It is especially amusing that Finder, the user desktop interface application written at Apple itself, seems to lie routinely about its processing time needs, having been determined to be acting greedily for 64.7% of its quanta. This supports our suggestion that designers of operating systems and applications for single-user systems are not generally concerned with rigorous management of processor time.

To examine where the savings and performance impact of a strategy are coming from, it is useful to observe what happens to different classes of time in the original trace when that strategy is used. Some time in the original trace is spent running processes, some is spent running the scheduler, and the rest is spent running other OS code. Each of these classes of time can be broken down into useful and nonuseful time. Useful time will always be scheduled in the simulation, but some of it will be delayed if the technique simulated decides to cycle instead of running it. This delayed time is what contributes to performance impact. Some nonuseful time will be scheduled by the technique, while the rest gets skipped over and contributes to cycling time. This skipped time is what contributes to energy savings. Table 3 shows this breakdown in time and number of quanta for each strategy.

First, let us compare the sources of the energy savings for each strategy; this involves looking at the italic figures in Table 3. Strategy C spends 4.31 hours cycling instead of idling in the OS, 1.95 hours cycling instead of performing nonuseful process quanta, and 2.61 hours cycling instead of switching nonuseful process quanta in and out. It is interesting to note that more time is saved from not having to switch pro-

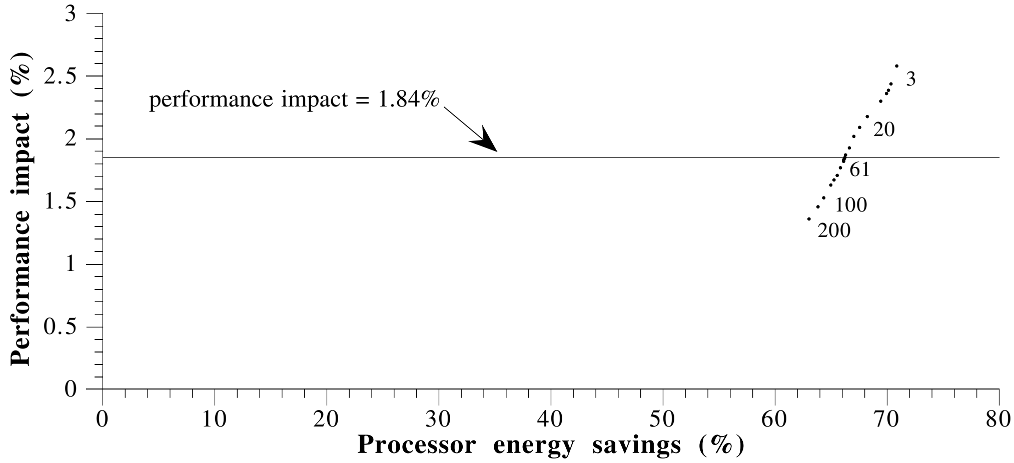


Figure 4: Performance impact measure versus processor energy savings for strategy BIG with forced sleep period of 0.52 seconds and various greediness thresholds. Certain points are labeled with the greediness thresholds to which they correspond.

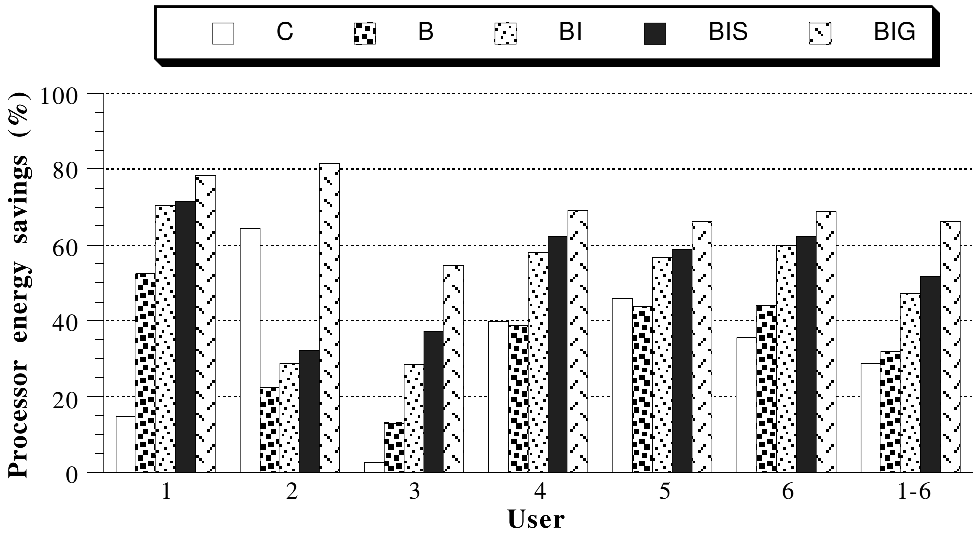


Figure 5: Processor energy savings for each strategy and each user. Strategy BIS uses a sleep multiplier of 2.25, while strategy BIG uses a greediness threshold of 61 and a forced sleep period of 0.52 seconds.

Strategy	Useful time (hours) Process	Useful time (hours) Scheduler	Useful time (hours) OS	Useful quanta	Nonuseful time (hours) Process	Nonuseful time (hours) Scheduler	Nonuseful time (hours) OS	Nonuseful quanta
C	**0.27**; 3.73	**0.23**; 0.63	**0.04**; 0.33	**88,055**; 473,075	2.38; *1.59*	8.31; *2.61*	5.14; *4.31*	6,303,597; *1,682,648*
B	**0.00**; 4.00	**0.00**; 0.86	**0.00**; 0.37	**0**; 561,130	3.96; *0.00*	10.92; *0.00*	0.00; *9.45*	7,986,245; *0*
BI	**0.20**; 3.80	**0.02**; 0.84	**0.10**; 0.27	**20,027**; 541,103	2.97; *1.00*	7.45; *3.47*	0.00; *9.45*	5,965,396; *2,020,849*
BIS	**0.37**; 3.63	**0.04**; 0.82	**0.12**; 0.25	**29,132**; 531,998	2.81; *1.15*	6.23; *4.69*	0.00; *9.45*	5,412,942; *2,573,303*
BIG	**0.33**; 3.67	**0.04**; 0.82	**0.18**; 0.19	**32,130**; 529,000	0.87; *3.09*	3.90; *7.02*	0.00; *9.45*	2,420,825; *5,565,420*

Table 3: A breakdown, for each strategy, of what happens to the time and quanta originally spent running processes, the scheduler, and the operating system. Time and quanta that are delayed and thus contribute to performance impact are shown in **bold**, time and quanta that are run on time are shown in the standard font, and time and quanta that are never run and thus are subsumed by cycling are shown in *italics*.

cesses in and out than from actually not running them. This is probably because the technique attempts to cycle the processor when processes are not doing anything, and when a process is not doing anything it should be doing little besides getting switched in and out. Strategy B never cycles instead of running or scheduling process quanta, but it makes up for this by always cycling when the OS would otherwise be idle, 9.45 hours. Strategies BI and BIS cycle even more because they can cycle instead of running or scheduling process quanta. Indeed, even if we did not consider idle time spent cycling, strategies BI and BIS still save more nonuseful process and scheduler time than strategy C. Interestingly, the increased savings stem from reducing the amount of time spent *scheduling* nonuseful process quanta, not from reducing the time spent *running* nonuseful process quanta. In fact, each of strategies BI and BIS spends longer running nonuseful process quanta than strategy C. The reason these strategies spend less time scheduling nonuseful process quanta is that, as shown in the table, the strategies schedule fewer nonuseful process quanta, and fewer quanta to schedule means less time spent scheduling quanta. So we see that the savings from strategies BI and BIS stem not from giving processes less time when they are not doing useful work, but from switching them in and out less often when they are not doing useful work, thus saving time associated with process switching. On the other hand, strategy BIG is superior to strategy C in its conversion of all types of nonuseful time. So, the savings from strategy BIG stems both from giving processes control less often when they are not busy and from giving them less time to run useless tasks.

Table 3 also allows us to compare the sources of the performance impact in each strategy; this involves looking at the bold numbers. We see that for strategy C, about half of the performance impact is due to delaying the running of useful process quanta and half is due to delaying the switching of those quanta. On the other hand, strategy BI delays about the same amount of useful process run time, but far less time spent switching process quanta. This is explained by the fact that it delays a much lower absolute number of useful process quanta. The low amount of delayed scheduler time in strategy BI extends to BIG and BIS, allowing them to have a greater amount of delayed process quantum and OS time while still maintaining the same performance impact as strategy C. Our main observation, then, is that strategies BI, BIG, and BIS delay far fewer useful quanta than strategy C even though by design the latter two delay the same amount of total useful time.

5 Discussion

In our simulations, we found the following. The basic strategy by itself saves barely more energy than the current MacOS strategy, although it does have less performance impact. Adding the simple scheduling technique allows the basic strategy to reduce energy consumption by 26% and to reduce workload completion time by 0.7% compared to the current MacOS strategy. Adding further techniques for managing processor time allows the basic strategy to even further surpass the current MacOS strategy in energy savings for the same level of performance impact. Using the sleep extension technique allows 32% less processor energy consumption, and using the greediness technique allows 53% less processor energy consumption. In all cases, the absolute level of performance impact is very low. Note, however, that at best we get about 66% absolute energy savings, compared to the approximately 82% that would be available with optimal power reduction.

Our simulations suggest that although both the greediness technique and the sleep extension technique are helpful in increasing the savings attainable from the basic strategy with the simple scheduling technique, the greediness technique does this job better than the simple scheduling technique. Furthermore, when attempting to achieve the same performance impact the current MacOS provides, it is not worthwhile to use the sleep extension technique if one is already using the greediness technique. The reason for this is that the cost in performance impact of an increase in energy savings is greater for the sleep extension technique than for the greediness technique. In other words, adjusting the parameters of any technique to improve its energy savings will necessarily cause it to cycle more often when the processor is doing useful work and thus increase performance impact. It happens that this effect is more pronounced for the sleep extension technique than for the greediness technique, at least for the low performance impact regime in which we are interested. Besides this difference between the two techniques, another reason to not use the sleep extension technique is that it is quite possible that extending real-time sleep times will produce undesirable effects in terms of system performance or functionality.

Analysis of the sensitivity of the techniques to changes in parameter settings and workloads suggests that ideal parameter setting is not necessary to the consistent functioning of the techniques. Thus, we expect the techniques to be successful with even naive parameter-setting by the operating system or user. However, one anomalous case, user 2, points out the vulnerability of the basic strategy to poor application behavior, suggesting that the greediness technique or something like it should always accompany the basic strategy.

To illustrate how percent time in low-power mode would translate into overall power savings, let us consider an example based on estimates of power consumption from previous work [7, 8]. In this example, based on the Duo 280c, turning off the processor saves 3.74 W, while the components that remain on consume, on average, 5.65 W, given current power management techniques. From these figures, the current strategy, with processor energy savings of 28.79% and performance impact of 1.84%, would make average total power consumption

$$5.65\,\mathrm{W} + \left(\frac{1-0.2879}{1+0.0184}\right)(3.74\,\mathrm{W}) = 8.27\,\mathrm{W}.$$

Note that we are assuming that during extra time spent with the processor on, other components are, on average, at their average power levels given current power management techniques. The BIG strategy, with processor energy savings of 66.18% and the same performance impact, would make average total power consumption

$$5.65\,\mathrm{W} + \left(\frac{1-0.6618}{1+0.0184}\right)(3.74\,\mathrm{W}) = 6.89\,\mathrm{W}.$$

thus achieving a 16.7% savings in total power and yielding a 20.0% increase in battery lifetime relative to strategy C, assuming that battery lifetime is inversely proportional to average power drain. The BIS strategy, with its processor energy savings of 51.72% and same performance impact, would make average total power consumption 7.42 W, yielding a power savings of 10.3% and battery lifetime increase of 11.5% compared to strategy C. Note that all these figures are only applicable to the workload studied here, i.e. the 30 hours of traces obtained from these six users. It is unclear how much more or less effective these methods would be for some other workload; for instance, a previous study with a different workload [7, 8] found that 48.1% of processor power was saved by the current MacOS strategy. Even within the workload studied here, we have seen that the savings from each strategy varies greatly from one user to another.

We believe that even though our simulations were performed on MacOS traces, our techniques are generally applicable to other single-user operating systems. For example, Microsoft Windows essentially uses the basic strategy with the simple scheduling technique. However, it makes no attempt to police processes that make unfair processor time requests; in fact, one recognized problem with its power management is that if a single process requests events using `PeekMessage` rather than `GetMessage` or `WaitMessage`, then processor power management cannot take place [2, 9]. This problem is exactly the sort that the greediness technique was designed to alleviate, so we may find that, as we demonstrated for the MacOS, the effectiveness of the basic strategy is greatly improved by the use of such a technique.

We would like to see the work described here continued in several ways. First, we would like to actually implement our proposed strategies. This would have the advantage of demonstrating their feasibility and utility in an easily measured way, since one would be able to measure the power consumption and performance of systems with different strategies running the same benchmark workloads. Second, we would like to collect additional traces, so that the effectiveness of our techniques over a larger variety of users and workloads can be established, so that we can better select parameter values, and so that we can determine if dynamic variation of parameter values is useful. In particular, we want to obtain traces from different types of user environments; our current set of traces is rather limited in this respect. Third, we would like to see if our power-saving techniques are applicable in practice to other single-user operating systems (such as those from Microsoft). Finally, we do not believe that we have exhausted the possibilities for software strategies to put the CPU into power saving modes; our work to date has not, for example, considered systems in which the clock rate and voltage can be varied simultaneously and dynamically. We also do not believe that we have found or used all of the information useful in predicting "useless" quanta. At best, we save around 66% of the CPU energy, compared to a theoretical optimum of 82%.

6 Conclusions

Reducing the power consumption of computer systems is becoming an important factor in their design, especially in the case of portable computers. An important component for power management is the processor, as it accounts for a large percentage of total power. Obviously, the CPU can be turned off when it is inactive; in some operating systems, however, such as MacOS, the CPU is frequently running even when there isn't any "useful" work to do. We have found several heuristic techniques to decrease power use, such as (a) never running a process that is still blocked and waiting on an event, (b) delaying processes that execute without producing output or otherwise signaling useful activity, and (c) delaying the frequency of periodic processes, most of which seem to wake up, look around, find nothing interesting to do, and go back to sleep. To the extent that similar phenomena occur in other operating systems, these techniques should apply to them also.

We have used trace-driven simulation to evaluate these techniques on the basis of processor energy saved and increase in processing time. We found that our techniques save 47–66% of processor energy, thus decreasing processor energy consumption by 26–53% compared to the simple strategy currently employed by MacOS. Taking into consideration the fraction of system power used by the CPU, we estimate that implementing the best subset of our techniques will increase battery lifetime for Macintosh portables by around 20%. These savings do imply a small loss in effective system performance (due to the fact that the CPU may be inactive even though there is real work to do); in all cases, this loss is less than 2%.

Work is continuing on this topic. We hope to be able to implement our proposed algorithms, although this work may or may not find its way into Apple products. We will also be collecting additional traces, for a wider variety of workloads, and will be further evaluating the techniques described here and others yet to be discovered.

7 Acknowledgments

Many people aided in the preparation of this work and deserve our gratitude here. Marianne Hsiung provided support, on behalf of Apple Computer, Inc., for this research. Phil Sohn wrote the *SETC* module that *IdleTracer* uses, and also provided general guidance during the project. Steve Sfarzo arranged the distribution of the tracer to, and the collection of data files from, many PowerBook users working with him. Helder Ramalho provided invaluable information about the way the current processor cycling technique works. Dave Falkenburg and Jim Goche provided information about trap handler programming critical to the writing of *IdleTracer*. Larry Heil coordinated cooperation with the Apple Portables group. Several users ran the tracer on their machines to help us collect the data. And, finally, we thank the reviewers of the original version of this paper for their helpful and insightful comments.

References

[1] Apple Computer, Inc. *Inside Macintosh, Volume VI*, Addison Wesley Publishing Company, Reading, MA, 1991.

[2] Conger, J. *Windows API Bible*, Waite Group Press, Corte Madera, CA, 1992.

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

[4] Gary, S., Dietz, C., Eno, J., Gerosa, G., Park, S., and Sanchez, H. The PowerPC^{TM} 603 microprocessor: a low-power design for portable applications. *Proceedings of the IEEE International Computer Society Conference*, San Francisco, CA, 307–315, February 1994.

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

[6] Lorch, J. Modeling the effect of different processor cycling techniques on power consumption. *Performance Evaluation Group Technical Note #179*, ATG Integrated Systems, November 1995.

[7] Lorch, J. A complete picture of the energy consumption of a portable computer. *Masters Thesis*, Computer Science, University of California at Berkeley, December 1995.

[8] Lorch, J. and Smith, A. J. How energy is consumed and saved in portable computers. *In preparation*, 1996.

[9] Pietrek, M. *Windows Internals*, Addison Wesley, Reading, MA, 1993.

[10] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: an operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[11] Ritchie, D. M. and Thompson, K. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–275, July 1974.

[12] Sohn, P. SETC: A System Event Tracer and Counter. *Apple Computer Inc. Performance Evaluation Group Technical Report*, June 1994.

[13] Srivastava, M. B., Chandrakasan, A. P., and Broderson, R. W. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):42–55, March 1996.

[14] Suessmith, B. and Paap, G., III. PowerPC 603 microprocessor power management. *Communications of the ACM*, 43–46, June 1994.

[15] Suzuki, N. and Uno, S. Information processing system having power saving control of the processor clock. *United States Patent #5,189,647*, February 1993.