Policies for Dynamic Clock Scheduling

Dirk Grunwald
Charles B. Morrey III
Philip Levis
Michael Neufeld

{grunwald,levis,cbmorrey,neufeldm}@cs.colorado.edu
Department of Computer Science
University of Colorado
Boulder, Colorado

Keith I. Farkas

keith.farkas@compaq.com
Compaq Computer Corp.
Western Research Lab
Palo Alto, California

Abstract

Pocket computers are beginning to emerge that provide sufficient processing capability and memory capacity to run traditional desktop applications and operating systems on them. The increasing demand placed on these systems by software is competing against the continuing trend in the design of low-power microprocessors towards increasing the amount of computation per unit of energy. Consequently, in spite of advances in low-power circuit design, the microprocessor is likely to continue to account for a significant portion of the overall power consumption of pocket computers.

This paper investigates *clock scaling* algorithms on the Itsy, an experimental pocket computer that runs a complete, functional multitasking operating system (a version of Linux 2.0.30). We implemented a number of clock scaling algorithms that are used to adjust the processor speed to reduce the power used by the processor. After testing these algorithms, we conclude that currently proposed algorithms consistently fail to achieve their goal of saving power while not causing user applications to change their interactive behavior.

1 Introduction

Dynamic clock frequency scaling and voltage scaling are two mechanisms that can reduce the power consumed by a computer. Both voltage scaling and frequency scaling are important; the power consumed by a component implemented in CMOS varies linearly with frequency and quadratically with voltage.

To evaluate the relative importance and the situations in which either is useful, it is necessary to consider energy, the integral of power over time. By reducing the frequency at which a component operates, a specific operation will consume less power but may take longer to complete. Although reducing the frequency alone will reduce the average power used by a processor over that period of time, it may not deliver a reduction in energy consumption overall, because the power savings are linearly dependent on the increased time. While greater energy reductions can be obtained with slower clocks and lower voltages, operations take longer; this exposes a fundamental tradeoff between energy and delay.

Many systems allow the processor clock to be varied. More recently, there are a number of processors that allow the processor voltage to be changed. For example, the StrongARM SA-2 processor, currently being designed by Intel, is estimated to dissipate 500mW at 600MHz, but only 40mW when running at 150MHz – a 12-fold energy reduction for a 4-fold performance reduction [1]. Likewise, the Pentium-III processor with SpeedStep technology dissipates 9W at 500MHz but 22W at 650MHz [2], AMD has added clock and voltage scaling to the AMD Mobile K6 Plus processor family and Transmeta has also developed processors with voltage scaling. Because of this tradeoff in speed *vs.* power, the decision of when to change the frequency or the voltage and frequency of such processors must be made judiciously while taking into account application demand and quality of user experience.

We believe that the decision to change processor speed and voltage must be controlled by the operating system. The operating system or similar system software is the only entity with a global view of resource usage and demand. Although it is clear that the operating system should control the scheduling mechanism, it is not clear what inputs are necessary to formulate the scheduling

policy. There are two possible sources of information for policies. The application can estimate activity, providing information to the operating system about computation rates or deadlines, or the operating system can attempt to infer some policy for the applications from their behavior. These can be used separately or in concert to control voltage and processor speed.

A number of studies have investigated policies to automatically infer computation demands and adjust the processor accordingly. We have implemented those previously described algorithms; this paper describes our experience.

In the next section, we present some background material. We discuss related work in Section 3. In Section 4 we describe the schedulers we examine, our workload and our measurement methodology. We then discuss our results in Section 5.

2 Background

To better understand the importance of voltage and clock scheduling, we begin by reviewing energy-consumption concepts, then present an overview of scheduling algorithms. Lastly, we give an overview of our test platform, the Itsy Pocket Computer.

2.1 Energy

The energy E, measured in Joules (J), consumed by a computer over T seconds is equal to the integral of the instantaneous power, measured in Watts (W). The instantaneous power consumed by components implemented in CMOS, such as microprocessors and DRAM, is proportional to $V^2 \times F$, where V is the voltage supplying the component, and F is the frequency of the clock driving the component. Thus, the power consumed by a computer to, say, search an electronic phone book, may be reduced by reducing V, F, or both. However, for tasks that require a fixed amount of work, reducing the frequency may result in the system taking more time to complete the work. Thus, little or no energy will be saved. There are techniques that can result in energy savings when the processor is idle, typically through *clock gating*, which avoids powering unused devices.

In normal usage pocket computers run on batteries, which contain a limited supply of energy. However, as discussed in [3], in practice, the amount of energy a battery can deliver (i.e., its capacity) is reduced with increased power consumption. As an illustration of this effect, consider the Itsy pocket computer that was used in this study (described in Section 2.3). When the system is idle, the integrated power manager disables the processor core but the devices remain active. If the system clock is 206 MHz, a typical pair of alkaline batteries will power the system for about 2 hours; if the system clock is set to 59 MHz, those same batteries will last for about 18 hours. Although the battery lifetime increased by a factor of 9, the processor speed was only decreased by a factor of 3.5. The capacity of the battery can also be increased by interspacing periods of high power demand with much longer periods of low power demand resulting in a "pulsed power" system [4]. The extent to which these two non-ideal properties can be exploited is highly dependent on the chemical properties and the construction of a battery as well as the conditions under which the battery is used. In general, the former effect (minimizing peak demand) is more important than the latter for the domain of pocket computers because pulsed power systems need a significant period of time to recharge the battery, and most computer applications place a more constant demand on the battery.

If a system allows the voltage to be reduced when clock speed is reduced (*i.e.* it supports voltage scaling), it is better to reduce the clock speed to the minimum needed rather than running at peak speed and then being idle. For example, consider a computation that normally takes 600 million instructions to complete. That application would take one second on a StrongARM SA-2 at 600MHz and would consume 500 mJoules. At 150MHz, the application would take four seconds to complete, but would only consume 160 mJoules, a four-fold savings assuming that an idle computer consumes no energy. There is obviously a significant benefit to running slower when the application can tolerate additional delay. Pering [5] used the term *voltage scheduling* to mean scheduling policies that seek to adjust both clock speed and energy. The goal of voltage scheduling is to reduce the clock speed such that all work on the processor can be completed "on time" and then reduce the voltage to the minimum needed to insure stability at that frequency.

2.2 Clock Scheduling Algorithms

In scheduling the voltage at which a system operates and the frequency at which it runs, a scheduler faces two tasks: to predict what the future system load will be (given past behavior) and to scale the voltage and clock

frequency accordingly. These two tasks are referred to as *prediction* and *speed-setting* [6]. We consider one scheduler better than another if it meets the same deadlines (or has the same behavior) as another policy but reduces the clock speed for longer periods of time.

The schedulers we implemented are *interval schedulers*, so called because the prediction and scaling tasks are performed at fixed intervals as the system runs [7]. At each interval, the processor utilization for the interval is predicted, using the utilization of the processor over one or more preceding intervals. We consider two prediction algorithms originally proposed by Weiser *et al.* [7]: `PAST` and $\texttt{AVG}_N$. Under `PAST`, the current interval is predicted to be as busy as the immediately preceding interval, while under `AVG`, an exponential moving average with decay N of the previous intervals is used. That is, at each interval, we compute a "weighted utilization" at time t, W_t, as a function of the utilization of the previous interval U_{t-1} and the previous weighted utilization W_{t-1}. The $\texttt{AVG}_N$ policy sets $W_t = \frac{N \times W_{t-1} + U_{t-1}}{N+1}$. The `PAST` policy is simply the $\texttt{AVG}_0$ policy, and assumes the current interval will have the same resource demands as the previous interval.

The decision of whether to scale the clock and/or voltage is determined by a pair of boundary values used to provide hysteresis to the scheduling policy. If the utilization drops below the lower value, the clock is scaled down; similarly, if the utilization rises above the higher value, the clock is scaled up. Pering et al. [8] set these values at 50% and 70%. We used those values as a starting point but, as we discuss in Section 5.3, we found that the specific values are very sensitive to application behavior.

Deciding *how much* to scale the processor clock is separate from the decision of *when* to scale the clock up (or down). The SA-1100 processor used in the Itsy supports 11 different clock rates or "clock steps". Thus, our algorithms must select one of the discrete clock steps. We use three algorithms for scaling: `one`, `double`, and `peg`. The `one` policy increments (or decrements) the clock value by one step. The `peg` policy sets the clock to the highest (or lowest) value. The `double` policy tries to double (or halve) the clock step. Since the lowest clock step on the Itsy is zero, we increment the clock index value before doubling it. Separate policies may be used for scaling upwards and downwards.

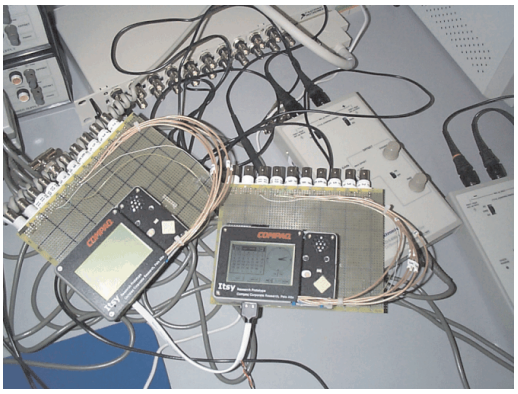


Figure 1: Equipment setups used to measure power.

2.3 The Itsy Pocket Computer

The Itsy Pocket Computer is a flexible research platform, developed to enable hardware and software research in pocket computing. It is a small, low-power, high-performance handheld device with a highly flexible interface, designed to encourage the development of innovative research projects, such as novel user interfaces, new applications, power management techniques, and hardware extensions. There are several versions of the basic Itsy design, with varying amount of RAM, flash memory and I/O devices. We used several units for this study that were modified by Compaq Computer Corporation's Western Research Lab to include instrumentation leads for power measurement. Figure 1 shows the units along with the measurement equipment we used. We investigate the energy and power consumption of the Itsy Pocket Computer when it is run at between 59 MHz and 206 MHz, and when its StrongARM SA-1100 [9, 10] processor is powered at two different voltage levels.

All versions of the Itsy are based on the low-power StrongARM SA-1100 microprocessor. All versions have a small, high-resolution display, which offers 320×200 pixels on a 0.18mm pixel pitch, and 15 levels of greyscale. All versions also include a touchscreen, a microphone, a speaker, and serial and IrDA communication ports. The Itsy architecture can support up to 128 Mbytes both of DRAM and flash memory. The flash memory provides persistent storage for the operating system, the root file system, and other file systems and data. Finally, the Itsy also provides a "daughter card" interface that allows the base hardware to be easily extended. The Itsy uses two voltage supplies powered by the same power source. The processor core is driven by a 1.5 V supply while the peripherals are driven by a 3.3 V

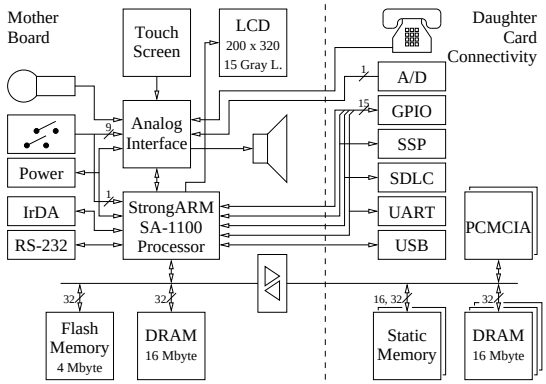


Figure 2: Itsy System Architecture

supply. Both power supplies are driven by a single $3.1V$ supply connected to the electrical mains.

The Itsy version 1.5 units used as the basis for this work have 64 Mbytes of DRAM and 32 Mbytes of flash memory. These units were modified to allow us to run the StrongARM SA-1100 at either 1.5 V or 1.23 V. Although 1.23 V is below the manufacturer's specification, it can be safely used at moderate clock speeds and our measurements indicate the voltage reduction yields about a 15% reduction in the power consumed by the processor; the percentage of power reduction for the *system* may be less than this (depending on workload) because voltage scaling only reduces the power used by the processor. The Itsy can be powered either by an external supply or by two size AAA batteries. Figure 2 shows a schematic of the Itsy architecture.

The system software of the Itsy includes a monitor and a port of version 2.0.30 of the Linux operating system. The Linux system was configured to provide support for networking, file systems and multi-user management. Applications can be developed using a number of programming environments, including C, X-Windows, SmallTalk and Java. Applications can also take advantage of available speech synthesis and speech recognition libraries.

3 Related Work

We believe that our evaluation of dynamic speed and voltage setting algorithms to be the first such empirical evaluation – to our knowledge, all previous work from different groups has relied on simulators [7, 6, 5, 11, 12]; none modeled a complete pocket computer or the workload likely to be run on it.

Weiser et al. [7] proposed three algorithms, `OPT`, `FUTURE`, and `PAST` and evaluated them using traces gathered from UNIX-based workstations running engineering applications. These algorithms use an interval-based approach that determines the clock frequency for each interval. Of the algorithms they propose, only `PAST` is feasible because it does not make decisions using future information that would not be available to an actual implementation. Even so, the actual version of `PAST` proposed by by Weiser *et al.* is not implementable because it requires that the scheduler know the amount of work that had to be performed in the preceding intervals. This information was used by the scheduler to choose a clock speed that allows this delayed work to be completed in the next interval, if possible. For example, suppose post-processing of a trace revealed that the processor was busy 80% of the cycles while running at full speed. If, during re-play of the trace, the scheduler opted to run the processor at 50% speed for the interval, then 30% of the work could not be completed in that interval. Consequently, in the next interval, the scheduler would adjust the speed in an effort to at least complete the 30% "unfinished" work. Without additional information from the application, the scheduler can simply observe that the application executed until the end of the scheduling quanta, and does not know the amount of "unfinished" computing left. Because most pocket computer applications do not provide a means for the processor to know how much work should be done in a given interval, the `PAST` algorithm is not tractable for such systems.

The early work of Weiser et al. has been extended by several groups, including [6, 12]. Both of these groups employed the same assumptions and the same traces used by Weiser. Govil *et al.* [6] considered a large number of algorithms, while Martin [12] revised Weiser's `PAST` algorithm to account for the non-ideal properties of batteries and the non-linear relationship between system power and clock frequency. Martin argues that the lower bound on clock frequency should be chosen such that the number of computations per battery lifetime is maximized. While Martin correctly assumed a non-zero energy cost for idling the processor and changing clock speed, neither Govil nor Weiser did.

Both our work and that of Pering et al. [5, 11] addresses some of the limitations of the above noted earlier work. In particular, we both evaluate *implementable algorithms* using workloads that are representative of those that might be run on pocket computers. We assess the success of our algorithms under the assumption that our applications have *inelastic performance con-*

straints and that the user should see no visible changes induced by the scheduling algorithms. By comparison, Pering *et al.* assume that frames of an MPEG video, for instance, can be dropped and present results which combine a combination of energy savings *vs.* frame rates. Our goal was to understand the performance of the different scheduling algorithms without introducing the complexity of comparing multi-dimensional performance metrics such as the percentage of dropped frames *vs.* power savings.

Pering *et al.* use intervals of 10-50ms for their scheduling calculations. In comparison to the earlier approaches presented in [7, 6, 12] in which work was considered overdue if it was not completed within an interval, both Pering *et al.* and our study consider an event to have occurred on time if delaying its completion did not adversely affect the user. However, a number of important differences exist between our work and Pering *et al.*. First, Pering *et al.* model only the power consumed by the microprocessor and the memory, thus ignoring other system components whose power is not reduced by changes in clock frequency. Second, by virtue of our work using an actual implementation, we are able to evaluate longer running applications and more complex applications (e.g., Java). By virtue of their size, our applications exhibit more significant memory behavior, and thus, expose the non-linear relationship between power and clock speed noted by Martin. Lastly, by using an actual system, our scheduling implementations were exposed to periodic behaviors that are captured by traces; for example, the Java implementation uses a 30ms polling loop to check for I/O events. This periodic polling adds additional variation to the clock setting algorithms, inducing the sort of instability we will explain in §5.3.

4 Methodology

Before describing the implementation of the clock and voltage scheduling algorithms we used, it is important to understand how we did our measurements. Section 4.1 describes how we measure power and energy. We then describe the implementation of the schedulers and the workloads we used to assess their performance.

4.1 Measuring Power and Total Energy

To measure the instantaneous power consumed by the Itsy, we use a data acquisition (DAQ) system to record the current drawn by the Itsy as it is connected to an external voltage supply, and the voltage provided by this supply. Figure 1 presents a picture of our setup along with the wires connected to the Itsy to facilitate measuring the supply current[1] and voltage. We configured the DAQ system to read the voltage 5000 times per second, and convert these readings to 16-bit values. These values were then forwarded to a host computer, which stored them for subsequent analysis. From these measurements, we can compute a time profile of the power used by an application as it runs on the Itsy.

To determine the relevant part of the power-usage profile of a workload, we measure the time required to execute the workload and then select the relevant set of measurements from the data collected by the DAQ system. For each benchmark, we used the `gettimeofday` system call to time its execution; this interface uses the 3.6 MHz clock available on the processor to provide accurate timing information. To synchronize the collection of the voltages with the start of execution of a workload, as the workload begins executing, we toggle one of the SA1100's general-purpose input-output (GPIO) pins. This pin is connected to the external trigger of the DAQ system; toggling the GPIO causes the DAQ system to begin recording measurements. As our measurement technique is very similar to that which we used in [13], we refer the reader to this reference for a more in-depth description.

Once the relevant part of the profile has been determined, we use it to calculate the average power and the total energy consumed by the Itsy during the corresponding time interval. To compute the energy, we make the assumption that the power measured at time t represents the average power of the Itsy for the interval t to $t + 0.0002$ seconds, where 0.0002 seconds is the time between each successive power measurement. Thus, the energy E is equal to $\sum_{i=1}^{n} p_i(t) \times 0.0002$, where $p_1(t), \ldots, p_n(t)$ are the n power readings of interest.

In making our power measurements, we used a similar approach as the one used in [13] to reduce a number of sources of possible measurement error. We mea-

[1] The supply current was measured by measuring the voltage drop across a high precision small-valued resistor of a known resistance (0.02Ω). The current was then calculated by dividing the voltage by the resistance.

sured multiple runs of each workload; in general, we found the 95% confidence interval of the energy to be less than 0.7% of the mean energy. This implies that the runs were very repeatable, despite the possible variation that would arise from interactions between application threads, other processes and system daemons.

4.2 Workload

We used a varied workload to assess the performance of the different clock scaling algorithms. Since it's not clear what applications will be common on pocket computers, we used some obvious applications (web browsing, text reading) and other less obvious applications (chess, mpeg video and audio). The applications ran either directly on top of the Linux operating system or within a Java virtual machine [14]. To capture repeatable behavior for the interactive applications, we used a tracing mechanism that recorded timestamped input events and then allowed us to replay those events with millisecond accuracy. We did not trace the mpeg playback because there is no user interaction, and we found little inter-run variance. We used the following applications:

MPEG: We played a 320x200 color MPEG-1 video and audio clip at 15 frames a second. The mpeg video was rendered as a greyscale image on the Itsy. Audio was rendered by sending the audio stream as a WAV file to an audio player which ran as a separate process, forked from the video player. There is no explicit synchronization between the audio and video sequences, but both are sequenced to remain synchronized at 15 frames/second. The clip is 14 seconds and was played in a loop to provide 60 seconds of playback.

Web: We used a Javabean version of the IceWeb browser to view content stored on the itsy. We selected a file containing a stored article from `www.news.com` concerning the Itsy. We scrolled down the page, reading the full article. We then went back to the root menu and opened a file containing an HTML version of WRL technical report TN-56, which has many tables describing characteristics of power usage in Itsy components. The overall trace was 190 seconds of activity.

Chess: We used a Java interface to version 16.10 of the Crafty chess playing program. Crafty was run as a separate process. Crafty uses a play book for opening moves and then plays for specific periods of time in later stages of the games and plays the best move available when time expires. The 218 second trace includes a complete game of Crafty playing against a novice player (who lost, badly).

TalkingEditor: We used a version of the "mpedit" Java text editor that had been modified to read text files aloud using the DECtalk speech synthesis system (which is run in a separate process). The input trace records the user selecting a file to be opened using the file dialogue, (*i.e.* moving to the directory of the short text file and selecting the file), then having it spoken aloud and finally opening and having another text file read aloud. The trace took 70 seconds.

The Kaffe Java system [14] uses a JIT, makes extensive use of dynamic shared libraries and supports a threading model using setjmp/longjmp. The graphics library used by Java is a modified version of the publically available GRX graphics library and uses a polling I/O model to check for new input every 30 milliseconds. The MPEG player renders directly to the display.

4.3 Implementing the Scheduling Algorithms

We made two modifications to the Linux kernel to support our clock scheduling algorithms and data recording. The first modification provides a log of the process scheduler activity. This component is implemented as a kernel module with small code modifications to the scheduler that allow the logging to be turned on and off. For each scheduling decision, we record the process identifier of the process being scheduled, the time at which it was scheduled (with microsecond resolution) and the current clock rate.

We also implemented an extensible clock scaling policy module as a kernel module. We modified the clock interrupt handler to call the clock scheduling mechanism if it has been installed, and the Linux scheduler to keep track of CPU utilization. In Linux, the idle process always uses the zero process identifier. The idle process enters a low-power "nap" mode that stalls the processor pipeline until the next scheduling interval. If the previous process was not the idle process, the kernel adds the execution time to a running total. On every clock interrupt, this total is examined by the clock scaling module and then cleared. The CPU utilization can be calculated by comparing the time spent non-idle to the time length

of a quantum. Our time quantum was set to 10 msec, the default scheduling period in Linux; Pering et al. [5, 11] used similar values for their calculations.

Normally, a process can run for several quanta before the scheduler is called. The executing process is interrupted by the 100Hz system clock when the O/S decrements and examines a counter in the process control block at each interrupt. When that counter is zero, the scheduler is called. We set the counter to one each time we schedule a process, forcing the scheduler to be called every 10ms. While this modification adds overhead to the execution of an application, it allows us to control the clock scaling more rapidly. We measured the execution overhead and found it to be very small (about 6 microseconds for each 10ms interval, or 0.06%).

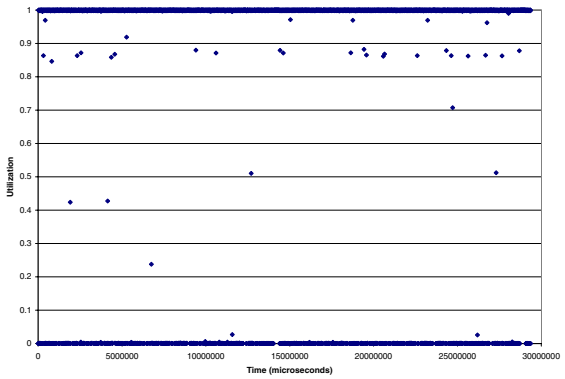
5 Results

The purpose of our study is to determine if the heuristics developed in prior studies can be practically applied to actual pocket computers. We examined a number of policies, most of which are variants of the $\texttt{AVG}_N$ policy. As described in §4.3, we used three different speed setting policies. Our intent was to focus on systems that could be implemented in an actual O/S and that did not require modifications to the applications (such as requiring information about deadlines or schedules). We assumed that our workloads had *inelastic constraints*; in other words, we assumed the applications had no way to accommodate "missed deadlines".

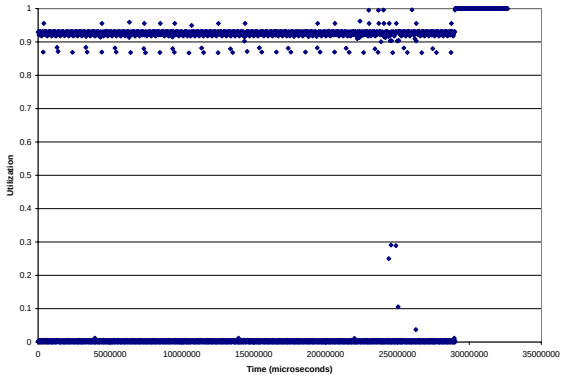
We split the discussion of our results into three parts. The first section describes aspects of the applications and how they differ from those used in prior work and the second section discusses the performance of the different clock scheduling algorithms. Finally, we examine the benefit of the limited voltage scaling available on the Itsy and summarize the results.

5.1 Application Characteristics

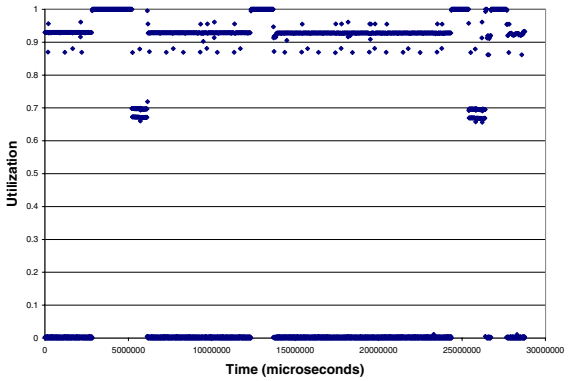
Figure 3 presents plots of the processor utilization over time for each of the benchmark applications. This information was gathered using the on-line process logging facility that we added to the kernel. Due to kernel memory limitations, we could only capture a subset of the process behavior. Each application was able to run at 132MHz and still meet any user interaction constraints



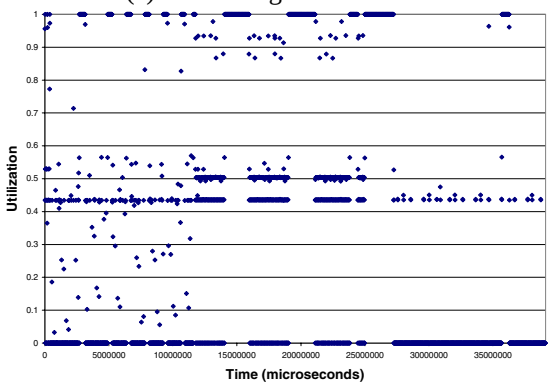
(a) MPEG Program at 206MHz



(b) Web Program at 206MHz



(c) Chess Program at 206MHz



(d) TalkingEditor Program at 206MHz

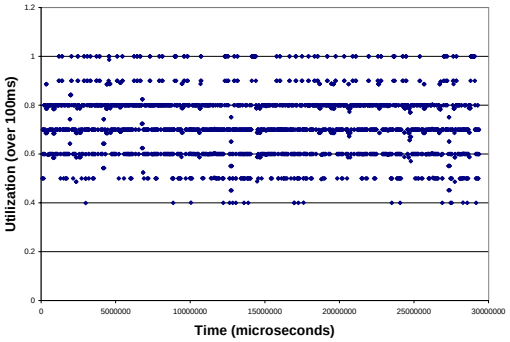
Figure 3: Utilization using 10ms Moving Average For Between 30 to 40 Second Intervals Using 206MHz Frequency Setting

(*i.e.* the application did not appear to behave any differently).

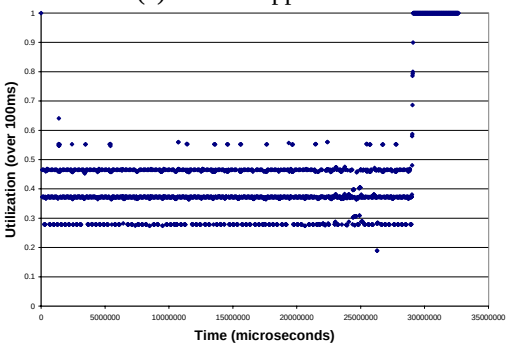
The utilization is computed for each 10ms scheduling quantum. We used the same 10ms interval for logging that is used for scheduling within Linux. Since most processes compute for several quanta before yielding, the system is usually either completely idle or completely busy during a given quantum. Some processes execute for only a short time then yield the processor prior to the end of their scheduling quanta; for example, the Java implementation we used has a 30ms I/O polling loop – thus, when the Java system is "idle," there is a constant polling action every 30ms that takes about a millisecond to complete.

The behavior of the applications is difficult to predict, even for applications that should have very predictable behavior and each application appears to run at a different time-scale. The MPEG application renders at 15 frames/sec; there are 450 frames in the 30 second interval shown in Figure 3. Each frame is rendered in 67ms or just under 7 scheduling quanta. Any scheduling mechanism attempting to use information from a single frame (as opposed to a single quanta) would need to examine at least 7 quanta. Other applications have much coarser behavior. For example, the TalkingEditor application consumes varying amount of CPU time until the text is being loaded for speech synthesis. The bursty behavior prior to the speech synthesis results from dragging images, JIT'ing applications and opening files. Following this are long bursts of computation as the text is actually synthesized and send to the OSS-compatible sound driver. Finally, more cycles are taken by the sound driver. Thus, this application is bursty at a higher level.

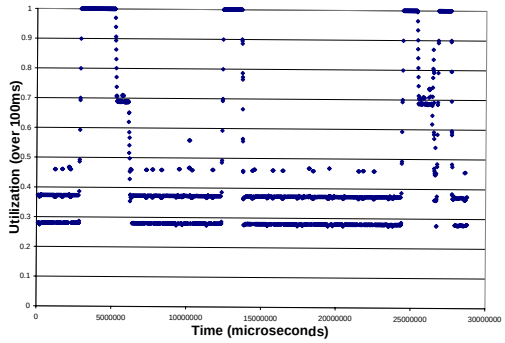
For most applications, patterns in the utilization are easier to see if you plot the utilization using a 100ms moving average, as shown in Figure 4. The MPEG application, in Figure 4(a), is still very sporadic because of inter-frame variation; for MPEG, there is even significant variance in CPU utilization (60-80%) when considering a 1 second moving average (not shown). The Chess and TalkingEditor applications show patterns influenced by user interaction. It's clear from Figure 4(c) that utilization is low when the user is thinking or making a move and that utilization reaches 100% when Crafty is planning moves. Likewise, Figure 4(d) shows the aforementioned pattern of synthesis and sound rendering.



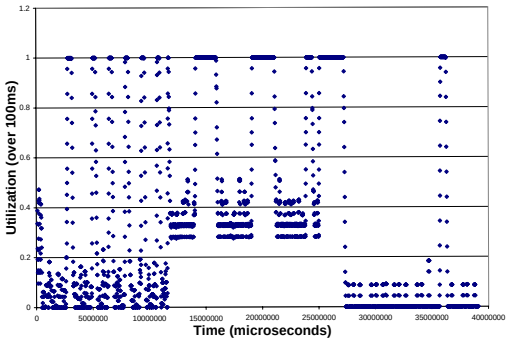
(a) MPEG Application



(b) Web Application



(c) Chess Application



(d) TalkingEditor Application

Figure 4: Utilization using 100ms Moving Average For Between 30 to 40 Second Intervals Using 206MHz Frequency Setting

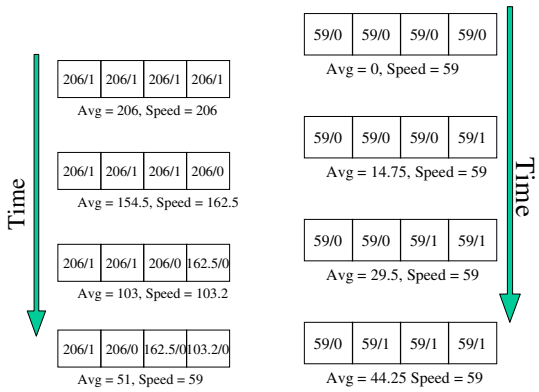
5.2 Clock Scheduling Comparison

The goal of a clock scheduling algorithm is to try to predict or recognize a CPU usage pattern and then set the CPU clock speed sufficiently high to meet the (predicted) needs of that application. Although patterns in the utilization are more evident when using a 100ms sliding average for utilization, we found that averaging over such a long period of time caused us to miss our "deadline". In other words, the MPEG audio and video became unsynchronized and some others applications such as the speech synthesis engine had noticeable delays. This occurs because it takes longer for the system to realize it is becoming busy.

This delay is the reason that the studies of Govil *et al.* [6] and Weiser [7] argued that clock adjustment should examine a 10-50ms interval when predicting future speed settings. However, as Figure 3 shows, it is difficult to find any discernible pattern at the smaller time-scales. Like Govil *et al.*, we also allowed speed setting to occur at any interval; Weiser *et al.* did not model having the scheduler interrupted while an application was running, but rather deferred clock speed changes to occur only when a process yielded or began executing in a quanta.

There are a number of possible speed-setting heuristics we could examine; since we were focusing on *implementable* policies, we primarily used the policies explored by Pering *et al.* [5]. We also explored other alternatives. One simple policy would determine the number of "busy" instructions during the previous N 10ms scheduling quanta and predict that activity in the next quanta would have the same percentage of busy cycles. The clock speed would then be set to insure enough busy cycles.

This policy sounds simple, but it results in exceptionally poor responsiveness, as illustrated in Figure 5. Figure 5(a) shows the speed changes that would occur when the application is moving from period of high CPU utilization to one of low utilization; the speed changes to 59MHz relatively quickly because we are adding in a large number of idle cycles each quanta. By comparison, when the application moves from an idle period to a fully utilized period, the simple speed setting policy makes very slow changes to the processor utilization and thus the processor speed increases very slowly. This occurs because the total number of non-idle instructions across the four scheduling intervals grows very slowly.



(a) Going to Idle (b) Speeding up

Figure 5: Simple averaging behavior results in poor policies. Each box represents a single scheduling interval, and the scheduling policy averages the number of non-idle instructions over the four scheduling quanta to select the minimum processor speed. To simplify the example, we assume each interval is either fully utilized or idle. The notation "206/0" means the CPU is set to 206MHz and the quanta is idle while "206/1" means the CPU is fully utilized.

5.3 The $\texttt{AVG}_N$ Scheduler

We had initially thought that a policy targeting the necessary number of non-idle cycles would result in good behavior, but the previous example highlights why we use the speed-setting policies described in §4.3. We used the same $\texttt{AVG}_N$ scheduler proposed by Govil [6] and Pering [5] and also examined by Pering *et al.* in [5]; Perings later paper in [11] did not examine scheduler heuristics and only used real-time scheduling with application-specified scheduling goals.

Our findings indicate that the $\texttt{AVG}_N$ algorithm can not settle on the clock speed that maximizes CPU utilization. Although a given set of parameters can result in optimal performance for a single application, these tuned parameters will probably not work for other applications, or even the same application with different input. The variance inherent in many deadline-based applications prevents an accurate assessment of the computational needs of an application. The $\texttt{AVG}_N$ policy can be easily designed to ensure that very few deadlines will be missed, but this results in minimal energy savings. We use an MPEG player as a running example in this section, as it best exemplifies behavior that illustrates the multitude of problems in past-based interval algorithms. Our intuition is that if there's is a single application that illustrates simple, easy-to-predict behavior, it should be

MPEG. Our measurements showed that the MPEG application can run at 132MHz without dropping frames and still maintain synchronization between the audio and video. An ideal clock scheduling policy would therefore target a speed of 132MHz.

However, without information from the user level application, a kernel cannot accurately determine what deadlines an application operates under. First, an application may have different deadline requirements depending on its input; for example, an MPEG player displaying a movie at 30fps has a shorter deadline than one running at 15fps. Although the deadlines for an application with a given input may be regular, the computation required in each deadline interval can vary widely. Again, MPEG players demonstrate this behavior; I-frames (key or reference) require much more computation than P-frames (predicted), and do not necessarily occur at predictable intervals.

One method of dealing with this variance is to look at lengthy intervals which will, by averaging, reduce the variance of the computational observations. Our utilization plots showed that even using 100ms intervals, significant variance is exhibited. In addition to interval length, the number of intervals over which we average (N) of the $\texttt{AVG}_N$ policy can also be manipulated. We conducted a comprehensive study and varied the value of N from 0 (the `PAST` policy) to 10 with each combination of the speed-setting policies (*i.e.* using "`peg`" to set the CPU speed to the highest point, or "`one`" to increment or decrement the speed).

Our conclusions from the results with our benchmarks is that the weighted average has undesirable behavior. The number of intervals not only represents the length of interval to be considered; it also represents the lag before the system responds, much like the simple averaging example described above. Unlike that simple policy, once $\texttt{AVG}_N$ starts responding, it will do so quickly. For example, consider a system using an $\texttt{AVG}_9$ mechanism with an upper boundary of 70% utilization and "`one`" as the algorithms used to increment or decrement the clock speed. Starting from an idle state, the clock will not scale to 206MHz for 120 ms (12 quanta). Once it scales up, the system will continue to do so (as the average utilization will remain above 70%) unless the next quantum is partially idle. This occurs because the previous history is still considered with equal weight even when the system is running at a new clock value.

The boundary conditions used by Pering in [5] result in a system that scales more rapidly down than up. Table 1 illustrates how this occurs. If the weighted average is

Time(ms)	Idle/Active	AVG<9>	Notes
10	Active	1000	
20	Active	1900	
30	Active	2710	
40	Active	3439	
50	Active	4095	
60	Active	4685	
70	Active	5217	
80	Active	5965	
90	Active	6125	
100	Active	6513	
110	Active	6861	
120	Active	7175	Scale up
130	Active	7458	Scale up
140	Active	7712	Scale up
150	Active	7941	Scale up
160	Idle	7146	Scale up
170	Idle	6432	
180	Idle	5789	
190	Idle	5210	
200	Idle	4689	Scale down

Table 1: Scheduling Actions for the $\texttt{AVG}_9$ Policy

70%, a fully active quantum will only increase the average to 73% while a fully idle quantum will reduce it to 63% – thus, there is a tendency to reduce the processor speed.

The job of the scheduler is made even more difficult by applications that attempt to make their own scheduling decisions. For example, the default MPEG player in the Itsy software distribution uses a heuristic to decide whether it should sleep before computing the next frame. If the rendering of a frame completes and the time until that frame is needed is less than 12ms, the player enters a spin loop; if it is greater than 12ms, the player relinquishes the processor by sleeping. Therefore, if the player is well ahead of schedule, it will show significant idle times; once the clock is scaled close to the optimal value to complete the necessary work, the work seemingly increases. The kernel has no method of determining that this is wasteful work.

Furthermore, there is some mathematical justification for our assertion that $\texttt{AVG}_N$ fundamentally exhibits undesirable behavior, and will not stabilize on an optimal clock speed, even for simple and predictable workloads. Our analysis only examines the "smoothing" portion of $\texttt{AVG}_N$, not the clock setting policy. Nevertheless, it works well enough to highlight the instability issues with $\texttt{AVG}_N$ by showing that, even if the system is started out at the ideal clock speed, $\texttt{AVG}_N$ smoothing will still result in undesirable oscillation.

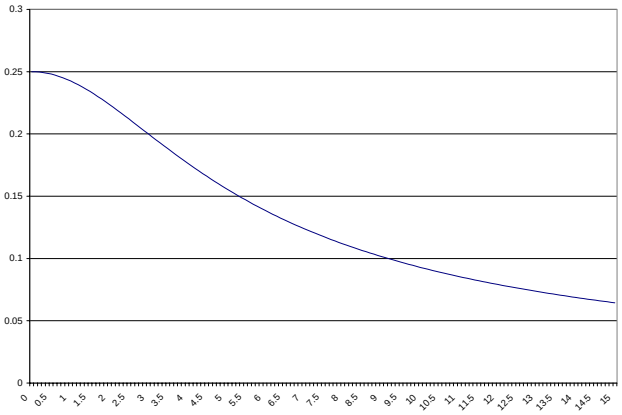


Figure 6: Fourier Transform of a Decaying Exponential

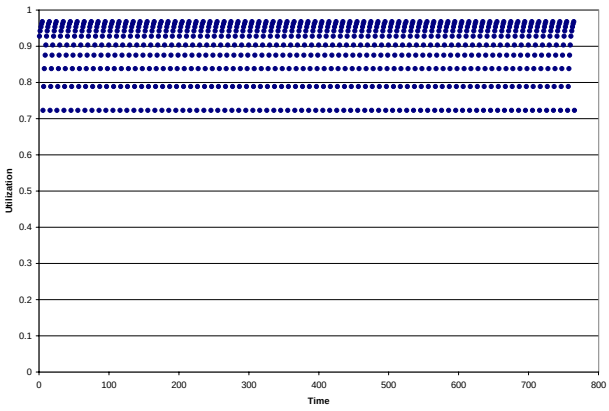


Figure 7: Result of $\texttt{AVG}_3$ Filtering on a the Processor Utilization for a Periodic Workload Over Time

A processor workload over time may be treated as a mathematical function, taking on a value of 1 when the processor is busy, and 0 when idling. Borrowing techniques from signal processing allows us to characterize the effect of $\texttt{AVG}_N$ on workloads in general as well as specific instances. $\texttt{AVG}_N$ filters its input using a decaying exponential weighting function. For our implementation, we used a recursive definition in terms of both the previous actual (U_{t-1}) and weighted (W_{t-1}) utilizations: $W_t = \frac{N \times W_{t-1} + U_{t-1}}{N+1}$. For the analysis, however, it is useful to transform this into a less computationally practical representation, purely in terms of earlier unweighted utilizations. By recursively expanding the W_{t-1} term and performing a bit of algebra, this representation emerges: $W_t = \frac{1}{N+1} \sum_{k=0}^{t-1} (\frac{N}{N+1})^{k-(t-1)} U_k$. This equation explicitly shows the dependency of each W_t on all previous U_t, and makes it more evident that the weighted output may also be expressed as the result of discretely convolving a decaying exponential function with the raw input. This allows us to examine specific types of workloads by artificially generating a representative workload and then numerically convolving the weighting function with it. We can also get a qualitative feel for the general effects $\texttt{AVG}_N$ has by moving to continuous space and looking at the Fourier transform of a decaying exponential, since convolving two functions in the time domain is equivalent to multiplying their corresponding Fourier transforms.

Lets begin by examining the Fourier transform of a decaying exponential: $x(t) = e^{-\alpha t} u(t)$, where $u(t)$ is the unit step function, 0 for all $t < 0$ and 1 for $t \geq 0$. This captures the general shape of the $\texttt{AVG}_N$ weighting function, shown in Figure 6. Its Fourier transform is $X(\omega) = \frac{1}{i\omega + \alpha}$. The transform attenuates, but does not eliminate, higher frequency elements. If the input signal oscillates, the output will oscillate as well. As α gets smaller the higher frequencies are attenuated to a greater degree, but this corresponds to picking a larger value for N in $\texttt{AVG}_N$ and comes at the expense of greater lag in response to changing processor load.

For a specific workload example, we'll use a simple repeating rectangle wave, busy for 9 cycles, and then idle for 1 cycle. This is an idealized version of our MPEG player running roughly at an optimal speed, i.e. just idle enough to indicate that the system isn't saturated. Ideally, a policy should be stable when it has the system running at an optimal speed. This implies that the weighted utilization should remain in a range that would prevent the processor speed from changing. However, as was fore-shadowed by our initial qualitative discussion, this is not the case. A rectangular wave has many high frequency components, and these result in a processor utilization as shown in Figure 7. This figure shows the oscillation for this example, and shows that oscillation occurs over a surprisingly wide range of the processor utilization. As discussed earlier, our experimental results with the MPEG player on the Itsy also exhibit this oscillation because that application exhibits the same step-function resource demands exhibited by our example.

We also simulated interval-based averaging policies that used a pure average rather than an exponentially decaying weighting function, but our simulations indicated that that policy would perform no better than the weighted averaging policy. Simple averaging suffers from the same problems experienced by the weighted averaging if you do not average the appropriate period.

5.4 Summary of Results

We are omitting a detailed exposition on the scheduling behavior of each scheduling policy primarily because most of them resulted in equivalent (and poor) behavior. Recall that the best possible scheduling goal for MPEG would be to switch to a 132MHz speed and continue to render all the frames at that speed. **No heuristic policy that we examined achieved this goal**. Figure 8 shows the clock setting behavior of the best policy we found. That policy uses the PAST heuristic (*i.e.* $\texttt{AVG}_0$) and "pegs" the CPU speed either to 206MHz or 59MHz depending on the weight metric. The bounds on the hysteresis where that a CPU utilization greater than 98% would cause the CPU to increase the clock speed and a CPU utilization less than 93% would decrease the clock speed.

This policy is "best" because it never misses any deadline (across all the applications) and it also saves a small but significant amount of energy. This last point is illustrated in Table 2. This table shows the 95% confidence interval for the average energy needed to run the MPEG application. The reduction in energy between 206MHz and 132MHz occurs because the application wastes fewer cycles in the application idle loop used to meet the frame delays for the MPEG clip. A $\approx 8\%$ energy reduction occurs when we drop the processor voltage to 1.23V – this is less than the 15% maximum reduction we measured because the application uses resources (*e.g.* audio) that are not affected by voltage scaling.

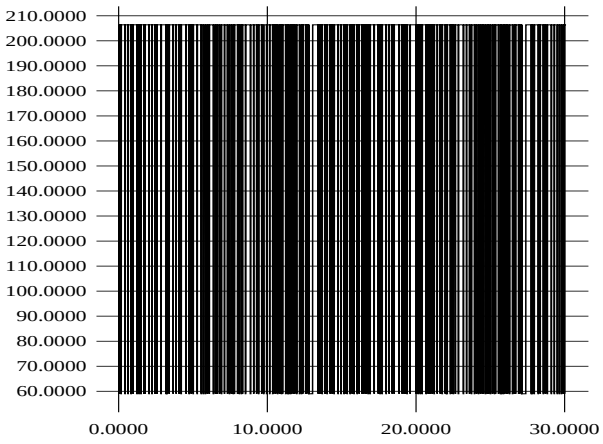


Figure 8: Clock frequency for the MPEG application using the best scheduling policy from our empirical study – the scheduling policy only select 59Mhz or 206MHz clock settings and changes clock settings frequently. This scheduling policy results in suboptimal energy savings but avoids noticeable application slowdown.

The PAST policy we described results in a small but statistically significant reduction in energy for the MPEG application. Allowing the processor to scale the voltage when the clock speed drops below 162.2MHz results in no statistical decrease.

We initially surmised that there is no improvement because the cost of voltage and clock scaling on our platform out-weighs any gains. We measured the cost of clock and voltage scaling using the DAQ. To measure clock scaling, we coded a tight loop that switched the processor clock as quickly as possible.

Before each clock change, we inverted the state of a specific GPIO and used the DAQ to measure the interval with high precision. We took measurements when the clock changed across many different clock settings (*e.g.* from 59 to 206MHz, from 191 to 206MHz and so on).

Clock scaling took approximately 200microseconds, independent of the starting or target speed. During that time, the processor can not execute instructions. Thus, frequency changing varies between $11,200$ clock periods at 59MHz and $40,000$ clock periods at 200MHz.

We measured the time for the voltage to settle following a voltage change. It takes ≈ 250 microseconds to reduce voltage from $1.5V$ to $1.23V$; in fact, the voltage slowly reduces, drops below $1.23V$ and then rapidly settles on $1.23V$. Voltage increases were effectively instantaneous. We suspect the slow decay occurs because of capacitance; many processors use external decoupling capacitors to provide sufficient current sourcing for processors that have widely varying current demands.

These measurements indicate that the time needed for clock and voltage changes are less than 2% of the scheduling interval; thus, we would be able to change the clock or voltage on every scheduling decision with less than 2% overhead. The fact that we see little energy reduction is related to the limited energy savings possible with the voltage scaling available on this platform and the efficacy of the policies we explored.

6 Conclusions and Future Work

Our implementation results were disappointing to us – we had hoped to be able to identify a prediction heuristic that resulted in significant energy savings, and we thought that the claims made by previous studies would be born out by experimentation. Although we have

Algorithm	Energy
Constant Speed @ 206.4 MHz, 1.5 Volts	85.59 - 86.49
Constant Speed @ 132.7 MHz, 1.5 Volts	79.59 - 80.94
Constant Speed @ 132.7 MHz, 1.23 Volts	73.76 - 74.41
PAST, Peg - Peg, Thresholds: $> 98\%$ scales up, $< 93\%$ scales down, 1.5 Volts	85.03 - 85.47
PAST, Peg - Peg, Thresholds: $> 98\%$ scales up, $< 93\%$ scales down, Voltage Scaling @ 162.2 MHz	84.60 - 85.45

Table 2: Summary of Performance of Best Clock Scaling Algorithms

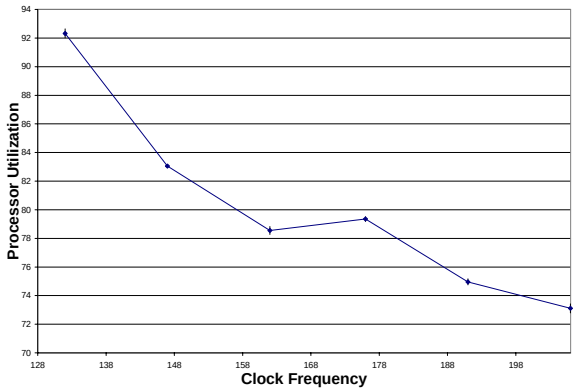


Figure 9: Non-linear change in Utilization with Clock Frequency (in MHz)

Processor Freq.	Cycles/Mem. Reference	Cycles / Cache Reference
59.0	11	39
73.7	11	39
88.5	11	39
103.2	11	39
118.0	13	41
132.7	14	42
147.5	14	49
162.2	15	50
176.9	18	60
191.7	19	61
206.4	20	69

Table 3: Memory access time in cycles for reading individual words as well as full cache lines.

found a policy that saves some energy, that policy leaves much to be desired. The policy causes many voltage and clock changes, which may incurr unnecessary overhead; this will be less of a problem as processors are better designed to accommodate those changes. However, the policy did result in both the most responsive system behavior and most significant energy reduction of all the policies we examined.

As with all empirical studies, there are anomalies in our system that we can not explain and that may have influenced our results. We found that the processor utilization does not always vary linearly with clock frequency. Figure 9 shows the processor utilization *vs.* clock frequency for the MPEG benchmark. There is a distinct "plateau" between 162MHz and 176.9MHz. We believe that this delay may be induced by the varying number of clock cycles needed for memory accesses as the processor frequency changes, as shown in Table 3. That table shows the memory access time for EDODRAM for reading individual words or a full cache line; there is an obvious non-linear increase between 162MHz and 176.9MHz. The potential speed mismatch between processor and memory has been noted by others [12], but we have not devised a way to verify that this is the only factor causing the non-linear behavior we noted.

This paper is the first step on an effort to provide robust support for voltage and clock scheduling within the Linux operating system. Although our initial results are disappointing, we feel that they serve to stop us from attempting to devise clever heuristics that could be used for clock scheduling. It may well be that Pering [11] reached a similar conclusion since their later publications discontinued the use of hueristics, but their publications don't describe the implementation of their operating system design or the rational behind the policies used. Furthermore, they don't describe how deadlines are to be "synthesized" for applications such as Web, TalkingEditor and Web where there is no clear "deadline".

Our immediate future work is to provide "deadline" mechanisms in Linux. These deadlines are not precisely the same mechanism needed in a true real-time O/S – in a RTOS, the application does not care if the deadline is reached early, while energy scheduling would prefer for the deadline to be met as late as possible. A further challenge we face will be to find a way to automatically synthesize those deadlines for complex applications.

7 Acknowledgments

We would like to thank the members of Compaq's Palo Alto Research Labs who created the Itsy Pocket Computer and built the infrastructure that made this work possible. We would also like to thank Wayne Mack for adapting the Itsy to fit our needs.

This work was supported in part by NSF grant CCR-9988548, a DARPA contract, and an equipment donation from Compaq Computer.

References

[1] Jay Heeb. The next generation of strongarm. In *Embedded Processor Forum*. MDR, May 1999.

[2] Intel Corporation. Moblie pentium iii processor in bga2 and micro-pga2 packages. Datasheet Order #245302-002, 2000.

[3] David Linden (editor). *Handbook of Batteries, 2nd ed.* McGraw-Hill, New York, 1995.

[4] Carla-Fabiana Chiasserini and Ramesh R. Rao. Pulsed Battery Discharge in Communication Devices. In *The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 88–95, August 1999.

[5] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation of dynamic voltage scaling algorithms. In *IEEE Symposium on Low Power Electronics*. IEEE Symposium on Low Power Electronics, 1995.

[6] Kinshuk Govil, Edwin Chan, , and Hal Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *Proceedings of The First ACM International Conference on Mobile Computing and Networking*, Berkeley, CA, November 1995.

[7] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.

[8] Trevor Pering. Private communication.

[9] J. Montanaro and *et. al*. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *Digital Technical Journal*, volume 9. Digital Equipment Corporation, 1997.

[10] Dan Dobberpuhl. The Design of a High Performance Low Power Microprocessor. In *International Symposium on Low Power Electronics and Design*, pages 11–16, August 1996.

[11] Trevor Pering, Tom Burd, and Robert Brodersen. Voltage scheduling in the lparm microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Design*, August 2000.

[12] Thomas L. Martin. *Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computers*. PhD thesis, Carnegie Mellon University, 1999.

[13] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of the ACM SIGMETRICS '00 International Conference on Measurement and Modeling of Computer Systems*, 2000. (to appear).

[14] Transvirtual Technologies Inc. Kaffe Java Virtual Machine. `http://www.transvirtual.com`.