# $\mu$Sleep: A Technique for Reducing Energy Consumption in Handheld Devices

Lawrence S. Brakmo
DoCoMo USA Labs
181 Metro Drive, Suite 300
San Jose, CA 95110
brakmo@docomolabs-usa.com

Deborah A. Wallach
Google
1600 Ampitheatre Parkway
Mountain View, CA 94043
kerr@google.com

Marc A. Viredaz
Logitech
Switzerland
viredaz@computer.org

## ABSTRACT

Energy management has become one of the great challenges in portable computing. This is the result of the increasing energy requirements of modern portable devices without a corresponding increase in battery technology. *$\mu$Sleep* is an energy reduction technique for handheld devices that is most effective when the handheld's processor is lightly loaded, such as when the user is reading a document or looking at a web page. When possible, rather than using the processor's idle mode, $\mu$Sleep tries to put the processor in sleep mode for short periods (less than one second) without affecting the user's experience. To enhance the perception that the system is on, an image is maintained on the display and activity is resumed as a result of external events such as touch-screen and button activity. We have implemented, analyzed and evaluated $\mu$Sleep on a prototype pocket computer, where it has reduced energy consumption by up to 60%.

## Categories and Subject Descriptors

C.5 [**Computer System Implementation**]: General

## General Terms

Design, Measurement, Performance

## Keywords

energy management, power management, processor sleep

## 1. INTRODUCTION

The energy requirements of modern portable computing devices continue to increase as a result of various factors. Among these factors are the use of more powerful processors, the inclusion of more functionality, such as wireless networking and imaging capabilities, as well as an increase in their usage time as they displace other pieces of equipment.

These increases in energy requirements have been partially offset by advances in battery technology and advances in low power electronics. However, these advances have not been sufficient to satisfy the users' continual requests for longer battery life. As a result, energy management has become an integral part in the design of portable computing devices.

Most portable computing devices, and in particular handheld computers, differ from non-portable computing devices in some important ways. For example, the processors used in these systems are highly integrated and include a large number of I/O components such as LCD controllers, serial communication interfaces, etc. Another important difference is their usage patterns. Most handheld devices are used interactively and, as a result, there is a lot of idle time between the user interactions, usually as a result of the user viewing or reading the result of the previous interaction.

We have implemented a technique, called *$\mu$Sleep* (pronounced micro-sleep), that takes advantage of this usage pattern. Rather than always putting the processor in its idle mode during short (less than a second) periods of inactivity, we put the processor in its sleep mode whenever possible. The sleep state we use differs from the usual system sleep state in important ways. The first difference consists of keeping the display on while showing the image that was present before the system went to sleep. The second difference consists of waking the system up before the next operating system (OS) scheduled event, such as servicing a kernel timer. The final difference consists of waking the system up when an external event, such as a press on the touch screen, occurs. The goal of these differences is to make the user unaware that the processor is sleeping. As far as the user is concerned, the system is just idle. The display is on, and the system responds to system and user events as usual. The only difference is the added latency when waking up as a result of external events, such as touchscreen or button events. However, this latency is unlikely to be noticed by the user since the worst case delay for the processor to awaken and resume executing is less than 12 ms, well below the perceptual threshold.

Although the use of short duration sleeps is not new, our work differs from earlier work in important ways. These differences are described in Section 2. In order to evaluate $\mu$Sleep we have done a full implementation of this technique on *Itsy*, a prototype pocket computer developed by our team at the former Compaq Laboratories in Palo Alto, California. Section 3 contains a detailed description of Itsy, followed by a description of $\mu$Sleep in Section 4. Section 5 describes our evaluation infrastructure, which allows us to measure energy consumption while we replay interactive scenarios. The results of our evaluation experiments are given in Section 6, which show that energy consumption can be reduced by up to 60%. Section 7 discusses potential improvements to $\mu$Sleep and our experience implementing $\mu$Sleep on a Pocket PC device.

## 2. RELATED WORK

Most recent processors targeted at battery-powered electronics feature low-power modes [3, 16, 8]. There is often a mode aimed at implementing the idle thread of a typical OS and another mode which is intended to be used for longer periods of inactivity (e.g., when the device is "turned off"). The former is often referred to as *idle* or *doze mode*, while the latter is often called *sleep mode*. Other names have also been used. However, for the sake of clarity, we only use the terms "idle" and "sleep" in this paper. Entering and exiting idle mode are usually lightweight operations in terms of time and energy, but the power savings realized in idle mode are moderately important. On the other hand, much more power can be saved in sleep mode, at the cost of a larger time and energy overhead to enter and exit this mode. Many processors offer several flavors of idle and sleep modes, each with different tradeoffs between power savings and overhead.

Our work explores how, under certain conditions, sleep mode can be used instead of idle mode. One approach is to wait until the processor is idle and then turn it off when it is predicted that the processor will not be needed for a time interval long enough for the shutdown to be worthwhile. This prediction is generally made on the basis of past observations [15], under the assumption that the time when the next event requiring the processor will occur is unknown. In some cases, the end of the shutdown time can be predicted, so that the processor can be woken up in time for the next (predicted) event [6]. The task of predicting when to shut off a processor offers similarities with that of spinning down a hard disk [10, 1, 14], however the time scales are very different.

Our work differs from these earlier uses of short duration sleeps in the following ways. M.B. Srivastava et al.'s work [15] focuses on devices where all computation is driven by I/O events, such as portable wireless terminals. Furthermore, they don't implement their techniques on a real system; their results are based on analysis and modeling. C.-H. Hwang's work [6] focuses solely on mechanisms for predicting the length of idle periods on event driven applications (X Window System server, Netscape, telnet, and tin). J.R. Lorch and A.J. Smith's work [11] is based on processors that can be put to sleep by turning off the clock signal, which incurs very little latency, and which preserve most of their state when they are sleeping.[1]

In comparison, our work focuses on a handheld device, the Itsy pocket computer, where computation is driven by both internal events as well as external events. This device, when entering and exiting the sleep mode, incurs high overhead and latency — about 16 ms to go from running to sleeping and back to running. Most importantly, we have implemented $\mu$Sleep on a real device which allowed us not only to measure the performance of the technique, but also to gain deep insight into the issues surrounding this technique.

The main concept of $\mu$Sleep, keeping the display on and waking up before the next OS scheduled event, was first published by Kamijoh et al. [9] in 2001 as part of their work with the IBM wristwatch computer. Our work differs from theirs in the following important ways. On the implementation side, we are able to exploit much shorter duration sleeps than they do (250 ms compared to 30 ms). Regarding the analysis of the technique, we discuss the energy overheads of going into and out of sleep and how they must be taken into account when deciding whether to sleep. We also present equations and techniques that can be used to predict the energy savings as a result of using $\mu$Sleep on any device. Finally, we do an experimental evaluation of $\mu$Sleep where we show the energy saved by the technique in three different scenarios, two of which consist of interactive applications. We evaluate $\mu$Sleep with the processor running at twelve different frequencies and also measure the differences in response time for the interactive applications. On the other hand, Kamijoh et al. do not include any energy or power measurements of their device while using the technique.

A related promising technique is *dynamic voltage-frequency scaling*. Since the power used by a CMOS circuit is proportional to the product $f \cdot v^2$, where $f$ is the clock frequency and $v$ is the voltage, reducing the voltage — and hence the maximum clock frequency — provides an important benefit. Even if the time required to complete a given task increases inversely proportionally to the frequency, the energy required for this task decreases at the same rate as the voltage squared. Policies of when to to modify the voltage and frequency have been extensively studied [18, 4, 12, 2, 13].

This work explores how to save power when the processor completes its tasks faster than necessary. It should be noted that dynamic voltage-frequency scaling algorithms could be used at the same time as $\mu$Sleep. However, each technique would affect the potential savings of the other and finding an optimum could be a very hard problem. One potential way to make use of both techniques would be to use only one at a time, picking whichever performs best. In particular, dynamic voltage-frequency scaling could be used when $\mu$Sleep can not be used.

## 3. EVALUATION PLATFORM

For an evaluation platform, we used the Itsy pocket computer [5]. Itsy version 2 is a complete handheld device based on the StrongARM SA-1100 processor [7] with 32 Mbyte of flash memory, 32 Mbyte of DRAM, a $320 \times 200$-pixel grayscale LCD, a touch screen, audio input and output, a rechargeable lithium-ion battery, and several serial interfaces. Itsy can also accommodate a daughter-card which can be used to add more memory and/or interface additional peripherals.

In many aspects, Itsy is very similar to contemporary commercial handheld devices. Its size and weight ($118 \times 65 \times 16\,\text{mm}^3$, 130 g) makes it even smaller and lighter than most of its commercial counterparts. However, the most important difference is that Itsy was designed as a research plat-

[1]This state is actually very similar to the idle mode on our system, which is implemented by gating off the processor core's clock.

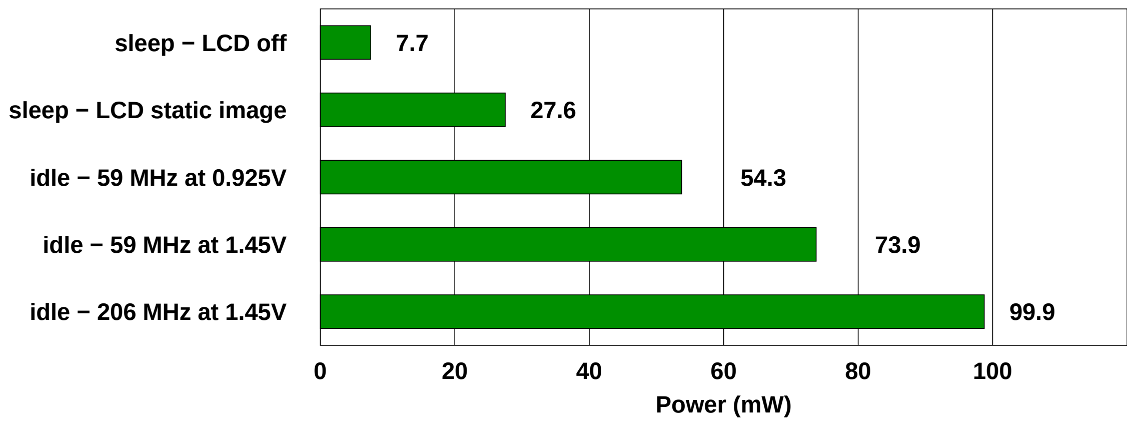


**Figure 1: Itsy power consumption in sleep and idle modes.**

form, with flexibility being the most important driving force behind all design decisions. Many features have been added to allow easy monitoring and modification of the hardware, as well as to avoid constraining how the software manages the hardware.

The StrongARM SA-1100 is a low-power 32-bit processor which implements the ARM instruction set. It provides a useful collection of peripheral devices, as well as power-saving features. In particular, it features software-controllable clock frequency and two low-power modes: idle and sleep. In idle mode, the clock to the processor core is gated off (saving power thanks to the circuit's CMOS technology), but the power is maintained and all peripherals remain enabled. In sleep mode, most of the processor is unpowered. Only the real-time clock and the wake-up circuit remain enabled. Optionally, the system clock can remain enabled for faster wakeup.

Several of Itsy's external peripherals (i.e., not integrated in the processor) can be disabled or offer low-power modes. To avoid constraining the software, each unit can be controlled individually and independently of whether the processor is in sleep mode or not. This strategy lets the OS disable any of these units while the processor is running, or conversely, if possible, any of the units can remain active while the processor is in sleep mode. This is particularly useful for the LCD, the touch screen, and the push-buttons. However, in the case of the display, some additional hardware was required because the LCD controller is integrated in the StrongARM SA-1100 processor. Itsy's LCD has a built-in 1-bit-per-pixel memory. With the help of an auxiliary controller, which is implemented in a programmable logic device and generates the appropriate timing signals, it is possible possible to display a static monochrome image (i.e., no gray levels) while the processor is in sleep mode.

Similarly, the DRAM can be kept in self-refresh mode during sleep or can be completely unpowered (although the latter is not used in this particular study). Therefore, the OS can implement a wide variety of sleep modes, ranging from "deep sleep," which maintains only the real-time clock, to "light sleep," which keeps all clocks on, the DRAM contents preserved, the LCD enabled, and most interrupts (e.g., touch screen, push-buttons) configured to wake up the processor.

Itsy's StrongARM SA-1100 processor supports *frequency scaling*, that is, the processor's frequency can be selected by software. However, this processor is not specified to operate at different voltages depending on the frequency, a property known as *voltage scaling*. At the time that Itsy was developed, there were no processors targeted to handheld devices that were specified for voltage scaling. However, processors supporting voltage scaling, like the Intel XScale family [8], had already been announced. In order to study energy management techniques involving voltage-frequency scaling, our team decided to perform our own characterization of the voltage-frequency behavior of the StrongARM SA-1100 processor.

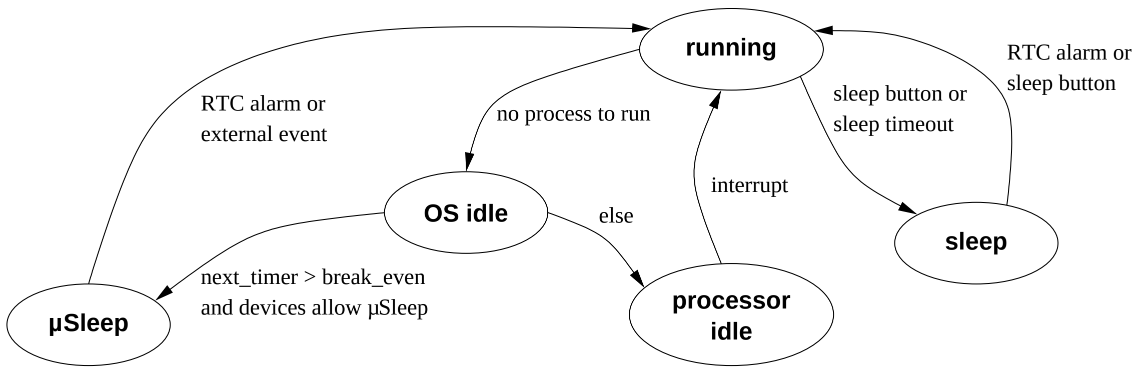
A strenuous set of benchmarks (including booting the OS and executing several power-hungry applications) was run on Itsy while decreasing the core voltage at the end of each set (for these experiments the processor core was powered by a laboratory power supply instead of the built-in one). Using a fully-automated set-up, several tens of Itsy systems were characterized.

Allowing for a reasonable margin, the voltage-frequency characteristics of each Itsy was established, with voltages below specifications at low frequencies and voltages slightly above specifications for above-specifications frequencies. Itsy units with "good" characteristics were then selected and modified to support voltage scaling. It should be stressed that these prototypes are research platforms intended to explore techniques targeted at next-generation processors. Therefore, the fact that they do not meet the level of reliability expected from a commercial device was not considered as a problem. In particular, all tests were performed at room temperature and no attempt was made to characterize the processor over its intended temperature range.

In this study we used an Itsy variant, referred to as *Itsy v2.6*, which is an *Itsy v2.4* system with a modified core power supply implemented on a daughter-card. Itsy v2.6 supports 30 voltages between 0.925 V and 2.0 V, allowing frequencies from 59 MHz to 265 MHz.

## 4. $\mu$SLEEP

$\mu$Sleep reduces energy consumption by taking advantage of idle periods, which are common with handheld computers. Rather than putting the processor in idle mode, $\mu$Sleep puts the processor in sleep mode for short durations, on the order of 40 ms to 1 second. Figure 1 shows the power consumed



running
RTC alarm or external event
no process to run
sleep button or sleep timeout
RTC alarm or sleep button
OS idle
else
interrupt
sleep
next_timer > break_even and devices allow µSleep
µSleep
processor idle

**Figure 2: State diagram of a generic implementation of $\mu$Sleep.**

by Itsy in sleep and idle modes. At 59 MHz, Figure 1 shows the idle power both for a core voltage of 0.925 V and of 1.45 V. These numbers are representative of a system with and without *voltage* scaling, respectively. The idle power at 206 MHz would be representative of a system without *frequency* scaling. As can be seen, there is a significant power difference between sleep mode with the LCD on and idle mode in the different configurations shown. Our goal was to create a new power state that would bridge this gap.

Although $\mu$Sleep puts the processor in sleep mode, it differs from the common implementation of the system sleep state, where the device appears as if it was turned off. In contrast, $\mu$Sleep attempts to fool the user into believing that the device is running as usual. In other words, the two goals of $\mu$Sleep are: (1) to have no effect on the user's experience and (2) to reduce the device's energy consumption.

The first goal of having no effect on the user's experience imposes some requirements. First, the system must mimic the physical appearance of a running device. This is achieved by keeping all active peripherals working as on a running system, in particular, the display must maintain the same image as just before the processor entered sleep mode. Second, the system must mimic the same computational behavior as if it was running. This is achieved in two ways: (1) by only putting the processor in sleep mode when it is not needed (i.e., the OS is idle and none of the active peripherals require the processor to be running[2]) and (2) by waking up the device just before the next OS scheduled event, or when the user interacts with the device (touch-screen or button press). By waking up the processor just before the next OS scheduled event, the device behaves just like a system that is continuously awake. For example, if an application has a blinking cursor implemented in software, the cursor will continue to blink because the system will be woken up every time the cursor needs to change appearance.

The goal of reducing energy consumption is achieved by preventing the processor from going into sleep mode, unless it has been determined that it will sleep long enough to save energy. On Itsy, entering sleep mode and exiting it immediately after consumes more energy than keeping the system idle for the same duration. This is due to the hardware and software overhead of entering and exiting sleep mode (flushing the caches, executing suspend and resume code, etc.).

[2]For instance, on Itsy, some peripherals like audio input and output require the processor to be running and, therefore, $\mu$Sleep can not be used if one of these peripherals is active.

## 4.1 Implementing $\mu$Sleep

There are specific hardware and software requirements that need to be satisfied in order to be able to implement $\mu$Sleep. There are four hardware requirements. First, the processor must have a sleep mode. This is common for modern processors targeted at low-power applications. Second, the device must be able to display a static image while the processor is asleep. This can be achieved in many ways; for example, if the processor is able to keep an integrated LCD controller active while the rest of the processor goes to sleep, by having the LCD controller outside the processor, or by using an LCD display with a built in frame-buffer and which is able to display the image in the frame-buffer when the LCD controller is turned off. Third, the system must have the capability of waking up in response to external events such as touch-screen or button activity. Finally, the device must have a programmable timer, ideally with 1 to 10 ms resolution, which can wake the system up. This timer is used to wake the system up before the next OS scheduled event.

The software requirements are as follows. First, the code implementing $\mu$Sleep needs to know when the next OS timer event[3] will be, to determine if the system can sleep long enough to save energy. Second, the code needs to know if currently active peripherals will allow the processor to enter its sleep mode. For example, on Itsy any audio input or output activity requires the processor to stay awake. Finally, there must be a way for the code implementing $\mu$Sleep to put the system to sleep, which includes notifying device drivers before the system goes to sleep and just after it wakes up. These requirements are easily satisfied if implementing $\mu$Sleep on a device with an open operating system (we used Linux).

Figure 2 shows the system states in a generic implementation of $\mu$Sleep (some details have been left out for simplicity). In this sample implementation, the real time clock (RTC) is used as the $\mu$Sleep wake up timer. The primary state is when the system is **running**; that is, a process or thread is running, or kernel code is executing on the behalf of a process or thread. The system can enter its **sleep** state as a result of a given button being pressed or as a result of an inactivity timer. The **OS idle** state is entered when the OS idle process starts running, as a result of no other

[3]An OS timer event is a software timer set by the OS; the OS uses a periodic hardware timer to implement the functionality of the software timer.

process or thread being able to run. When the system enters the **OS idle** state it checks whether it can do a short duration sleep ($\mu$Sleep). The check consists of determining if the system can sleep long enough to save energy and if all the devices allow the processor to enter sleep mode. If so, the system enters the **$\mu$Sleep** state, otherwise the system goes into the **processor idle** state where the processor is put in idle mode to conserve energy. The system exits the **processor idle** state and goes into the **running** state as a result of any enabled interrupt. The system exits the **$\mu$Sleep** state as a result of an enabled external event, such as touch-screen or button activity, or as a result of the RTC alarm.

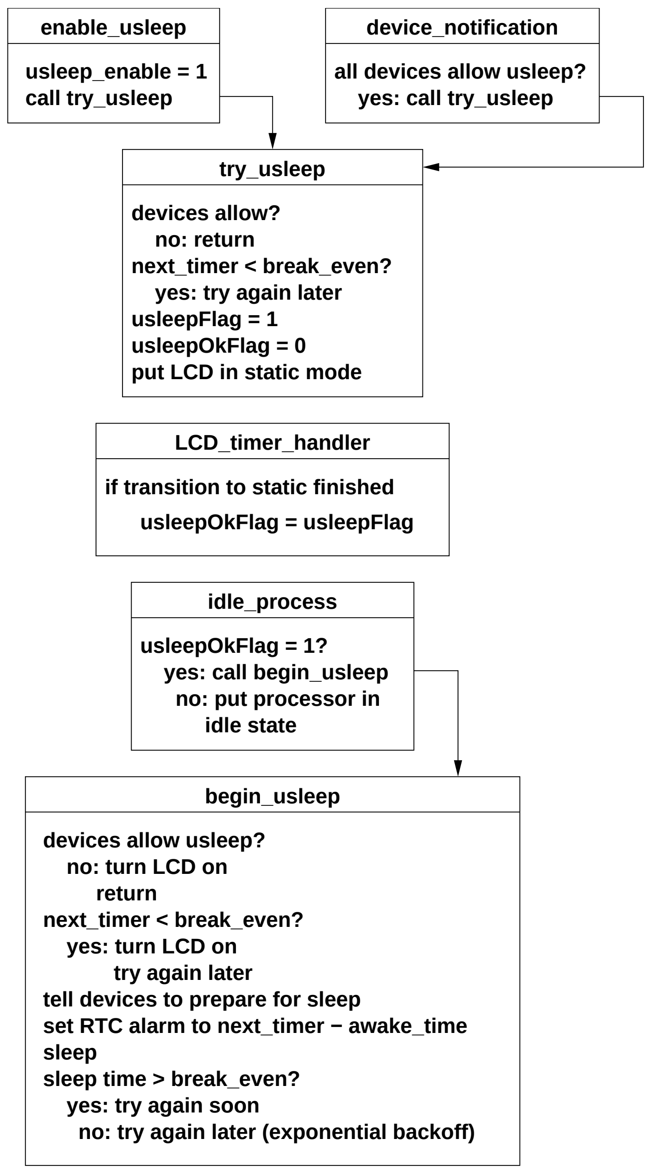
Our implementation of $\mu$Sleep on Itsy — which has an LCD with an integrated frame-buffer to display an image when the processor is sleeping — is more complicated than the generic implementation, where we have assumed that the LCD controller can keep running when the processor is sleeping. This complication is the result of a 16 ms latency when switching from displaying images from the LCD controller to displaying a static image in the LCD's frame-buffer. Rather than immediately transitioning from **OS idle** to **$\mu$Sleep** when the conditions are satisfied,[4] the transition to a static image is started and the system goes in the **processor idle** state until this transition is completed. At this time the conditions must be rechecked to verify that no external events occurred which would prevent the system from going to sleep.

We also had to solve the problem of how to wake up the system from $\mu$Sleep since the only timers on Itsy are the StrongARM SA-1100 processor's timers and all of them are disabled in sleep mode, except for the RTC. However, the RTC has an nominal resolution of 1 second, rather than the 1 to 10 ms resolution required to optimally support $\mu$Sleep. Fortunately, the StrongARM SA-1100 processor features a programmable clock divider and trim value, which are intended to calibrate the RTC with an accuracy of $\pm 5$ seconds per month or better. However, by appropriately programming the clock divider, it is possible to make the RTC run much faster. In our case, we programmed an RTC frequency of 1024 Hz.[5]

Determining when the next OS timer event will occur was easily solved; we added code to Linux to implement this functionality. Finally, to make $\mu$Sleep more robust, we added a backoff mechanism that is activated whenever the actual $\mu$Sleep period is too short (i.e., the system uses more energy than not sleeping at all) as a result of waking up due to an external event. The goal of the backoff mechanism is to prevent a worst case scenario where the device is repeatedly woken up from $\mu$Sleep after a few milliseconds. The implementation is simple. When a $\mu$Sleep period is too short we first determine if the last $\mu$Sleep period was also too short. If not, a given variable is set to 40 ms, otherwise this variable's value is doubled as long as it does not exceed 1 second. This variable's value is then used as the minimum time before we allow to enter $\mu$Sleep again. This variable is reset as soon as one $\mu$Sleep period is long enough to save energy. Figure 3 shows a pseudo-code implementation of $\mu$Sleep.

[4]The condition next_timer $>$ break_even is modified to next_timer $>$ break_even $+$ 16 ms to account for the latency in transitioning to a static image.

[5]As a result of changing the RTC frequency, we had to add code to handle the increased RTC wrapping frequency, which increased from once every 136 years to about once every 50 days.



```
enable_usleep
    usleep_enable = 1
    call try_usleep

device_notification
    all devices allow usleep?
      yes: call try_usleep

try_usleep
    devices allow?
      no: return
    next_timer < break_even?
      yes: try again later
    usleepFlag = 1
    usleepOkFlag = 0
    put LCD in static mode

LCD_timer_handler
    if transition to static finished
        usleepOkFlag = usleepFlag

idle_process
    usleepOkFlag = 1?
      yes: call begin_usleep
        no: put processor in
            idle state

begin_usleep
    devices allow usleep?
      no: turn LCD on
          return
    next_timer < break_even?
      yes: turn LCD on
           try again later
    tell devices to prepare for sleep
    set RTC alarm to next_timer − awake_time
    sleep
    sleep time > break_even?
      yes: try again soon
        no: try again later (exponential backoff)
```

**Figure 3: Pseudo-code implementation of $\mu$Sleep.**

## 4.2 Energy savings

The energy used ($e_{\mathrm{used}}$) per $\mu$Sleep period of length $T \geq t_{\mathrm{s}} + t_{\mathrm{r}}$ is given by:

$$e_{\mathrm{used}} = e_{\mathrm{s}} + e_{\mathrm{r}} + p_{\mathrm{s}} \cdot (T - t_{\mathrm{s}} - t_{\mathrm{r}}) \qquad (1)$$

Where:
$T$: $\mu$Sleep duration (incl. entering and exiting)
$t_{\mathrm{s}} = t_{\mathrm{s}}(f)$: time to enter sleep mode
$t_{\mathrm{r}} = t_{\mathrm{r}}(f)$: time to exit sleep mode (enter run mode)
$e_{\mathrm{s}} = e_{\mathrm{s}}(f)$: energy to enter sleep mode
$e_{\mathrm{r}} = e_{\mathrm{r}}(f)$: energy to exit sleep mode (enter run mode)
$p_{\mathrm{s}}$: sleep mode power with LCD enabled

For simplicity, we have omitted the processor core fre-

| Core frequency | Core voltage | Sleep duration | Average power |
|---|---|---|---|
| 59 MHz | 0.925 V | — | **54.3 mW** |
| | | 20 ms | 59.4 mW |
| | | 30 ms | 58.3 mW |
| | | 40 ms | 57.3 mW |
| | | 50 ms | 56.5 mW |
| | | 60 ms | 55.4 mW |
| | | **70 ms** | **54.1 mW** |
| 206 MHz | 1.450 V | — | **99.9 mW** |
| | | 20 ms | 105 mW |
| | | 30 ms | 102 mW |
| | | **40 ms** | **99.8 mW** |
| | | 50 ms | 97.0 mW |

**Table 1: Determination of break-even times ($t_{\mathrm{be}}$).**

| Core frequency | Core voltage | $t_{\mathrm{be}}$ |
|---|---|---|
| 59 MHz | 0.925 V | 70 ms |
| 74 MHz | 0.975 V | 60 ms |
| 89 MHz | 1.025 V | 60 ms |
| 103 MHz | 1.075 V | 50 ms |
| 118 MHz | 1.075 V | 50 ms |
| 133 MHz | 1.125 V | 50 ms |
| 148 MHz | 1.175 V | 70 ms |
| 162 MHz | 1.225 V | 70 ms |
| 177 MHz | 1.275 V | 40 ms |
| 192 MHz | 1.400 V | 40 ms |
| 206 MHz | 1.450 V | 40 ms |
| 221 MHz | 1.550 V | 30 ms |
| 236 MHz | 1.600 V | 30 ms |

**Table 2: Break-even times ($t_{\mathrm{be}}$).**

quency ($f$) from equation (1) and the equations below. However, all time, power, and energy constants used here depend on the processor frequency, with the exception of the sleep mode power. This dependency is a result of the required computation to enter and exit sleep mode, while preserving the system's state.

One can also use the *break-even time* ($t_{\mathrm{be}}$) to find the energy used per $\mu$Sleep period, without first determining $e_{\mathrm{s}}$, $e_{\mathrm{r}}$, $t_{\mathrm{s}}$, or $t_{\mathrm{r}}$. Let $t_{\mathrm{be}} = t_{\mathrm{be}}(f)$ be the sleep duration ($T$) such that the energy used during this time, including entering and exiting sleep mode, is equal to the energy that would be consumed if the system was idle for the same time interval. That is:

$$e_{\mathrm{s}} + e_{\mathrm{r}} + p_{\mathrm{s}} \cdot (t_{\mathrm{be}} - t_{\mathrm{s}} - t_{\mathrm{r}}) \quad = \quad p_{\mathrm{i}} \cdot t_{\mathrm{be}} \tag{2}$$

Where:
$p_{\mathrm{i}} = p_{\mathrm{i}}(f)$: idle mode power.

Then, we can calculate the energy used ($e_{\mathrm{used}}$) for a $\mu$Sleep period as well as the energy saved ($e_{\mathrm{saved}}$) with respect to an idle period of the same duration ($T$):

$$\begin{aligned} e_{\mathrm{used}} &= p_{\mathrm{i}} \cdot t_{\mathrm{be}} + p_{\mathrm{s}} \cdot (T - t_{\mathrm{be}}) && (3)\\ e_{\mathrm{saved}} &= p_{\mathrm{i}} \cdot T - e_{\mathrm{used}} \\ &= (p_{\mathrm{i}} - p_{\mathrm{s}}) \cdot (T - t_{\mathrm{be}}) && (4) \end{aligned}$$

Note that $e_{\mathrm{saved}}$ can be negative if $T < t_{\mathrm{be}}$, indicating that energy has been wasted rather than saved.

The break-even time ($t_{\mathrm{be}}$) can be determined for each processor frequency ($f$) in the following way. First, we measure the average power ($p_{\mathrm{i}}$) when the system is idle (see Section 5.2 for details on our measurement procedure). Next, we measure average system power while doing short duration sleeps every 250 ms. In our case, we started with a sleep duration of 20 ms and increased it by 10 ms until the average power consumed is equal to $p_{\mathrm{i}}$. The sleep duration when this happens is $t_{\mathrm{be}}$. Table 1 shows the results of our experiments to determine the break-even time for 59 MHz and 206 MHz, which are 70 ms and 40 ms, respectively. The break-even times for all frequencies are shown in Table 2. Note that in these experiments, as well as in all subsequent experiments, we fix the processor's frequency for the duration of the experiment (we don't use dynamic frequency scaling). The voltage used for each frequency is the pre-determined lowest adequate voltage, except at 59 MHz where we also use 1.45 V to examine the case of a device that doesn't support voltage scaling. Using the lowest adequate voltage for each frequency corresponds to a worst case for $\mu$Sleep, since a system that does not support voltage-scaling would have shorter break-even times ($t_{\mathrm{be}}$) at low frequencies, hence, make $\mu$Sleep even more desirable.[6]

Although equation (4) does not account for all effects of $\mu$Sleep — in particular, the fact that a $\mu$Sleep period is sometimes longer than the corresponding idle period would have been — it is still a good approximation and can be used to predict the energy saved, without actually measuring it. Once the constants $t_{\mathrm{be}}$, $p_{\mathrm{i}}$, and $p_{\mathrm{s}}$ have been determined for a given system once and for all, the OS can easily be modified to record the number of $\mu$Sleep periods ($N_{\mathrm{s}}$) as well as the average duration of these periods ($t_{\mathrm{avg}}$). Alternatively, one can measure $N_{\mathrm{s}}$ and $t_{\mathrm{avg}}$ for potential $\mu$Sleep periods, if one wants to study the effectiveness of $\mu$Sleep before implementing it. Based on equation (4), the total energy saved ($E_{\mathrm{saved}}$) is:

$$E_{\mathrm{saved}} = N_{\mathrm{s}} \cdot (p_{\mathrm{i}} - p_{\mathrm{s}}) \cdot (t_{\mathrm{avg}} - t_{\mathrm{be}}) \tag{5}$$

Table 3 shows the accuracy of equation (5) at predicting the energy that is saved by using $\mu$Sleep in various scenarios. In the first scenario (idle), the system is idle for 10 minutes. In the second scenario (read), a user reads a document on the system. In the final scenario (calendar), a user interacts with a calendar program performing a fixed set of tasks. The scenarios are described in more detail in Section 6. We first measured the energy consumption when the scenarios ran with $\mu$Sleep disabled, then we measured the energy consumed when the scenarios ran with $\mu$Sleep enabled. At the same time, we also recorded $N_{\mathrm{s}}$ and $t_{\mathrm{avg}}$. As seen in column 7, for our experiments the prediction error is less than 7%.

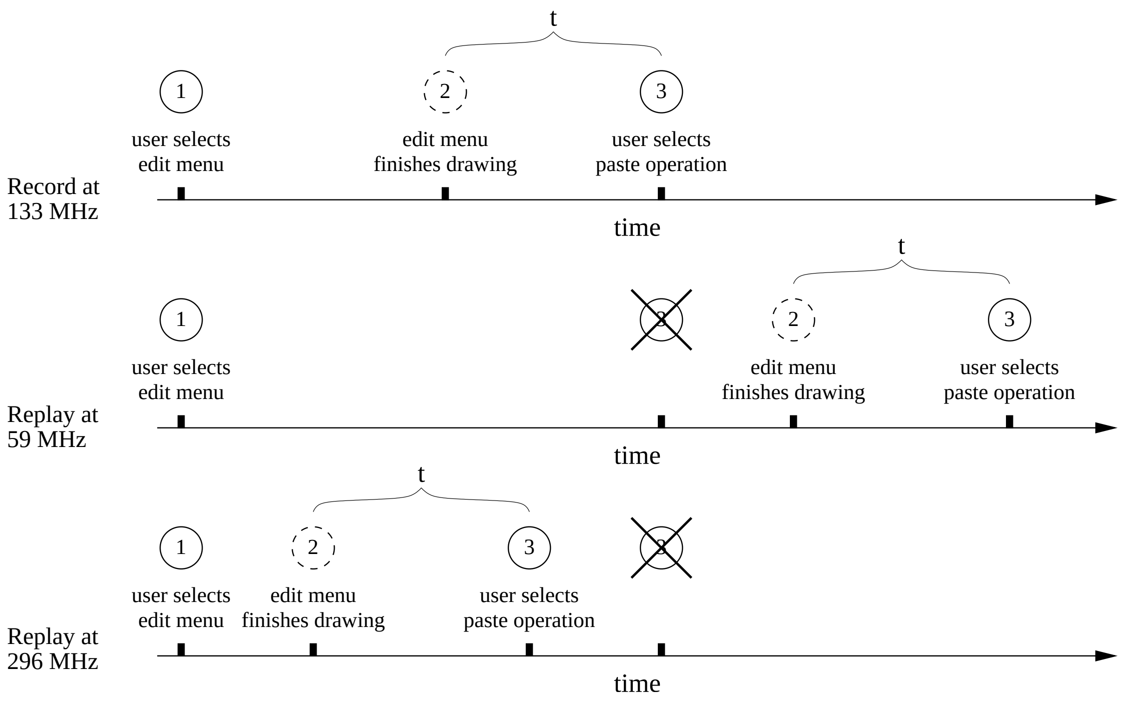
## 5. EVALUATION INFRASTRUCTURE

We have created an evaluation infrastructure in order to evaluate the performance of $\mu$Sleep, both in terms of its effectiveness at reducing energy consumption and of the effects that it has on system performance. This evaluation infrastructure consists of two parts. The record/replay system

[6]The break-even times would be shorter on a system without voltage-scaling because the power consumed while in idle mode would be larger but the power consumed while sleeping would be the same.

| Scenario | Core frequency | Energy w/o $\mu$Sleep | Energy w/ $\mu$Sleep | Actual $E_{\text{saved}}$ | Predicted $E_{\text{saved}}$ | Error | $N_s$ | $t_{\text{avg}}$ |
|---|---|---|---|---|---|---|---|---|
| idle | 59 MHz | 33.9 J | 20.6 J | 13.3 J | 13.4 J | 1% | 661 | 829 ms |
| idle | 206 MHz | 62.1 J | 23.7 J | 38.4 J | 37.7 J | 2% | 649 | 852 ms |
| read | 59 MHz | 33.7 J | 29.3 J | 4.4 J | 4.1 J | 7% | 292 | 671 ms |
| read | 206 MHz | 43.1 J | 29.4 J | 13.7 J | 12.8 J | 7% | 278 | 740 ms |
| calendar | 59 MHz | 64.2 J | 59.5 J | 4.7 J | 5.0 J | 6% | 603 | 370 ms |
| calendar | 206 MHz | 74.1 J | 60.4 J | 13.7 J | 14.2 J | 4% | 683 | 325 ms |

**Table 3: Comparison of measured and predicted energy savings.**



t
1
2
3
user selects edit menu
edit menu finishes drawing
user selects paste operation
Record at 133 MHz
time
t
1
3
2
3
user selects edit menu
edit menu finishes drawing
user selects paste operation
Replay at 59 MHz
time
t
1
2
3
3
user selects edit menu
edit menu finishes drawing
user selects paste operation
Replay at 296 MHz
time

**Figure 4: Effects of processor frequency on replaying events.**

allows us to record, for later replay, the user input events that occur as a user interacts with the device. The measurement system allows us to measure the power consumed by the device, as well as record other system information such as processor frequency, idle time, etc.

The user events recorded during a particular experiment is referred to as a *user scenario*. We consider three user scenarios, all of which are part of the the Qt Palmtop Environment (QPE) by Trolltech (version 1.1.1). QPE consists of a GUI library as well as common PDA applications such as a calendar, address book, etc. In the *idle* user scenario, QPE is running, but there are no user events so the device is mostly idle (the Linux OS is performing intermittent computation). In the *read* user scenario, the user starts programs to read information and presses buttons and the touch screen to navigate through the documents. Finally, in the *calendar* user scenario, the user is interacting with a calendar program, both viewing and entering events.

By performing experiments where we replay the same user interactions as we try different energy management techniques, we are able to meaningfully evaluate and compare these techniques. We can also measure the performance of the system with no energy management to determine a baseline against which to compare.

## 5.1 Record/replay system

As illustrated in Figure 4, one cannot record events on a system running at one processor frequency and replay them naively when the processor is running at a different frequency. The figure consists of three time lines, each showing three events. The first event corresponds to a user selecting a generic edit drop-down menu. The second event corresponds to the time when the system finishes drawing the drop-down menu. The last event corresponds to the user selecting the paste operation from the edit menu. The first time line is based on a system running at 133 MHz. The user events (1 and 3) are recorded in this system for later replay.

The second time line is based on a system running at 59 MHz. Because of the lower processor frequency (59 MHz instead of 133 MHz), the edit menu takes longer to draw. As a result, it would not be meaningful to replay event 3 at the same time as it was recorded. A more meaningful interpretation would be to assume that the delay (t) between events 2 and 3 is fixed since it corresponds to the “think time” of the user. Therefore, event 3 should be replayed after the same delay (t) has elapsed since event 2. A similar approach must be followed for the third time line, where the

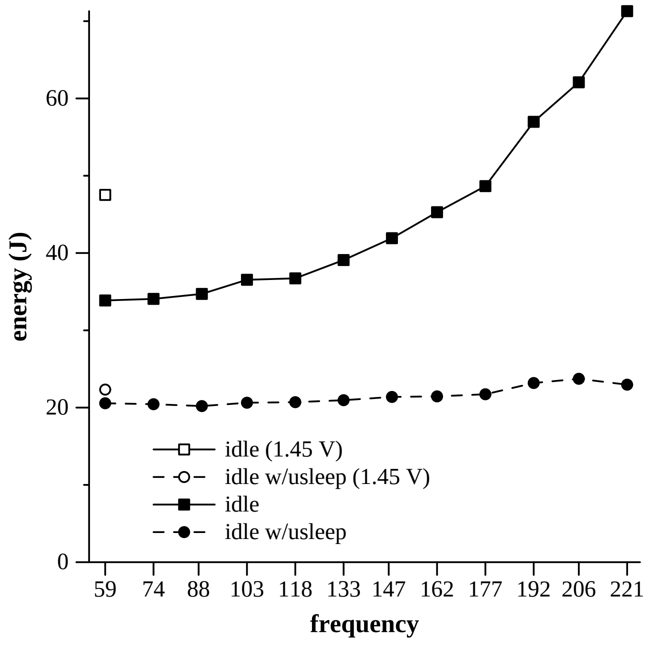


**Figure 5: Energy usage in idle scenario.**

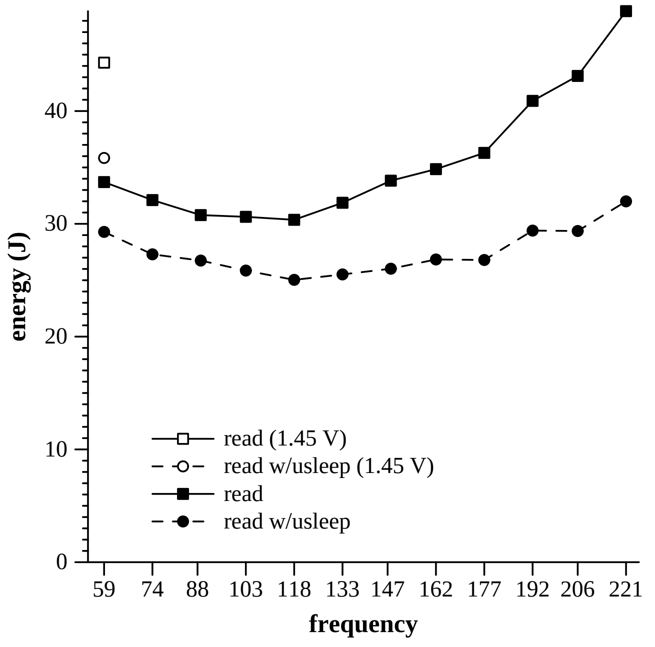


**Figure 6: Energy usage in read scenario.**

edit menu finishes drawing earlier as a result of the system running at 296 Mhz.

Our record and replay infrastructure is OS based and can be used with any application. It determines when processing for a user event is finished (system event 2 in Figure 4) and stores this information. When the next user interaction happens (event 3 in Figure 4), the record system determines the delay between these two events (time t in Figure 4) and stores it along with the user events. When replaying the events, the replay system determines when processing for a user event is finished and then delays the necessary time (t) before replaying the next user event. The record/replay system is not only able to meaningfully replay events independently of the processor frequency, but it is also able to record the processing time associated with each user event. This information allows us to determine the effect that a given energy management technique has on the response time of the system.

Our record/replay system is able to deal with both static changes in processor frequency, where the processor frequency is fixed for the whole experiment, as well as dynamic changes, where the processor frequency may change during the experiment. Finally, the replay system can also deal with the processor state transitions that occur with $\mu$Sleep. In particular, if the device is in a $\mu$Sleep short-duration sleep, the replay system will trigger an interrupt, as a result of a replayed external event, at the time when the external event would have happened, and not ahead of time (as it occurs with OS scheduled events). This behavior produces the correct response times when doing $\mu$Sleep.

## 5.2 Measurements

The Itsy features several sense resistors that allow us to monitor the power dissipated by the whole system. We used a measurement setup fairly similar to a setup designed elsewhere [17]. Briefly summarized, we measure voltages directly and currents indirectly as voltage drops across sense resistors. For all experiments, the battery was replaced by a power supply set to 3.75 V. Itsy, the multimeters, and the power supply are connected to a computer, which can run experiments fully automatically. When carefully implemented, this strategy results in error terms smaller than typical system-to-system variations.

## 6. RESULTS

As mentioned in section 5, we used three user scenarios in our experiments: when the system is mostly idle, when a user is reading documents and when the user is interacting with a calendar program. We ran each of these scenarios at 12 processor frequencies in the range between 59 MHz and 221 MHz while using voltage scaling. The voltages associated with each frequency are shown in table 2. The user scenarios ran at 59 Mhz twice, once with voltage scaling (0.925 V) and once without (1.45 V).[7]

The rest of this section discusses the results of our experiments. For the first three figures, where we compare the energy consumed when we replay a user scenario with and without $\mu$Sleep, a solid line is used to connect the experiments without $\mu$Sleep and a dashed line to connect the experiments where we enabled $\mu$Sleep. Furthermore, the case where there is no voltage scaling (59 MHz at 1.45 V) is indicated by marks that are not filled in.

Figure 5 shows the energy consumed during a 10 minute experiment when there is no user interaction; that is, the system is mostly idle. There is still computation going on, such as updating the clock shown on the display, periodic processing by system processes, etc. The advantage of $\mu$Sleep is clearly visible for each frequency, with energy

[7]Note that 1.45 V is the voltage required to run the processor at 206 MHz, which is the maximum frequency that the Itsy was designed to run at (prior to the voltage scaling modifications).

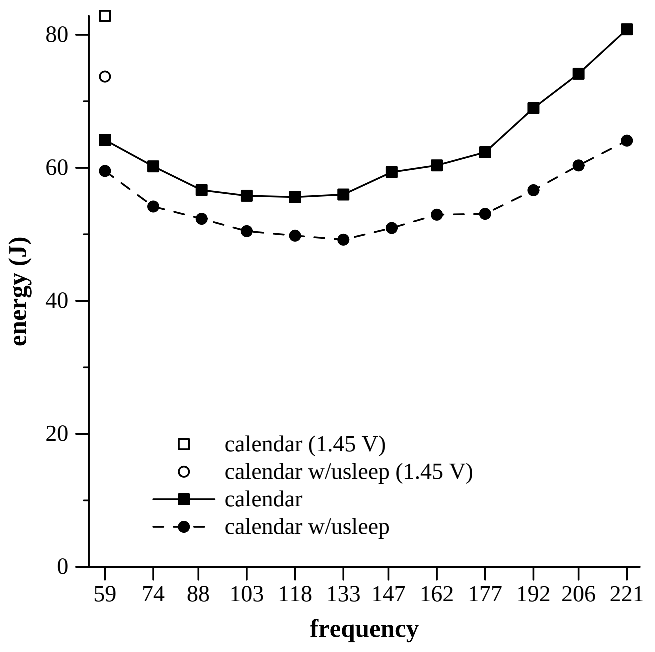


**Figure 7: Energy usage in calendar scenario.**

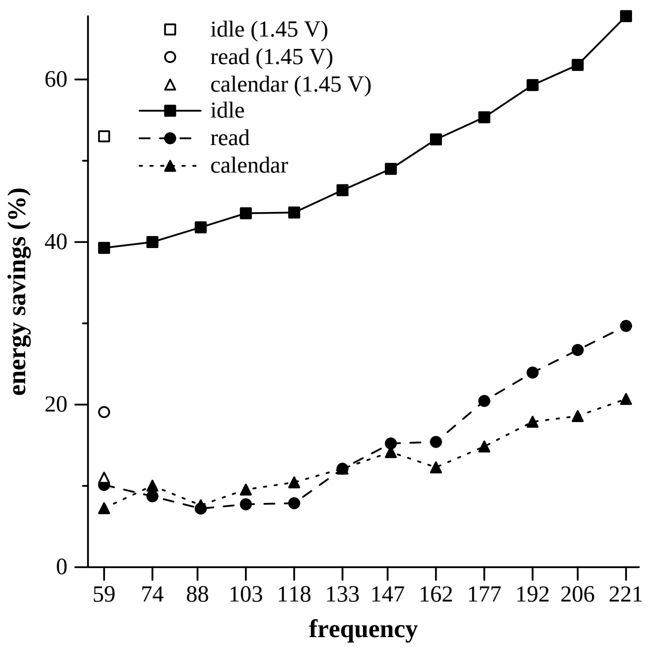


**Figure 8: Energy savings for all scenarios.**

savings ranging from 40% at 59 MHz to 68% at 221 MHz. Note that the energy consumed when $\mu$Sleep is used is almost constant and is independent of the processor frequency. The reason for this is that since the system is mostly idle, the processor is spending most of its time in its sleep mode in the form of short duration sleeps. The average sleep duration is 650 ms; the average awake duration is 55 ms.

Figure 6 shows the energy consumed by the read scenario. The read scenario lasted between 4.7 and 5.6 minutes depending on the processor frequency. The energy savings ranged from 13% at 59 MHz to 35% at 221 MHz. The average sleep duration depends on the processor frequency. At 59 MHz, the average sleep duration is 229 ms and the average awake duration is 278 ms. At 221 MHz, the sleep duration is a little larger, 271 ms, and the awake duration is much less, about 97 ms (note that the system goes through more sleep-awake cycles during the 221 MHz experiment).

Figure 7 shows the energy consumed by the calendar scenario. The scenario lasted between 6 to 8 minutes depending on the processor frequency. The energy savings were less than those achieved in the read scenario, ranging from 7% at 59 MHz to 21% at 221 MHz. At 59 MHz, the average sleep duration is 370 ms and the average awake duration is 453 ms. At 221 MHz, the sleep duration is 312 ms and the awake duration is 197 ms.

It is not surprising that the awake duration is less at the higher processor frequencies since the required computation can be performed much faster. Note that we used an early version of QPE (1.1.1) for our experiments which is known to have performance problems. We would expect much better energy savings from $\mu$Sleep if we were using a current version of QPE.

The "U" shape of the energy curve in Figures 6 and 7 can be explained as follows. The power consumed by the system can be broken into the power consumed by the processor core and the power used by the rest of the system. As the core frequency is increased, the core voltage needs to be increased following an approximately linear relationship. Therefore, we expect the graph of core energy vs. frequency to have the shape of a quadratic function, since:

$$E \propto t \cdot f \cdot v^2 \qquad (6)$$

On the other hand, although the power consumed by the rest of the system is indirectly affected by the core frequency and voltage, it is not affected in a monotonic fashion and the fluctuations are small enough that, to a first degree, it can be approximated as constant. As a result, since the execution time of the read and calendar scenarios is decreasing as the frequency increases, the energy is also decreasing. The energy used by the whole system is the sum of these two functions. One function, the energy consumed by the core, increases quadratically while the other, the energy used by the rest of the system, decreases approximately linearly, hence explaining the "U" shape of the curve.

Figure 8 shows the energy saved by using $\mu$Sleep as a percentage of the energy used when not using $\mu$Sleep. As expected, the energy savings increase as a function of the core frequency. When looking at the figures for the read and calendar scenarios one could assume that the best approach would be to always run at a core frequency of 133 MHz since this is where the lowest value of the energy curve occurs. However, running at lower fixed core frequencies increases the response time of the system (the time between a user event and when the system is ready to process the next event), as demonstrated in Figure 9.

## 7. DISCUSSION

One of the stated goals of $\mu$Sleep is that there should be no effect on the user experience. That is, the user should be unaware that the device is doing short duration sleeps. As mentioned earlier, when the processor is in the sleep mode, the Itsy can only display a monochrome static image. This

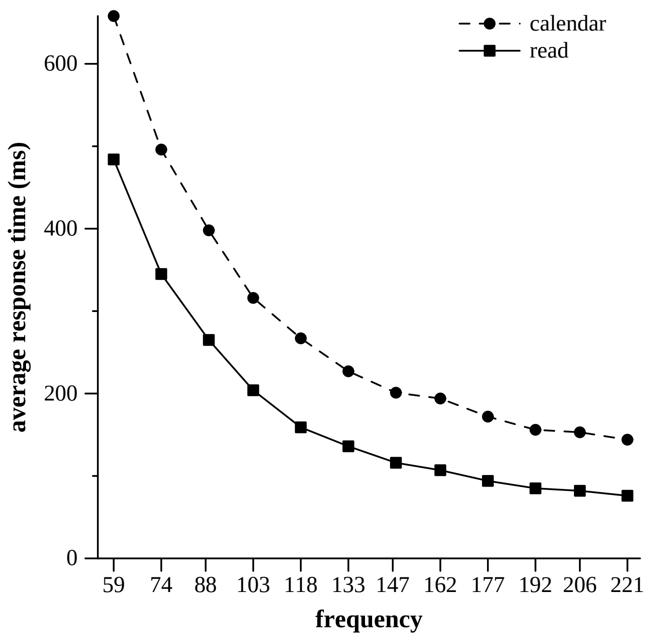


**Figure 9: Response time for read and calendar scenarios.**

is in contrast to the 16-level grayscale images that can be displayed when the processor is awake. However, the perceptual difference between the two images is reduced by the fact that the monochrome static image is a dithered version of the grayscale one.[8] We also fine-tuned our graphics library to further reduce the visual differences between the graylevel and dither versions of its 3D widgets.

For certain image types, such as photographs, the differences between the monochrome and graylevel images are more noticeable. If this behavior is not desired, $\mu$Sleep can be automatically disabled when certain types of programs, such as image viewers, are in the foreground. However, note that the visual effects of $\mu$Sleep on Itsy are the result of the platform being designed before $\mu$Sleep was conceived. These effects can be eliminated by the techniques described in Section 4.1, such as by using an external LCD controller. For example, the Pocket PC implementation discussed at the end of this section was done on a device with a color screen and with an LCD controller external to the processor. As a result, there are no visual artifacts during the sleep intervals on this device.

The most effective way to increase the performance of $\mu$Sleep is to increase the length of the short duration sleeps as well as their frequency. Since $\mu$Sleep wakes the system up right before the next OS scheduled event, the length of the short duration sleeps can be increased if we can reduce the number of OS scheduled events. Since one of the most common sources of OS scheduled events is application timers, reducing their number can significantly increase the performance of $\mu$Sleep. For example, we found that the default GUI on Itsy (the manager) was using distinct timers to update each of the status icons. These status icons show the time and date, the current processor load, remaining battery, volume level, etc. Each of the status icons were being updated every second, and since they were not synchronized, the average interval between timers was less than 250 ms. By using just one timer to update all the status icons we increased the interval between timers to more than 600 ms.

A general way to implement this functionality could be achieved by creating a new system function to set periodic timers. The function could have two arguments, the first argument specifies the interval between timers (1 second for our status icons example), the second argument indicates the maximum delay allowed before the first occurrence of the timer. Then, multiple calls to set timers with a period of one second, or multiples of one second, would result in timers that expire at the same time.

However, there may be times when we don't have access to the application's code to improve their handling of timers. In many instances the behavior of the application is not affected negatively if the timers are delayed by a short amount. Then, the code that checks for the next OS scheduled event could be modified to check if the scheduled event belongs to a particular set of applications which are known to be resilient to timer delays. If so, the sleep duration could be increased beyond the OS scheduled event.

On Itsy we cannot use $\mu$Sleep when an application is actively using the audio device[9] because the processor needs to be awake to transfer the data to, or from, the audio codec. However, if the audio codec had sufficient buffering, then it would be possible to transfer 200 ms or more of audio at one time, then sleep until it is time to start processing or sending more data. This way it could be possible, depending on the device particular characteristics, to use $\mu$Sleep while the device is playing an encoded audio file.

Similar issues arise with other I/O devices such as the processor's serial port. These issues can be resolved in a similar way; either by increasing the available buffering in the case that the device is external to the processor, or by allowing the device to stay awake while the processor sleeps.

One of the advantages that $\mu$Sleep has over the standard use of system sleep is that the OS and its applications are able to perform all their computation. As a result, network connections can stay alive because all code related to connection timers is executed as normal. Therefore one could create a variant of $\mu$Sleep to replace the standard system sleep, where network connections would stay alive. This variant of $\mu$Sleep would turn the display off as well as disable some of the wake up events of standard $\mu$Sleep (such as touch screen presses). Furthermore, to increase the amount of energy saved, this variant could also increase the duration of the short sleep periods, say to one or two seconds, ignoring the time of the next OS scheduled event (most common network connections would still stay alive even if their timers are delayed by a couple of seconds).

The final topic we want to cover in this section is our experience porting $\mu$Sleep to a Pocket PC device where we were constrained by the lack of access to the Pocket PC OS code. Although we never achieved a full port of $\mu$Sleep to the Pocket PC device due to time constraints, we achieved enough functionality to determine that it would be feasible to do a full implementation of $\mu$Sleep. Furthermore, initial measurements on a Pocket PC with a color screen showed

[8]This is automatically done by the LCD controller in the StrongARM.

[9]If an application has opened the audio device but hasn't used it in the last 5 seconds, then we enable $\mu$Sleep until the user accesses the audio device.

that we could achieve power savings of up to 35% when the backlight was off, and up to 20% when the backlight was set at a comfortable level.

## 8. CONCLUSIONS

We have introduced $\mu$Sleep, a new technique for reducing power consumption in computing devices. This technique is most useful in portable devices, where it can be used to significantly increase the battery life of the device in some cases. We implemented $\mu$Sleep on the Itsy pocket computer both to determine the viability of this technique and to measure the energy savings achieved by this technique. Our experiments have shown that $\mu$Sleep can reduce energy consumption by more than 60% in some instances, such as when the device is lightly loaded. We have also done a preliminary implementation of $\mu$Sleep on a Pocket PC based device, giving evidence that this technique could be implemented on diverse devices, as long as some basic requirements are met.

## Acknowledgements

The Itsy project was started at Digital's Western Research Laboratory, and led by William Hamburgen, who co-designed the hardware with one author (M. Viredaz). The other two authors (L. Brakmo and D. Wallach) and Carl Waldspurger ported the Linux operating system to the Itsy. Many other members of the Western Research Laboratory and the System Research Center made significant contributions to the project (while at Digital and then Compaq). The work on $\mu$Sleep was performed while the authors were members of the research staff at Compaq and then HP Labs. In addition, the authors wish to acknowledge the enthusiastic support of their management at all three companies.

## 9. REFERENCES

[1] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *USENIX Winter 1994 Technical Conf. Proc.*, pages 293 – 306, San Francisco, Jan. 1994. USENIX.

[2] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proc. Seventh Annual Int'l Conf. on Mobile Computing and Networking*, Rome, July 2001. ACM, IEEE, ACM Press.

[3] S. Gary, P. Ippolito, G. Gerosa, C. Dietz, J. Eno, and H. Sanchez. PowerPC 603,$^{\text{TM}}$ a microprocessor for portable computers. *IEEE Design & Test of Computers*, 11(4):14 – 23, Winter 1994.

[4] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proc. First Int'l Conf. on Mobile Computing and Networking*, pages 13 – 25, Berkeley, Nov. 1995. ACM, NASA, ACM Press.

[5] W. R. Hamburgen, D. A. Wallach, M. A. Viredaz, L. S. Brakmo, C. A. Waldspurger, J. F. Bartlett, T. Mann, and K. I. Farkas. Itsy: Stretching the bounds of mobile computing. *Computer*, 34(4):28 – 36, Apr. 2001.

[6] C.-H. Hwang and A. C.-H. Wu. A predictive system shutdown method for energy saving of event-driven computation. *ACM Trans. on Design Automation of Electronic Systems*, 5(2):226 – 241, Apr. 2000.

[7] Intel. *Intel® StrongARM® SA-1100 Microprocessor: Developer's Manual*, Aug. 1999. Document No. 278088-004.

[8] Intel. *Intel® XScale$^{\text{TM}}$ Microarchitecture: Technical Summary*, 2000.

[9] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami. Energy trade-offs in the ibm wristwatch computer. In *Proc. Fifth International Symp. on Wearable Computers*, pages 133 – 141, Zurich, Oct. 2001. IEEE.

[10] K. Li, R. Kumpf, P. Horton, and T. Anderson. A quantitative analysis of disk drive power management in portable computers. In *USENIX Winter 1994 Technical Conf. Proc.*, pages 279 – 291, San Francisco, Jan. 1994. USENIX.

[11] J. R. Lorch and A. J. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311 – 324, Oct. 1997.

[12] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. 1998 Int'l Symp. on Low Power Electronics and Design*, pages 76 – 81, Monterey, Aug. 1998. IEEE, ACM, ACM Press.

[13] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Int'l Symp. on Mobile Multimedia Systems & Applications*, Delft, Nov. 2000.

[14] T. Simunic, L. Benini, P. Glynn, and G. De Micheli. Dynamic power management of portable systems. In *Proc. Sixth Annual Int'l Conf. on Mobile Computing and Networking*, pages 11 – 19, Boston, Aug. 2000. ACM, IEEE, ACM Press.

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

[16] R. Stephany, K. Anne, J. Bell, G. Cheney, J. Eno, G. Hoeppner, G. Joe, R. Kaye, J. Lear, T. Litch, J. Meyer, J. Montanaro, K. Patton, T. Pham, R. Reis, M. Silla, J. Slaton, K. Snyder, and R. Witek. A 200MHz 32b 0.5W CMOS RISC microprocessor. In *1998 IEEE Int'l Solid-State Circuits Conf.: Digest of Technical Papers*, pages 238 – 239, 443, San Francisco, Feb. 1998. IEEE.

[17] M. A. Viredaz and D. A. Wallach. Power evaluation of a handheld computer. *IEEE Micro*, 23(1):66 – 74, Jan. – Feb. 2003.

[18] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. First Symp. on Operating Systems Design and Implementation*, pages 13 – 23, Monterey, Nov. 1994. USENIX.