RC23675 (W0507-165) July 21, 2005
Computer Science

IBM Research Report

PowerNap: An Efficient Power Management Scheme for Mobile Devices

C. Michael Olsen, Chandra Narayanaswami
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

PowerNap: An Efficient Power Management Scheme for Mobile Devices

C. Michael Olsen and Chandra Narayanaswami

IBM Research Division, 19 Skyline Dr., Hawthorne, NY 10532

Abstract: *We present PowerNap, an OS power management scheme, which can significantly improve the battery life of mobile devices. The key feature of PowerNap is the skipping of the periodic system timer ticks associated with the operating system. On an idle device, this modification increases the time between successive timer interrupts and enables us to put the processor/system into a more efficient low power state. This saves the energy consumed by workless timer interrupts and the excess energy consumed by the processor in less efficient low power states. PowerNap is tightly integrated with the kernel and is designed for optimal control of the latency and energy associated with transitioning in and out of the low power states. We describe an implementation of PowerNap and its impact on system software. Experiments with IBM's WatchPad verify the ability of PowerNap to extend battery life. An analytical model that quantifies the ability of the scheme to reduce power is also presented. The model is in good agreement with experimental results. We apply the model to small form-factor devices which use processors that have a PowerDown state. In such devices PowerNap may extend battery life by more than 42% for small processor workloads and for background power levels below 10 mW.*

Index Terms: Power management, operating systems, mobile systems, processors.

1. Introduction

Power management has become one of the most significant challenges in mobile computing. It is being investigated at the device, circuit, and architectural levels for processors, memories, displays, wireless subsystems, etc. Simultaneously, software architectures to exploit

hardware enhancements is evolving. Moreover, system software tradeoffs are being revisited with energy conservation as a main goal. Power management approaches largely fall into two categories; *active* and *passive*. In the active category the aim is to reduce the energy required to complete a task while in the passive the aim is to put devices into a low power state whenever possible. This paper focuses on the passive category but also addresses some active issues.

We present an operating system power management technique, called *PowerNap*, which utilizes a processor's power states more efficiently. We build on a technique presented in [1] that modifies the timing mechanism of the operating system. The technique applies best to a class of general purpose mobile devices that are mostly idle but need to be able to respond instantly, e.g., to a button press, and need to handle multiple applications when required. Our motivation comes from the fact that mobile devices spend the majority of their time idling. Our techniques are also useful for other mostly idle devices, such as equipment in offices and kiosks, because conserving power and reducing heat generation is becoming more important and is good for the economy.

The paper has the following goals. First, to give a detailed description of *PowerNap*. Second, to make a methodology available that a mobile device designer can use to estimate the potential benefits of deploying *PowerNap* on a mobile device. To our knowledge, such a methodology does not exist. Third, to present a power state selector to be used in conjunction with the scheme. This component dynamically determines the optimal low power state to exploit at any given time. We discuss the stringent demands this component puts on the kernel timer chain and on device driver interactions. Fourth, to quantify, experimentally and analytically, the achievable gains in battery life as a function of various types of workloads. Finally, to predict that significant gains in battery life may be achieved by deploying *PowerNap* in systems with state-of-the-art processors and which feature the efficient *PowerDown* state.

2. Processor and OS Characteristics

Modern System-On-a-Chip (SOC) processors have multiple low power states that can be utilized by an OS. However, an OS that uses a *periodic timing* (PT) scheme is not able to take full advantage of the most efficient power states. To do this requires modifications to the OS timing mechanism. To make the paper self-contained, we first explain power management in SOCs and then discuss the limitations of the PT scheme used in most popular OSs.

2.1 Processor Power Management States

Table 1 lists low power states found in most advanced SOCs. The table uses descriptive names for these states as typically the names of the power states vary between SOCs. To explain the power states in more detail, we shall first give a quick summary of SOC architecture. A SOC is composed of several cores. Examples of cores include the CPU, LCD controller, SDRAM controller, Power Management unit, UART, on-chip oscillator, PLL, etc. The CPU core is clocked independently and at a higher clock rate than the rest of the cores. Non-CPU cores are referred to as peripheral cores and their clock as the peripheral clock.

Power State	**Clock state: CPU, peripheral**	**Power**	**Transition time, energy**
		[mW]	[ms], [μJ]
Idle	Off, On	>5	0, 0
ClockSuspend	Off, Off	0.25-10	>0.1, >1
PowerDown	Off, Off	0.05-0.2	>2, >25

Table 1. Definition and characteristics of low power states found in recent 32-bit mobile processor. The "Clock" column indicates the clock state in the CPU core and in the Peripheral cores. The "Power" column indicates minimum processor power level. The "Transition" column indicates minimum time and energy required to enter and exit the power state. The system crystal oscillator is running in the *ClockSuspend* state and is turned off in the *PowerDown* state. The table is mostly from [2].

In the *Idle* state the clock to the CPU core is stopped. Peripheral cores remain clocked. All processors have this state. Many processors [3-6] also have a *ClockSuspend* state in which the clock is globally stopped. The only peripheral cores that remain active are the power management unit, the real-time clock, and the interrupt controller unit. The logical state in the

cores is preserved. The drawback of this state is that it disables cores such as the LCD controller and asynchronous interfaces such as UART and USB. Thus, the LCD controller, for example, can not maintain an image on an LCD in this state. It also takes longer to exit this state due to the PLL stabilizing upon wakeup (100-200 μs.) Some processors also disable the on-chip oscillator which then has to stabilize upon wakeup (1-10 ms). However, an older processor, such as Cirrus Logic EP7211, may take up to 250 ms to exit this state [3]. Last, some processors [6,7] have a *PowerDown* state in which power is removed from the CPU core and from most of the peripheral cores. The power management unit, real-time clock and interrupt controller unit remain active to enable fast wake-up and to maintain time. The PLL and the on-chip oscillator are typically powered off too. The drawback of this state is that all SOC state and cache content are lost. Thus SOC state must be saved on entering this state and restored on exit. This takes time and energy. In [2] it is shown that this time and energy can not be ignored. It is discussed more in Section 6.

Table 1 shows there can be a substantial difference in power consumption and in transition time and energy depending on the power states. When entering the *Idle* state, the peripheral bus frequency may be reduced to minimize switching power dissipation in the peripherals. Even though, in theory, the frequency can be reduced to below 1 MHz to make the active switching power insignificantly small, the combined power drain from the leakage current, the on-chip oscillator and the PLL will limit power consumption to ~5 mW in most modern SOCs. In the *ClockSuspend* state the power consumption may be significantly smaller, especially in older processors where leakage currents are small. But in modern processors fabricated in a 0.13 μm process, the leakage current limits the power consumption to several mW in this state (at 25°C). The problem is expected to worsen in next generation SOCs [8]. Leakage current is also strongly temperature dependent. At 70°C the leakage current increases by a factor of ~6 [9]

compared to 25°C. The fact that the power consumption in the *ClockSuspend* state is getting so large is making the *PowerDown* state increasingly more attractive.

2.2 Periodic Timing

To implement *PowerNap* we need to modify the OS. We selected the freely available Linux OS for this purpose. Linux, like many other OSs, is implemented around the notion that it will receive periodic timer interrupts. This periodic interrupt is known as the “tick”. We denote this type of timing as Periodic Timing (PT). In Linux, the variable *jiffies* counts the number of ticks since kernel startup and it is used to update kernel time and process times and to check expiration of callback timers. Ticks are also well suited for multitasking environments when several tasks are running. From a power perspective though there are drawbacks to a PT scheme.

Wasting energy in workless timer ticks: The tick periodically wakes the processor up and causes the timer interrupt handler to be executed. This happens even when the OS is idling (i.e., when no tasks are running.) However, whenever the OS is idling, the queues and lists that need to be checked are empty and contain no expired callback timers. Only time gets updated during such ticks. A periodic timer interrupt, however, is not needed to maintain time.

State transition delays exceeding the tick interval: There may be more power efficient low power states that can not be exploited because the time it takes to transition in and out of the low power state exceeds the tick interval. For example, the most efficient low power states in the Cirrus Logic EP7312 [10] is the *ClockSuspend* state (Cirrus denotes it STANDBY) and in the Intel StrongARM 1110 [11] it is the *PowerDown* state (Intel denotes it SLEEP). However, it may take up to 250 ms and 160 ms, respectively, to exit these states. With a periodic interrupt occurring, say, every 10 ms, entering these low power states would result in ticks getting missed. This would make the OS timer callback service unreliable and would be disastrous for time

keeping. Besides, less than 10 ms will be spent in the low power state before the next interrupt occurs which will transition the processor out of the low power state again.

Unnecessary state transition energy consumption: As may be seen from Table 1, the more power efficient low power states also require more energy and latency to transition into and out of the state. Therefore, even though the power consumption in a more power efficient state, say *pm2*, is smaller than the power consumption in a less efficient state, say *pm1*, the energy required to simply transition into and out of the *pm2* state may actually make it more expensive to use the *pm2* state, contrary to intuition. Which one of either *pm1* or *pm2* is the most efficient state will depend on the time between the two adjacent timer ticks, on the transition times and on the transition energies. A large transition energy is bad for the PT scheme since this energy is unnecessarily spent on every workless tick and adds to the overall average power consumption.

Disabling of the system timer: An internal system timer is initially populated with a load value corresponding to the timer interrupt interval. When the counter reaches zero, an interrupt is generated. The initial load value is automatically reloaded on the next clock edge. This way of operating the timer is known as the prescale mode and requires zero maintenance. Unfortunately, the system timer is disabled in the more efficient low power states. Since it is the system timer that generates the tick, another timer source must be set up before entering the more efficient power states. Usually, the real-time clock (RTC) can be used for this purpose. However, some processors do not offer fine grain resolution with the RTC. For example, the RTC in the Cirrus EP7312 has a resolution of only one second, which obviously cannot be used to generate, say, a 100 Hz timer interrupt. There is also more overhead associated with managing the RTC to generate a periodic interrupt as it can not run in prescale mode. RTCs have large monotonically incrementing counters which, when compared against a match register, generate an interrupt. A

high resolution external timer source would solve the problem. But this is more expensive, requires more board space and ties up an interrupt pin on the SOC (of which there are few).

3. *PowerNap*: Details and Implementation

We now discuss *PowerNap* with frequent references to the implementation within the Linux operating system for predominantly idle mobile devices. With *PowerNap* we are able to resolve the limitations of the conventional Periodic Timing (PT) scheme. *PowerNap* is based on a timing scheme that eliminates the periodic timer tick whenever the OS is idling. We denote this scheme as the *Work Dependent Timing* (WDT) scheme and we say that the system is in the WDT mode whenever the OS is idling. When in the WDT mode, the system is only woken up when there is real work to be done, thus turning the OS into an event driven OS. In contrast, during periods of work (e.g., tasks are running), *PowerNap* switches into PT mode to ensure consistent updating of time and to support multi-tasking. From a software architecture view, *PowerNap* is a power management technique functioning within the scope of a full scale OS power manager.

3.1 PowerNap and Work Dependent Timing

Figure 1 shows a generic flow chart of *PowerNap*. (Note that some components are Linux specific). It is assumed the processor has two low power states, namely *Idle* and *ClockSuspend*. In Linux, whenever the current work item is suspended, the execution returns from the scheduler to the main, and infinite, idle loop. At this point *PowerNap* is in PT mode. In the idle loop the first thing *PowerNap* aims to resolve is, "*Is there more work to be done?*". If the answer is “*Yes*”, *PowerNap* remains in the PT mode and enters the *Idle* state while waiting for the next periodic tick. Usually, however, the answer is "*No*" which causes entry into the WDT mode of operation. In this mode the callback timer list is first examined to determine the nearest timeout value. The timeout value is then passed to the Power State Selector (PSS) routine in

which the optimal low power state is selected according to the rules described below in Section 3.3. Based on the particular state selection, the appropriate hardware timer is then selected and an associated timeout value calculated. *PowerNap* then reprograms the selected hardware timer with the timeout value and passes control to the Power State Transition (PST) routine.

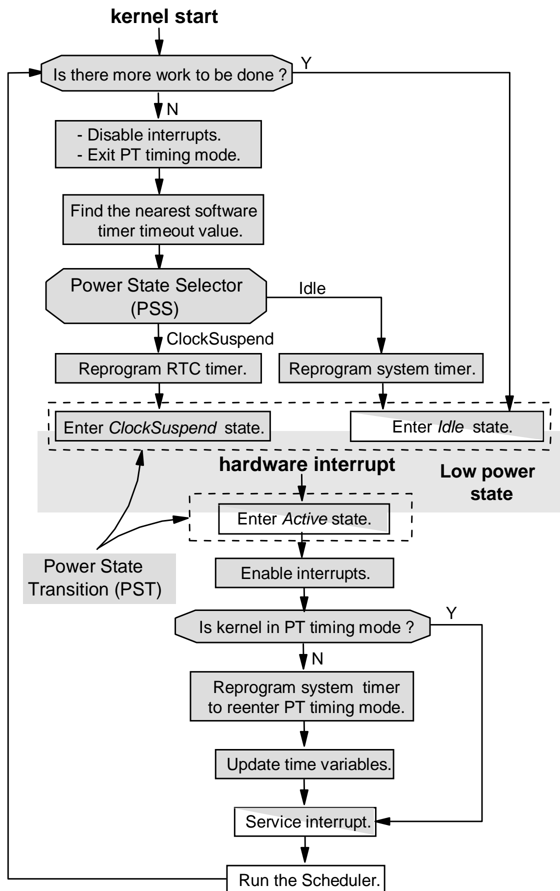

Figure 1. *PowerNap* flow chart. Gray boxes represent *PowerNap* extensions. White boxes represent conventional functions of the main idle loop. Boxes that are both white and gray can operate in either WDT or PT mode.

The PST routine transitions the processor and OS into the low power state, and upon detection of a hardware interrupt, it properly transitions the processor and OS out of the low power state and into the *Active* state where the CPU is running. Transitioning may be as simple as writing a bit in the register of the processor's Power Management unit on entry into the state which is the case for the *Idle* state. Entering *ClockSuspend* is more involved since it affects the state of externally connected devices, most importantly the DRAM which is usually put in self-

refresh mode. Entering a *PowerDown* state is even more involved and may require saving/restoring SOC state, flushing the cache, interacting with drivers for the state change, etc.

While in the low power state, all execution is stopped and the processor remains in this state until a hardware interrupt occurs. On exit from the low power state, the OS first determines which timing mode it is in, since the OS may have put the processor into a low power state either while in the PT mode or while in the WDT mode. If the system is not in PT mode, *PowerNap* then sets up the system timer to generate periodic timer interrupts while there is work to be done, and the OS reenters the PT mode. Note that PT mode is always in effect whenever there is process/task/device related work to be done since periodic updating of time and process variables is indeed required to preserve application semantics whenever there is work to be done. After reentry into PT mode, *PowerNap* updates *jiffies* and then kernel reference time (see Section 3.2.) At this point, regardless of the source of hardware interrupt, the OS now services the interrupt in regular fashion. On return from the interrupt handler, the scheduler is run.

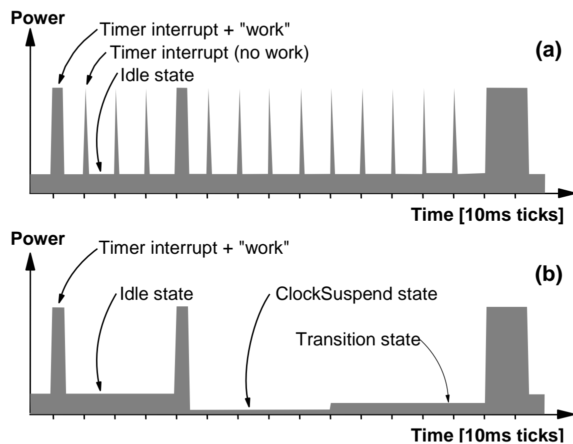

Figure 2. Illustration of the dynamic power consumption a) when in the PT mode and b) when in the WDT mode. The 50 ms duration of the transition state is a hypothetical value.

Figure 2 illustrates the effect of the PT and WDT modes on the dynamic power consumption. As seen, in the WDT mode all the workless timer interrupts are eliminated which creates extended idle periods. The *ClockSuspend* state is entered if the nearest timer callback timeout value is greater than 50 ms (the exit transition delay in this hypothetical example.)

3.2 Keeping time

Clearly, the method for keeping time in the WDT mode of operation cannot rely on a timer tick that is no longer present. Instead, time is maintained by reading a monotonically incrementing counter, such as a real-time clock (RTC) register. Whenever the OS detects an interrupt while in the WDT mode, the very first thing is to read the RTC time and subsequently update *jiffies* and kernel time. Thus, *jiffies* no longer governs time, as in conventional Linux. Rather, time governs *jiffies*. In the PT mode and on non-timer interrupts, *jiffies* and kernel time are updated in largely the same fashion.

3.3 Selecting the optimal low power state

The Power State Selector (PSS) routine selects the optimal low power state that reduces overall energy consumption while meeting timing constraints. For example, when exploiting the *ClockSuspend* state in the Cirrus Logic EP7312, the PSS must know how long it takes to exit the low power state in order to properly program the hardware timer to generate an interrupt that reflects the exit delay. Further, it must compare the exit delay to user or application demands to response time. If a user must press a touchscreen for more than 250 ms for the press to be registered (say, if the *ClockSuspend* exit delay is 250 ms), that may be regarded as unreasonable, since a pen press may be as short as 15 ms. In another example, when considering to use the *PowerDown* state in favor of the *ClockSuspend* state, the PSS must know how long it takes to transition in and out of that state and know how much energy is consumed during the transition. If too much energy is spent entering and exiting the *PowerDown* state compared with the energy savings experienced once in it, it may be better to remain in the less efficient *ClockSuspend* state. Finally, the PSS must know the resolution of the all available hardware timer resources as well as their phase relationships in order to calculate when a timer will generate an interrupt.

3.4 Kernel modifications for predictable latency

The assumption for calculating the optimal low power state is that the latency and energy associated with transitioning in and out of each low power state is predictable. To ensure this predictability, we had to modify the way the timer list is implemented. Currently Linux categorizes timers into five arrays according to their timeout value. Array 1 contains the timers with the earliest timeout values. The timer list is examined for expired timers for each tick the timer list is lagging behind the current count of *jiffies*. Usually, the lag is one tick. Secondly, on every 256th timer tick, the timers (if any) in the "spill-over" slot in array 2 are first removed and then re-added to the chain. This ensures that these timers get properly re-positioned into array 1.

When skipping timer ticks, the number of lagging ticks equals one plus the number of skipped timer ticks. Thus the larger the idle time, the more time is spent examining and reorganizing the timer list. Searching the arrays for the nearest timeout value adds even more overhead. On our test device in Section 4, we measured the overhead to be 370 μs/s (i.e., 370 μs for each second the OS idles.) The examination/reorganization of the timer chain and the searching for the nearest timeout value account for 90% and 10% of this overhead, respectively. Suppose the idle time is 60 s, this would amount to a delay of 22 ms. To eliminate the dependency on the idle time, we replaced the timer list with a single double-linked timer list where timers are inserted in order of increasing timeout value. The first timer in the list has the nearest timeout value. Finding the nearest timeout value (needed for programming the hardware timer) and retrieving expired timers is very fast and doesn't depend on the idle time or the number of timers. The only downside to our approach is that adding a timer to the timer chain can be slower than the array approach since the insertion time is proportional to the number of timers. However in systems with few tasks, and thus few timers, this is not a significant problem.

To further ensure the predictability of transition latency and energy, we also had to develop a suitable method for *PowerNap* to interact with device drivers. Interacting with drivers before transitioning into a low power state is a necessity since the driver may be in a state where shutting down its device, or disabling certain interrupts, is not acceptable to the driver. In this context, *PowerNap* allows the drivers to tell it which power states their devices support. The driver state is dynamically updated by the drivers so *PowerNap* instantly knows the power states it may include in the calculation. In essence our drivers are proactively power aware in contrast to the more common passively power aware drivers that don't deal with power issues until requested to do so. We will discuss the latter issue in more detail in Section 3.6 and in Section 9.

3.5 Patch size and overhead

To implement *PowerNap* in ARM Linux 2.4.2-rmk1-bluemug7 requires adding about 800 lines of code and removing about 800 lines of code as well.. The computational overhead is case dependent. With respect to the test device described in Section 4, the time to service a timer interrupt increased from 79 μs in the conventional PT based kernel to 100 μs (or 27% more) and 130 μs (or 65% more) with the *PowerNap* kernel in the PT mode and WDT mode, respectively. In the mostly idling device in Section 4, it is not unusual that more than 99% of the timer interrupts can be eliminated. So the "price" of spending an extra 65% time in the timer ISR in the WDT mode, is offset multi-fold by savings resulting from the elimination of the timer interrupts.

3.6 Impact on other software components

Modifications to other software components are required to enable optimal operation of *PowerNap*. Here we discuss illustrative obstacles experienced with IBM's WatchPad (Section 4.)

Device driver interactions: Device drivers interact with peripheral hardware devices such as the UART, the LCD controller, the synchronous serial interface (SSI), etc. These devices

are disabled in the *ClockSuspend* state, including their ability to generate an interrupt (if applicable). Therefore they cannot exchange data with the external devices they are connected to. See [12] for more details. To resolve the issue we introduced an API through which drivers can prevent *PowerNap* from using the *ClockSuspend* state, or other states that render devices non-functional, until they decide the devices are no longer needed. See Section 9 for more discussion.

Blinking cursor: Graphical user interfaces often have blinking cursors, or other animation, to catch the attention of the user, e.g., in a web browser's URL field or in the command line of a shell prompt. Cursors typically blink at 1 Hz, which means the screen needs to be updated two times per second. One way to implement this is to register a timer function for callback every 500 ms. In the case where it takes 220 ms to exit the *ClockSuspend* state in IBM's WatchPad, and where the RTC is only able to interrupt on whole second boundaries, this blinking effect renders the *ClockSuspend* state useless, and thus voids the chance of any significant battery life gains. Solutions to this problem include reducing the blinking period to, say, 2 s, and to let the user decide on the blinking period or to select a non blinking cursor.

Keyboard tasklet: Some unwanted effects are harder to predict. For example when we bring up X11 it opens a virtual terminal which keeps looking for a keyboard to be attached. A "tasklet" is put on a kernel queue to handle this inquiry. The tasklet is initially put into disabled mode. It remains in this mode until a keyboard is attached which will enable the tasklet so it can run and remove itself from the queue. On the test device we have no keyboard attached. Thus the tasklet remains permanently on the queue. Unfortunately, the tasklet queue is run on every timer tick as long as the queue is not empty. This causes the answer to the question, "*Is there more work to be done?*" in Figure 1 to be "*Yes*" which keeps *PowerNap* in the PT mode. We resolved the issue by disabling the initial queuing of the tasklet if no keyboard is attached at boot time.

Persistent kernel daemons: A number of daemons in the Linux kernel are scheduled to run with intervals of one second or more. But they can safely run with much larger intervals when the system is idling. For example, the kernel memory swap out daemon, *kswapd()*, is executed every 1 s. This will cripple the operation of the *PowerNap* scheme. We simply extended the interval to 30 s permanently for the experiments in Section 5 where the device is mostly idling. We did the same for the *bdflush()* daemon which writes out dirty and aged file buffers to disk. In practice, these intervals should be adjusted according to the system load.

RTC/software timer phase: In early experiments with the *PowerNap* based kernel on IBM's WatchPad, the measured average power would vary significantly every time the kernel was rebooted. By examining the dynamic power consumption, we noticed that sometimes the system would transition out of the *ClockSuspend* state prematurely and then remain in the *Idle* state for up to a whole second before executing a software timer callback function. The root of the problem was the phase of the RTC which can not be adjusted. For the sake of ensuring reliable power measurements, we adjusted the phase of long-term timers (i.e., which exceed 1 s) to coincide with the phase of the RTC clock. Short term timers and timers that are not a multiple of one second are not phase adjusted. In the WatchPad, the vast majority of timers fall in the long-term category. Proper adjustment of the timer phase optimizes the use of the RTC timer interrupt, increases the time spent in *ClockSuspend* state and maximizes battery life. We discuss the benefits and propose a method for adjusting the timer phase in Section 9.

4. Experimental Results

In this section we present measurements of the average power consumption on an embedded device, namely IBM's WatchPad, using both *PowerNap* and the conventional PT scheme and for varying computational loads. The IBM WatchPad [1] employs a Cirrus Logic EP7211 ARM based 32-bit RISC processor running at 18 MHz and which has 8 MB of DRAM

and a small LCD. The LCD remains on during all measurements (it consumes 1.8 mW.) In order to enable *PowerNap* to perform optimally, we implemented the kernel fixes discussed in Section 3.4 and 3.6. For fairness, the same modifications were made to the conventional PT kernel even though they have a near zero impact on the power consumption with the PT based kernel.

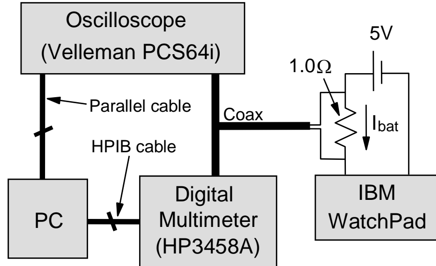

Figure 3. Experimental setup for measuring average power consumption of the test device.

4.1 Experimental Setup

Figure 3 shows the experimental setup used for measuring average system power consumption. The current consumption, I_{bat}, is found by measuring the voltage across a 1 ohm resistor inserted in series with the 5 V DC supply. The digital multimeter (DMM) measures the voltage with a resolution of 10 nV. The minimum current draw of the test device is around 500 μA. The DMM samples I_{bat} every 0.99 ms. A sampling time of 0.99 ms is not able to capture the instantaneous power consumption of every computing event. By virtue of sampling over an extended time, during which the computing events are sufficiently repeated, the occasional hit or miss of computing events will average out. This assumes the computing events are not phase-aligned with the sampling time. Since computing events occur at whole multiples of 10 ms and we are sampling every 0.99 ms, we have effectively eliminated this problem. The computer (PC) is used to collect data from and control the DMM. It is also used to collect data from a sampling oscilloscope for real-time display of the power traces on the PC's monitor. This gives us visual assurance that the test device is operating as expected, which is an invaluable debugging tool.

On the test device we run a program, *simm_load()*, to simulate a real task in a controlled fashion. *simm_load()* may be adjusted to run for any continuous length of time and to be scheduled with any periodicity. *simm_load()* repeatedly executes two loops, Loop1 and Loop2, within a master loop. Loop1 executes memory bound instructions for 75 μs, and Loop2 executes CPU bound instructions for 150 μs. During the memory and CPU bound periods, the average current consumption is 63 mA and 16 mA, respectively. The load function is executed as a timer callback function that can be adjusted to simulate different types of repetitive work loads.

4.2 Measurements

Table 2 shows key parameters measured on WatchPad. These are typical parameters that a system designer should measure to determine if *PowerNap* can extend battery life. We use the parameters in Section 5 to evaluate the accuracy of the analytical model of the battery life gain.

Parameter	Value
P_{active}: *active* power	155mW
$P_{pm,PT}$: *pm* power in PT mode (*Idle* state)	23.3mW
$P_{pm,WDT}$: *pm* power in WDT mode (*CS* state)	4.57mW
$t_{trans,PT}$: *trans* time in PT mode (*Idle* state)	0
$t_{trans,WDT}$: *trans* time in WDT mode (*CS* state)	220 ms
$P_{trans,WDT}$: *trans* power in WDT mode (*CS* state)	22.8mW
$f_{pops,0}$: Frequency of background timer pops	0.125 Hz

Table 2. Parameters measured on the WatchPad device.

The experimental procedure is as follows. The Linux kernel and X11 are loaded onto the WatchPad device. The relative computational workload is set to one of the following values, {0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.25, 0.5}. We consider 3 timer periods for executing this load corresponding to 1 minute, 3 s and 1 s. The corresponding timer pop frequencies of 1/60 Hz, 1/3 Hz and 1 Hz represent the granularity of the load. As an example, if the load is 0.03 (3%), then the load routine will run for a continuous period of 1.8 s once every 1 minute, for 90 ms once every 3 s or for 30 ms once every second, respectively. The PC then collects the sampling data

from the DMM over a period of 4 min. During this time, 242424 data points are collected. From this we calculate the average power consumed by the WatchPad device. The error in the measurement is less than 1%.

Figure 4 shows the gain in battery life achievable with *PowerNap* as a function of the computational load on the system and with the workload timer pop frequency as parameter, i.e., either 1/60 Hz, 1/3 Hz or 1 Hz. As seen the load timer pop frequency has a significant impact on the battery life gain. The reason is that the more fragmented the load is in time, the more the *ClockSuspend* exit transition energy is taking its toll on the total energy consumption. As the load decreases, the transition energy starts to dominate thus amplifying the effect of the timer pop frequency. Even for an infinitely small load, the processor has to wake up and transition out of the power state just to execute a couple of instructions. The case of the 1 Hz timer and a small load actually simulates the case of a slow blinking cursor. It demonstrates how important it is, on this device, to eliminate the blinking cursor as discussed in Section 3.6.

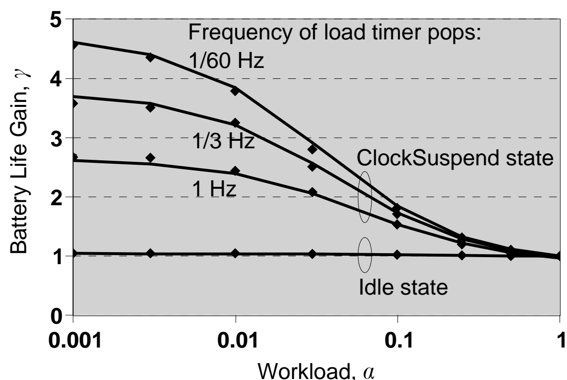

Figure 4. Battery life gain, γ, versus workload, α, obtained with *PowerNap*. Measured results are represented by diamond shaped markers and modeled results (from Section 5.4) are represented by solid lines.

For load timer frequencies below 0.1 Hz., the gain begins to saturate, and 1/60 Hz represents the maximum achievable gain. The reason for the saturation is the presence of the background timers which limit the effective timer frequency to 0.125 Hz (see Table 2). As expected, the smaller the load is, the more the system idles and the larger is the gain. As the load

increases beyond 5.5%, the power contribution from the load starts to dominate the smaller contribution from the low power state. This is true regardless of the timer pop frequency.

Also shown on the figure is the result of using only the *Idle* state (bottom curve and markers) but still skipping timer ticks. This is intended to simulate the case where a user informs the system that he wants to have, say, <100 ms response time. By implication, the *ClockSuspend* state cannot be used since it takes 220 ms to exit this state (see Table 2.) The gain in this case is independent of the load frequency since there is no transition energy penalty when exploiting the *Idle* state. As seen from the figure, when only using the *Idle* state, the gain from skipping timer ticks is minor. For example, for workloads smaller than 0.1%, the gain is 4.4%. However, in devices where large amounts of energy is consumed during the transition periods into and out of a more efficient low power state, such as a *PowerDown* state, there is indeed significant power savings to be gained due to *PowerNap*'s ability to eliminate transition energy of workless timer interrupts. As we shall see in Section 7, in devices that can exploit the efficient *PowerDown* state in both timing cases, the battery life gain may be 70%, or more, for certain devices.

5. Estimation of Battery Life Gain

In this section we shall first introduce a simple formula for estimating the gain in battery life which may be achievable with the *PowerNap* scheme over the conventional PT scheme. Then the formula will be used to model the battery life gains for various types of systems.

5.1 Modeling Power Consumption

The lifetime, T_{bat}, of an ideal battery with capacity, C_{bat}, supplying an average power of P_{avg} may be expressed as $T_{bat} = C_{bat} / P_{avg}$. In practice the C_{bat} is a function of P_{avg} [13]. This makes T_{bat} non-linear in P_{avg} and will be most dominant for large values of P_{avg}. Since, we are

concerned with mostly idling devices which consume small amounts of power most of the time, this effect is disregarded. Figure 5 illustrates variables to be used in the following.

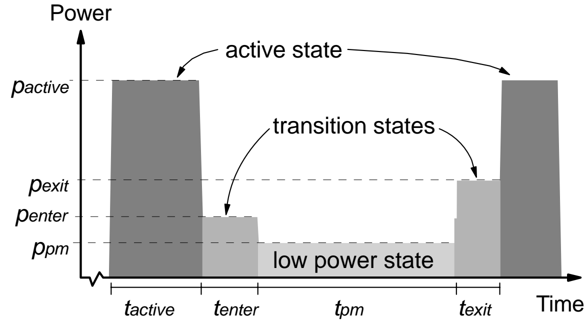

Figure 5. Illustration of variables associated with the dynamic power consumption.

P_{avg} can be expressed as

$$P_{avg} = P_{active} \bullet \tau_{active} + P_{trans} \bullet \tau_{trans} + P_{pm} \bullet \tau_{pm} \quad \text{(Eq. 1)}$$

In Equation 1 $\tau_k = t_k / (t_{active} + t_{trans} + t_{pm})$, *k∈{active, pm, trans}* is the relative time spent in the system states. In the *active* state the processor is executing instructions and memory may be accessed. In the *pm* state the processor and memory are in a low power state. In the *trans* state the processor is transitioning in and out of the *pm* state and memory may be accessed. P_k and t_k are the average system power and the average time spent in system state *k*, respectively. The *pm* and *trans* states have sub-states as there may be multiple *pm* states. We shall consider the case where each timing scheme uses only one *pm* state and one *trans* state. Based on our experiences with embedded systems this is a reasonable assumption. An embedded system may have several *pm* states but predominantly uses the *pm* state that yields the lowest power consumption and with which the system can still satisfy timing requirements. The associated time and power values are fairly easy to measure. Lastly we assume a long term average active power consumption for P_{active}. For a device running a given set of applications and usage scenario, this should be fairly easy to gauge. The *trans* time and *trans* power are expressed as

$$t_{trans} = t_{enter} + t_{exit} \quad \text{(Eq. 2)}$$

$$P_{trans} = (P_{enter} \bullet t_{enter} + P_{exit} \bullet t_{exit}) / t_{trans} \quad \text{(Eq. 3)}$$

where t_{enter} and P_{enter} are the transition delay and average power associated with entering the *pm* state while t_{exit} and P_{exit} are the transition delay and average power associated with exiting the *pm* state. To quantify the gain in battery life that may be achieved by using a WDT based timing scheme rather than a conventional PT scheme, we calculate the battery life gain

$$\gamma = T_{bat,WDT} / T_{bat,PT} = P_{avg,PT} / P_{avg,WDT} \quad \text{(Eq. 4)}$$

5.2 Zero Device Activity and Zero Transition Time

First we derive an expression for the gain in battery life assuming zero device activity, i.e., there are no asynchronous interrupts and no user related software timers running. Thus, in the WDT case the hardware timer is programmed to interrupt the processor at an infinite time, so τ_{pm} = 1.000. In the PT case, we assume the processor *Idle* state is used between ticks. As may be seen from Table 1, the transition time is so small that it may be ignored, so τ_{trans} = 0. We now make the analysis somewhat specific to WatchPad in Section 4 in that we use the measured duration of servicing a "workless" timer interrupt which is 0.079 ms. Since the kernel behaves in a highly repetitive fashion in the PT case, with a periodicity of $t_{active} + t_{trans} + t_{pm}$ = 10 ms, we get τ_{active} = 0.079ms/10ms = 0.0079. Thus, τ_{pm} = 0.992 is spent in the *pm* state. Using these values in Equations 1 and 4, the battery life gain becomes

$$\gamma = (P_{active} \bullet 0.0079 + P_{pm,PT} \bullet 0.992) / P_{pm,WDT} = (P_{active} / P_{pm,PT} \bullet 0.0079 + 0.992) \bullet (P_{pm,PT} / P_{pm,WDT}) \quad \text{(Eq. 5)}$$

If the same *pm* state, $P_{pm} = P_{pm,WDT} = P_{pm,PT}$, is used in both cases, Equation 5 reduces to $\gamma = P_{active} / P_{pm} \bullet 0.0079 + 0.992$ which represents the battery life gain associated solely with the elimination of the active power consumption of "workless" system timer ticks. As seen, we do not achieve significant battery life gains by eliminating workless timer interrupts while using the *Idle* state. We measured $P_{active} / P_{pm} = 6.7$ on WatchPad which according to Equation 5 should yield a 4.5% gain in battery life. The measured gain was 4.4% (see Section 4.2.) The only way to increase γ, while also increasing absolute battery life, is to reduce P_{pm}. Increasing P_{active} also increases γ but overall results in reducing the absolute battery life. We show elsewhere [12], that even with state-of-the-art components, at best, only a ~9% gain in battery life may be achieved.

A more interesting aspect of Equation 5 is when *PowerNap* enables the use of a low power state which could not be used in the PT case. The gain in Equation 5 is the product of two distinct gains; namely, the gain due to elimination of the active switching energy in workless timer interrupts and the gain represented by the factor, $P_{pm,PT} / P_{pm,WDT}$, which is due to exposure of a more efficient low power state. It is evident from Equation 5 that battery life gains may be significant if the power level in the WDT low power state is much smaller than the PT low power state. The results in Section 4.2 demonstrate this.

5.3 Non-Zero Device Activity

In reality, mobile devices will occasionally wake up to perform work such as on user interrupts, network interrupts, updating a clock, slow changing screen savers, daemon timer pops, etc. To get an idea of how much mobile devices really do idle, we examined our office computers. On laptops running Windows OS, 90-95% time is spent in the System Idle process. On desktop Linux systems we observed similar numbers for the Idle process. We believe that smaller devices such as PDAs and wearable computers, may spend even more time idling.

We shall introduce the workload parameter, α, which accounts for the relative time spent in the *active* state performing real work but excluding time spent in the timer interrupt handler. Thus α represents work triggered by the user, applications, network interrupts and OS daemons. The load due to timer interrupts is denoted as α_0. The total relative time spent in the *active* state becomes $\tau_{active,PT} = \alpha + \alpha_0$ in the PT case and $\tau_{active,WDT} = \alpha + \alpha \bullet \alpha_0$ in the WDT case where $\alpha \bullet \alpha_0$ accounts for the energy in periodic timer interrupts that occur while there is work to be done. Still assuming $\tau_{trans} = 0$ and using Equations 1 and 4, the battery life gain becomes

$$\gamma = (P_{active} / P_{pm,PT} \bullet (\alpha + \alpha_0) + 1 - \alpha - \alpha_0) /$$

$$(P_{active} / P_{pm,PT} \bullet (\alpha + \alpha \bullet \alpha_0) + P_{pm,WDT} / P_{pm,PT} \bullet (1 - \alpha - \alpha \bullet \alpha_0)) \qquad \text{(Eq. 6)}$$

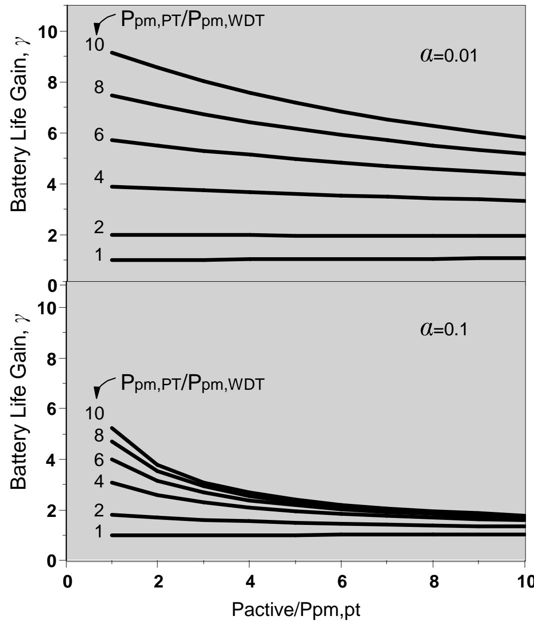

Figure 6. Battery life gain, γ, as function of $P_{active} / P_{pm,PT}$ for α_0 = 0.0079, workloads, α = {0.01, 0.1}, and with $P_{pm,PT} / P_{pm,WDT}$ = {1, 2, 4, 6, 8, 10} as parameter.

Figure 6 shows the battery life gain of Equation 6 for some select values of α and with $P_{pm,PT} / P_{pm,WDT}$ as parameter. As seen, for a given $P_{pm,PT} / P_{pm,WDT}$ the presence of a workload reduces the gain with increasing $P_{active} / P_{pm,PT}$, and the larger α is, the more pronounced is this

effect. The reason is that the larger the *active* power is relative to the PT *pm* power, the more relative energy is spent by the workload, and as $P_{pm,PT} / P_{pm,WDT}$ increases, γ simply becomes more sensitive to $P_{active} / P_{pm,PT}$. Nevertheless, it appears from Figure 6 that if in fact *PowerNap* exposes a more efficient low power state, even if this state is only twice as efficient, that for workloads below 10%, γ is still large enough to merit the adoption of the *PowerNap* scheme.

5.4 Non-Zero Transition Time

We now incorporate the transition time and energy into the analysis. A unit of transition energy is consumed on every interrupt that causes the processor to transition out of the low power state. In the WDT case, transition energy is strongly dependent on the nature of the workload, α. For example, assume a 1% workload executes for 0.6 s every 60 s. Then, only one unit of transition energy is consumed every 60 s. If however the workload is more fragmented, say it executes for 0.06 s every 6 s, then 10 times more transition energy is consumed than in the first case. So the transition time is a wasteful workload, just like α_0 is (due to timer interrupts).

Disregarding all non-timer interrupts, such as touch screen and key press interrupts, the time averaged relative transition times for the PT and the WDT cases may be approximated as

$$\tau_{trans,PT} = t_{trans,PT} \bullet f_{pops,0} \bullet (1-\alpha) \quad \wedge \quad \tau_{trans,WDT} = t_{trans,WDT} \bullet (f_{pops,load} + f_{pops,bg} \bullet (1-\alpha)) \qquad \text{(Eq. 7)}$$

where $f_{pops,<x>}$ is the timer pop frequency (i.e., the average number of timer pops per second) due to periodic timer interrupts (<x>=0), due to the workload (<x>=load), and due to the background tasks (<x>=bg.) Background tasks include kernel daemons and other smaller applications such as a clock application, which may always exist, and pop, regardless of the presence of an additional workload. Note that as α increases, the likelihood of a non-workload related timer pop occurring during the workload also increases. However, a timer that pops during the workload does not

give rise to wasted transition energy since the processor is already in the *active* state. Thus, the effective number of non-workload related timer pops is reduced by $(1-\alpha)$ as shown in Equation 7. This correction is valid for non-workload related timers that are independent of the workload.

Equation 7 mainly applies to bursty workloads which have small timer pop frequencies (smaller than the timer interrupt frequency.) For example, assume α is a 50% workload and that it is configured to execute on every timer tick (e.g., every 10 ms). In this case, the workload does not give rise to a reduction in the transition time since the execution never "bridges" across consecutive timer ticks. Thus, Equation 7 fails to accurately represent the impact of such a load. However, user workloads often don't get scheduled to run at fine grain intervals. Rather, user workloads tend to be bursty and bridge across several timer ticks, and often run to completion, or partial completion, before setting a "long term" timer before it runs again, or wait for the user to issue another command. Using Equation 7, Equation 4 can now be expressed more completely as

$$\gamma = \frac{P_{active} \bullet (\alpha + \alpha_0) + P_{trans,PT} \bullet \tau_{trans,PT} + P_{pm,PT} \bullet (1 - \alpha - \alpha_0 - \tau_{trans,PT})}{P_{active} \bullet (\alpha + \alpha \bullet \alpha_0) + + P_{trans,WDT} \bullet \tau_{trans,WDT} + P_{pm,WDT} \bullet (1 - \alpha - \alpha \bullet \alpha_0 - \tau_{trans,WDT})} \quad \text{(Eq. 8)}$$

With Equation 8, it is now possible to determine the potential battery life gains of the test device in Section 4. We use the measured parameters listed in Table 2, let the workload range from $\alpha \in$ {0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.25, 0.5, 1} and consider three values of the workload timer pop frequency, $f_{pops,load} \in$ {0.0167, 0.3333, 1.0} Hz (or 1, 20 and 60 timer pops per minute, respectively.) The results are shown in Figure 4 (solid lines). As may be seen there is good agreement between measurements and modeled results. It indicates that the designer may accurately estimate battery life gains with *PowerNap* using our modeling methodology.

6. Selecting the Optimal Power State

We now present the Power State Selector (PSS) routine and apply it to a realistic case.

6.1 Power State Selection Routine

We denote the time and average power spent in power state *i* as $t_{pm,i}$ and $P_{pm,i}$, and the time and power spent in the associated transition state as $t_{trans,i}$ and $P_{trans,i}$. $i \in \{1,N\}$ and *N* is the number of power states. The PSS routine first selects the states that satisfy the latency criteria

$$t_{trans,i} < \min\{t_{idle}, t_{response}\}, \quad t_{idle} = t_{trans,i} + t_{pm,i} \qquad \text{(Eq. 9)}$$

t_{idle} is the maximum time the system may idle and $t_{response}$ is the user/application specified maximum response time. Equation 9 states that power state *i* is a legal state to use if the total transition time of power state *i* is less than both the idle time and the response time. Note that the effective time spent in the power state, $t_{pm,i}$, is reduced by the state transition time. Having now identified the legal power states, PSS determines the total energy consumption of each state as

$$E_{total,i} = E_{trans,i} + E_{pm,i} = P_{trans,i} \bullet t_{trans,i} + P_{pm,i} \bullet t_{pm,i} \qquad \text{(Eq. 10)}$$

The optimal low power state is the state that satisfies

$$E_{total,i} < E_{total,j}, \; j \neq i \qquad \text{(Eq. 11)}$$

We anticipate that, in many cases, the only parameter in Equations 9-11, that may change dynamically, is the idle time, t_{idle}. Thus, it is possible to calculate the boundaries for t_{idle} at which the optimal low power state changes. These boundaries may be calculated during OS boot or during a reconfiguration step and stored in an array. In turn, on every reentry into the OS idle loop, PSS can quickly select the optimal state by comparing t_{idle} with the boundary array.

6.2 PowerDown versus ClockSuspend State

We now determine the sleep, or idle, time, t_{idle}, at which the *PowerDown* (PD) state becomes the optimal state and where the only competing state is the *ClockSuspend* (*CS*) state. The analysis is independent of workload and is not specific to small form-factor devices.

Applying the power ranges in Table 1 to Equations 9-11, the following approximations can be made. First, the size of $t_{trans,CS}$ is very small and for idle times of 10 ms, or larger, $t_{pm,CS}$ >> $t_{trans,CS}$ in Equation 9. In conjunction with our observation that $P_{trans,CS}$ is only slightly larger than $P_{pm,CS}$, it is safe to ignore $E_{trans,CS}$ in Equation 10. Last, we point out that the only difference between the system power consumption in the *ClockSuspend* and *PowerDown* states is the power consumed by the SOC. From Equations 9-11, using the above approximations and assuming $P_{soc,pm,PD} << P_{soc,pm,CS}$, we can derive the following compact expression for the threshold idle time

$$t_{idle} > t_{idle,th} = (E_{trans,PD} - P_{pm,PD} \bullet t_{trans,PD}) / P_{soc,pm,CS} \qquad \text{(Eq. 12)}$$

Equation 12 says what the threshold idle time, $t_{idle,th}$, the actual idle time, t_{idle}, must be greater than before it pays of to use the *PowerDown* state. Figure 7 shows the value of $t_{idle,th}$ as a function of the SOC's power consumption in the *ClockSuspend* state, $P_{soc,pm,CS}$, and for select values of the *PowerDown* transition energy, $E_{trans,PD}$. For a given value of $E_{trans,PD}$, in the region above the curve it is better to be in the *PowerDown* state and in the region below it is better to be in the *ClockSuspend* state. The range we chose for $E_{trans,PD}$ is partly based on the analysis in [2] which assumes the SOC state is saved and restored by software. Larger energies can be envisioned. The figure shows the threshold idle time decreases as $P_{soc,pm,CS}$ increases and as $E_{trans,PD}$ decreases. OSs such as Windows and Linux typically use a timer interrupt of t_{HZ} = 10 ms. The figure shows that for *ClockSuspend* power levels below 2 mW and transition energies

above 25 μJ, it is not economical to use the *PowerDown* state in such PT based OSs. In other words, using the *ClockSuspend* state instead would produce an overall lower energy consumption. However, with a WDT based OS, where idle periods can easily exceed 100 ms , it may be quite possible to exploit the *PowerDown* state for increased battery life gains.

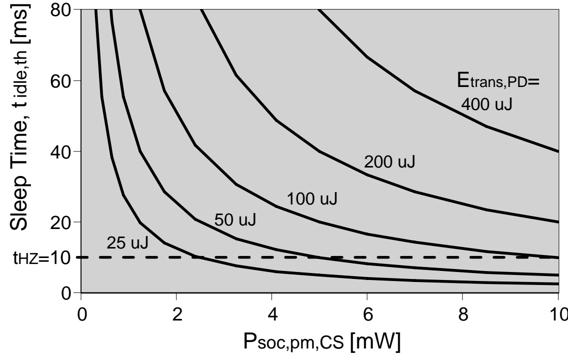

Figure 7. Sleep time, t_{idle}, versus processor *ClockSuspend* power. Curves indicate the idle time at which the total energy associated with exploiting the *PowerDown* state equals the total energy of the *ClockSuspend* state.

7. Impact of *PowerDown* Transition Energy

SOC leakage power keeps increasing in every new process technology release. This is due to the shrinking feature size and lowering of the threshold voltage to accommodate smaller supply voltages [8,9]. According to Figure 7, as the *ClockSuspend* SOC power grows it becomes increasingly more likely that the *PowerDown* state can be exploited with advantage with a PT based OS. There is, however, wasteful transition energy associated with using the *PowerDown* state. This section evaluates the battery life gain achievable with *PowerNap* which effectively eliminates the repetitive transition energies associated with workless timer interrupts.

We shall use Equation 8 and consider a hypothetical device which employs a state-of-the-art 0.13 um 1 V SOC and a 1.8 V SDRAM memory [14]. The SOC *PowerDown* power in such a device is quite small (<200 μW from Table 1). Therefore, the long term average system background power, $P_{sys,offset}$, will have a negative impact on any potential battery life gain. $P_{sys,offset}$

is defined as the accumulated power contributions from all other system components, except the memory power in the *active* and *trans* states and except the SOC power. Typical contributors to $P_{sys,offset}$ include DRAM self-refresh power, display and audio power, network interface, and power supply loss. With respect to Equations 1 and 8, $P_{sys,offset}$ is included in the P_k power contributions. To proceed, the parameter values in Table 3 are used. Equations 1 and 2 from [2] are used to obtain approximate values of the *PowerDown* transition energy and latency.

$P_{soc,active}$ @ 100MHz, 1V	25mW
Oscillator stabilization time: t_{osc}	3ms
Memory burst power: $P_{mem,burst}$ @ 100MHz, 1.8V	162mW
Memory utilization in *active* state: μ_{mem}	25%
Memory power in *active* state: $P_{mem,active}=P_{mem,burst} \bullet \mu_{mem}$	41mW
$P_{mem,clock\text{-}gating}$ & $P_{mem,self\text{-}refresh}$	<0.5mW
Timer interrupt frequency	100Hz

Table 3. Parameters used for modeling power consumption in a hypothetical small form-factor device using typical data for 0.13 um 1V SOCs and a 1.8 V mobile-SDRAM [14]. Other parameters used are listed in [2].

We can now express P_{active} in Equation 8 as $P_{active} = P_{soc,active} + P_{mem,active} + P_{sys,offset}$. From Equation 8 it may be seen that for small values of $P_{sys,offset}$ the *pm* power dominates the average power in the WDT case while the *trans* power dominates in the PT case. Figure 8 shows the battery life gain as a function of the background power and with the workload, a, as parameter. As seen, for a mostly idling device (say for a < 3%) and for an efficiently designed small form-factor mobile device (say for $P_{sys,offset}$ < 5 mW), a 71% improvement in battery life can be obtained. For $P_{sys,offset} = 10$ mW the improvement drops to 42%. Figure 8 is valid for $E_{soc+mem,trans,PD} = 50\ \mu J$ which according to Equation 1 in [2] may be incurred for the values in Table 3, 64 MB SDRAM, and a processor with 16 KB D-cache, 2KB SOC state and 7 mW leakage power (may be observed at higher temperatures.) The leakage power significantly impacts the *PowerDown* transition energy during oscillator stabilization [2].

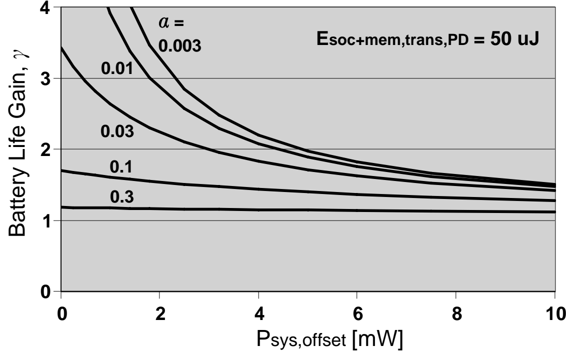

Figure 8. Battery life gain, γ, versus background power, $P_{sys,offset}$, due to elimination of transition energy and for various workloads, α. $E_{soc+mem,trans,PD}$ =50 μJ, $t_{trans,PD}$ =3.2 ms, α_0 = 0.0014 and $f_{pops,\{0,bg,load\}}$ ={100,7.5/60,1/60} Hz.

SOCs may have smaller or larger D-caches, have more or less state that needs to be saved, and have more or less leakage power which all affect $E_{soc+mem,trans,PD}$. Figure 9 shows the battery life gain for α = 3% and for some select values of $E_{soc+mem,trans,PD}$. Assuming a worst case *PowerDown* transition energy of 100 μJ, the battery life gain is x2.40 at $P_{sys,offset}$ = 5 mW and x1.82 at $P_{sys,offset}$ = 10 mW. According to Equation 1 in [2], a transition energy of 100 μJ may be incurred for a SOC with 32 KB D-cache, 4 KB SOC state and 15 mW leakage power. Note that neither the load timer pop frequency, the *PowerDown* transition time nor the power level of the *PowerDown* state have any significant impact on the results in Figures 8 and 9.

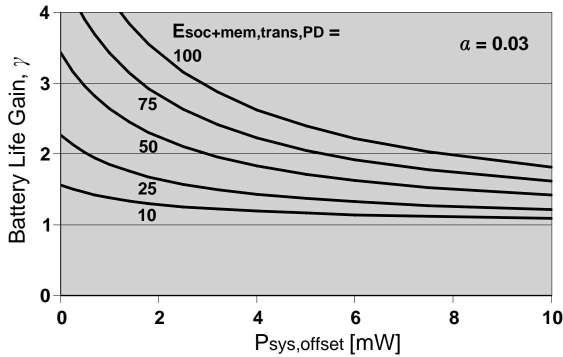

Figure 9. Battery life gain, γ, due to elimination of transition energy. γ is shown as a function of the background power, $P_{sys,offset}$, and with the transition energy, $E_{soc+mem,trans,PD}$, as parameter. Other assumptions include α = 0.03, $t_{trans,PD}$ = 3.2 ms, α_0 = 0.0014 and $f_{pops,\{0,bg,load\}}$ = {100,7.5/60,1/60} Hz.

8. Related Work

Advanced Configuration and Power Interface (ACPI) [15] is probably the most widely known power management interface. However, ACPI is not an OS power management technique. Rather, it enables the processor and connected devices, such as a display and a hard-drive, to register power management capabilities with the OS. This enables the power manager and the drivers to manage the power state and disable the devices when certain criteria are met, e.g., when a period of device inactivity has elapsed. In principle, *PowerNap* could use ACPI.

WinCE [16] allows the kernel to skip timer ticks according to the next scheduled computing task and if there are currently no active tasks/threads executing. From [16] there is no indication that WinCE takes the transition time and transition energy required to move to a low power state into account to determine when to use a more efficient low power state. Nor does it appear that WinCE has the capability to exploit more than one low power state, or that WinCE can exploit the most efficient low power state among two or more low power states.

RT-Linux [17] is implemented as a real-time kernel situated underneath Linux. Linux itself runs as a low-priority preemptible non-real-time task which is interrupted every 10 ms. The similarity between the real-time part of RT-Linux and *PowerNap* is that timer events are separately scheduled by programming a one shot timer. As far as we can tell the comparison holds true for other real-time OSs as well. Our WDT scheme differs from the above by only eliminating timer ticks during idle periods and by switching to PT mode during active periods.

Operating systems such as PalmOS, VxWorks, and REAL/IX use periodic timer interrupts, but some allow for variation in the frequency of the timer interrupt. Presumably one could increase the periodic timer interrupt interval from 10 ms to larger values so that the device could spend a longer time in a lower power state in between. But this will reduce the responsiveness of the system. In fact we attempted this as a first solution but abandoned it due to

complications in resolving side effects. For one thing, software timers are based on the assumption that the timer interrupt interval is constant. Also, several portions of Linux seem to have the 10 ms interval inherently assumed into its code base. Further, the timer interval can't be made larger than 100 ms in Linux. Changing the interval to 100 ms is not going to help much because it is much too small in comparison to practical idle periods which may easily exceed 1 s.

In [18] it is pointed out that most mobile computers are event driven and that the best power policy is to put the system into a low power state between events and wait for the next event. Their analysis, however, strictly assumes an X-server running on a dedicated processor and the decision to shutdown is based on prior history of the X-server process state. They do not take into consideration the impact of an OS and associated issues with timers, RTC, driver interactions, etc. They do discuss overhead associated with shutting the system down and waking it up and that the time spent in the low power state has to be great enough to compensate for the overhead. The magnitude of this overhead is only discussed superficially. They also don't address the predictability of the latency and energy of the overhead.

We can also report that our technique from [1,12] has already been implemented in [19] on a StrongARM-based device. This verifies the broad ability of our technique to save power.

Finally, extensive work [20-27] has been carried out in the area of *active* power management where the aim is to reduce energy consumption during the execution of tasks.

9. Discussion and Future Work

Other power managers: As mentioned in Section 3.4 and 3.6, device drivers convey their state to *PowerNap* enabling it to quickly determine the optimal power state. In contrast, in conventional Linux (if configured to use power management) and in the Windows OSs (including WinCE), power managers query the state of the device drivers just prior to transitioning into the power state [16,28,29]. Drivers are asked if the prospective loss of

functionality is OK. If all drivers reply "*Yes*", the manager sends a second signal instructing the drivers to prepare their devices for entering the power state. If any driver replies "*No*" to the initial query, the manager sends a second signal to the drivers to remain in the current power state. This can be a quite cumbersome process due to the potentially many drivers. Second, since the power manager has no control of what happens in the device drivers, there is no guarantee of how long the transition in and out of a low power state will take. Third, the power managers use OS signaling methods (e.g., IOCTL calls, I/O request packets) to communicate with the drivers. This results in further loss of control. In the end, a State Selector routine would not be able to calculate the optimal power state reliably since 1) it can't determine if the next scheduled event can be met, and 2) nor does it know, reliably, how much energy is associated with the transition.

***PowerNap* API:** Even though potentially detrimental to the State Selector, there may be times when *PowerNap* must relinquish control to a driver. This may happen if the power state transition routines can not guarantee that the context of a hardware component is sufficiently restored on wakeup. This situation is most likely to occur when exploiting states where the hardware context is lost and where the content of device registers must be saved in memory to enable subsequent restoring of the context. The context of the driver itself does not need to be saved since all software state is maintained (and D-cache is written out.) It is quite common that the way a device is being initialized is by configuring hardware registers. There is no reason why the driver couldn't point *PowerNap* to the relevant register and context structure to do the initialization. Furthermore, it may not always be required to restore the context of, or initialize, a device on every wakeup event if the device is not currently in use. This should be done on an on-demand basis. Though interruptible devices may require partial restoring of the context.

The point is that the minimal should be done, and that it should be done by the power manager, to facilitate a high level of control of, and to minimize, the transition latency and

energy. We plan to extend the *PowerNap* API to facilitate drivers to indicate what they allow the kernel to do on their behalf and where the device context and initialization structures are located.

Timer pooling: By pooling together software timers, we can further increase the time spent in a low power state and further reduce transition energies. To enable timer pooling, the OS must be allowed to modify the timers and the calling software module must supply parameters that the OS can work within. For example, by supplying a parameter set that presents timing ranges within which the OS may phase align and change the periodicity of the timer, the number of timer pops may be significantly reduced. Assume a system has two software timers, TMR1 and TMR2. Suppose TMR1 expires every 3 s, TMR2 expires every 5 s and that initially the timers are randomly kicked off. In the worst case, the timers may cause the system to wake up 60/3 + 60/5 = 32 times/min. Suppose the timers may be phase aligned within +/-1 s. This ensures that at least once every 3x5 s = 15 s they pop simultaneously and reduces the number of timer pops from 32 to 28 pops/min. Assume further that each timer has a +/-1 s range for the periodicity. This makes it possible to extend TMR1's period from 3 s to 4 s, and reduce TMR2's period from 5 s to 4 s, effectively reducing the timer pops to 60/4 = 15 times/min. The notion of timer pooling was also suggested in [19], though they limit the discussion to aligning timers which have identical periodicity and they don't discuss modifying the periodicity.

Supporting predictive user events: *PowerNap* will always exploit the most optimal power state as long as it satisfies Equation 9 which assumes random user events. If however user, or network, events can be somehow predicted, it may be possible to use a more efficient power state by ignoring $t_{response}$, for example by making it infinite. Thus *PowerNap* may be used in the context of predicting events. Of course, some other piece of software must determine the wake-up events and modify the OS timer list accordingly, which falls outside the scope of this paper.

10. Conclusion

We presented the efficient *PowerNap* OS power management technique. The foundation of *PowerNap* is the Work Dependent Timing scheme of the OS which skips timer ticks whenever the OS is idling and which uses the most optimal low power state at any given time. We discussed an implementation of *PowerNap* under Linux and showed associated battery life gain over a Periodic Timing scheme experimentally and through modeling. The implementation imposes a minor increase in code size and results in minimal computational overhead. In order to use our technique in Linux, some device drivers had to be modified, but we believe the changes are minimal. Our method can be combined with other power management schemes such as frequency and voltage scaling, application level power management, ACPI, etc.

PowerNap is tightly integrated with the core kernel. It is designed to have optimal control of the transitioning into and out of low power states. This is a key requirement in determining reliable values of the latency and energy associated with transitioning in and out of the low power states. No other power management technique offers this level of control and efficiency.

There are three power benefits of *PowerNap*. The most obvious benefit is the elimination of the *active* energy associated with executing workless timer interrupts. The analysis showed however that only 9% may be gained in battery life from this. A more subtle benefit arises from the extended time the processor may idle when skipping workless timer interrupts which may expose a more efficient low power state. Our experiments with IBM's WatchPad showed a x4.6 gain in battery life due to the exposure of the *ClockSuspend* state. The last benefit of *PowerNap* is in devices using processors that have a low power state, such as the *PowerDown* state, in which the transition energies are significant. This will be of relevance to future devices which are likely to deploy SOCs which have a *PowerDown* state. We predict that *PowerNap* can eliminate the relatively large transition energy associated with the repetitive transitioning in and out of the

PowerDown state. For example, for an efficiently designed small form-factor device that is mostly idling, the battery life gain may exceed 42% for background power levels below 10 mW.

We introduced a simple analytical model that may be used by mobile device designers to determine whether *PowerNap* would provide a worthwhile increase in the battery life for the device they are building. The model showed good agreement with the experimental results.

References

[1] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, C. Narayanaswami, "*Energy trade-offs in the IBM Wristwatch computer*," International Symposium Wearable Computing, pp. 133-140, 2001.

[2] C.M.Olsen, B. Brock, R. Snyder, M.Ware,"*Analysis of Transition Energy and Latency of the PowerDown State in Advanced System-On-a-Chip Processors*," IBM Research Report RC22970, Nov. 2003.

[3] Cirrus Logic, "EP7211: Data sheet," May 1999.

[4] Hitachi, "SH7750 series: Hardware manual, "July 2002.

[5] NEC, "Vr4181: User's manual," Sept. 2000.

[6] Intel, "*Intel PXA250 and PXA210 Applications Processors: Developer's Manual*," Feb 2002.

[7] K.J. Nowka et al., "A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling," IEEE J. Solid State Circuits, Vol. 37, No. 11, pp. 1441-1447, Nov. 2002.

[8] N.S. Kim et al., "*Leakage Current: Moore's Law Meets Static Power,*" IEEE Computer, pp. 68-75, Dec. 2003.

[9] S.M. Sze, "*Semiconductor Devices: Physics and Technology,*" J. Wiley & Sons, 1985.

[10] Cirrus Logic, "EP7312: Data sheet," May 2002.

[11] Intel, "*Intel StrongARM SA-11100 Microprocessor for Portable Applications*," Brief Data sheet, April 2000.

[12] C.M. Olsen, C. Narayanaswami, "*A Work Dependent OS Timing Scheme for Power Management: Implementation in Linux and Modeling of Energy Savings*," IBM Research Report RC 22784, April 2003.

[13] D. Linden, "*Handbook of Batteries*," 2nd Edition, McGraw Hill, 1994.

[14] Micron, "*256Mb: x32 Mobile SDRAM,*" MT48H8M32LF Advance Datasheet, 2003.

[15] Compaq, Intel, Microsoft, Phoenix, Toshiba Corporations, "Advanced Configuration and Power Interface Specification," Rev.2.0c, August 2003.

[16] http://msdn.microsoft.com/library. Search for "enabling power management".

[17] V. Yodaiken, M. Barabanov, "*A Real-Time Linux*," Proc. of USENIX Annual Tech. Conf., 1997.

[18] M. B. Srivastava, A.P. Chandrakasan, R.W. Brodersen, "*Predictive System Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation,*" IEEE Trans. VLSI Systems, Vol.4, No.1, p.42, March 1996.

[19] L.S. Brakmo, D.A. Wallach, M.A. Viredaz, "*uSleep: A Technique for Reducing Energy Consumption in Handheld Devices,*" 2nd Intl. Conference on Mobile Systems, Applications, and Services (MobiSys), June 2004.

[20] J. Flinn, M. Satyanarayanan, "*Energy-aware adaptation for mobile applications*," 17th ACM Symposium on Operating Systems Principles, pp. 48- 63, 1999.

[21] H. Zeng, X. Fan, C. Ellis, A. Lebeck, A. Vahdat, "*ECOSystem: Managing energy as a first class operating system resource*," Proc. ASPLOS, pp. 123-132, October 2002.

[22] M. Weiser, B. Welch, A. Demers, S. Shenker, "*Scheduling for Reduced CPU Energy*," Symp. on Operating Systems Design and Implementation, pp. 13-23, 1994

[23] D. Grunwald, P. Levis, C. Morrey III, M. Neufeld, K. Farkas, "*Policies for dynamic clock scheduling*," Symp. on Operating Systems Design and Implementation, pp 78-86, Oct 2000.

[24] Y.-H. Lu, L. Benini, G. D. Micheli, "*Low Power Task Scheduling for Multiple Devices*," International Workshop on Hardware/Software Codesign, pp. 39-43, 2000.

[25] A Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, C. Yoshikawa, "*WebOS: Operating System Services for Wide Area Applications*," Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems, pp. 52-63, July 1998.

[26] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, "*The Case for Cyber Foraging*," Proc Tenth ACM SIGOPS European Workshop, Sep 2002.

[27] J. Lorch, A. J. Smith, "*Software Strategies for Portable Computer Energy Management*," IEEE Personal Communications Magazine, Vol 5 No 3 pp. 60–73, June 1998.

[28] S. Vaddagiri, A.K. Santhanam, V. Sukthankar, M. Iyer, "*Power Management in Linux-Based Systems,*" Linux Journal, March 2004. (http://www.linuxjournal.com/article.php?sid=6699)

[29] W. Oney, "*Programming the Microsoft Windows Driver Model,*" 2nd Edition, Microsoft Press Press, 2003.