RC22784(Declassified 07/09/2003) (W0304-187) April 21, 2003
Computer Science

IBM Research Report

A Work Dependent OS Timing Scheme for Power Management: Implementation in Linux and Modeling of Energy Savings.

Claus Michael Olsen, Chandrasekhar Narayanaswami
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

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

A Work Dependent OS Timing Scheme for Power Management: Implementation in Linux and Modeling of Energy Savings

C. Michael Olsen and Chandra Narayanaswami
IBM Research Division, Hawthorne, NY 10532
{cmolsen,chandras}@ us.ibm.com

Abstract - *We present a Work Dependent Timing (WDT) scheme for the Linux operating system that skips periodic system timer ticks when the system is idle. Our scheme parses the internal lists and queues in Linux to determine when the next work item is due and eliminates "workless" timer ticks that solely update system time. Subsequently, we program the hardware timer with the timeout value of the nearest work item and then put the processor/system into an optimal low power state. Furthermore, when skipping timer ticks, the opportunity may arise to exploit a more efficient low power system and processor state that would not be possible with a conventional periodic timing scheme because periods where the system can sleep may now exceed the time to transition into and out of the low power state. We describe the implementation of the WDT scheme in detail and discuss its impact on system software. Experimental results with an embedded system verify the ability of the WDT scheme to extend battery life. An analytical power model that quantifies the ability of the scheme to reduce system power consumption is presented. The model is in good agreement with the experimental results. Also included in the model is a power state selection algorithm which, given full knowledge about the system power levels, transition times and timer events, can dynamically calculate the optimal low power state to exploit at any given time.*

1. Introduction

Power management has become one of the most urgent challenges in mobile devices. It is being investigated and implemented at the circuit level, processor architecture level, in circuit interconnections, and in the OS and applications. The power management approaches largely fall into two categories, namely, active and passive. In the active category, the aim is to reduce the energy required to complete a task. In the passive category, the aim is to put devices, including the processor, into a low power state while not in use. Our paper falls mostly into the passive category but also addresses a number of active issues

In this paper we shall present an operating system (OS) technique, called Work Dependent Timing (WDT) which aims to utilize a processor's resources and power states more judiciously. Specifically, we will build on a technique presented by Kamijoh et al. [1] that modifies the timing mechanism of the OS. This mechanism applies to a class of general purpose mobile devices that are mostly idle but need to be able to turn on instantly and handle multiple applications when they are active. On these systems, for a majority of the duration, there are no actively running applications, no open network connections, and no user interaction. Interrupts are enabled, so external events can wake up applications. Due to these constraints, such "instant on" devices can have the display turned off, the DRAM in self-refresh mode, the audio device disabled, and the processor in a low power state for a large fraction of their life. However, when active, several applications can run, and this requirement directs us towards multitasking operating systems that wake up periodically. To minimize the cost of software development the system designers are attracted to commonly available general purpose operating systems, such as Linux, due to the widespread availability of device drivers, applications, and systems programmers.

Our motivation comes from optimizing power consumption for mobile devices when they are idle and not for minimizing power consumption while the device is active - which is the subject of several papers on voltage and clock frequency scaling. Our techniques are also useful for other predominantly idle devices in the environment; such as equipment in conference rooms, offices, kiosks, etc., because conserving ac power, while often ignored, is good both for the environment and economy.

There are three goals for this paper. The first goal is to describe the detailed implementation of the WDT scheme under Linux and to discuss the impact on system software components and the usability of the system. The second is to propose a methodology that a mobile device designer can use to estimate the potential benefits of deploying the WDT scheme on his/her mobile device. The third is to present an algorithm to dynamically determine the optimal low power state to exploit at any given time.

2. Current Processors and Linux

We now describe the low power states that are available in modern system-on-chip (SOC) processors used for embedded mobile devices. Then we discuss the conventional Periodic Timing (PT) scheme used by the standard Linux OS and show why it limits power savings.

2.1 Processor Power Management States

Advanced mobile SOC processors have multiple Power Management states. Some of the processors that we have studied include Cirrus Logic EP7211/7312 [2,3], Intel SA1110 [4], Intel PXA250/210 [5,6], Motorola DragonBall MXL [7], Hitachi SH7705 [8], NEC VR4131 [9], and IBM PowerPC 405LP [10]. For a brief survey of some of these processors see [11].

In general the way these low power states are implemented differs somewhat from processor to processor. For simplicity we shall assume that a processor has three distinct power states, as outlined in Table 1.

Power State	CPU/Periph. Clock	Power [mW]	Transition state [us], [uJ]
Idle	Off/On	>5	0.1-0.01,>0.001
ClockSuspend	Off/Off	0.25-5	>100, >0.5
PowerDown	Off/Off	0.05-0.2	>250, >25

Table 1. Definition of processor low power states and characteristics associated with each state. The "Power" column indicates minimum power levels and the "Transition state" column indicates minimum total time and energy required to enter and exit the state. The values are representative of state-of-the-art 32-bit mobile processors.

In the *Idle* state the clock to the CPU core is stopped, but other peripheral on-chip resources remain actively clocked. All of the above mentioned processors have this state. Some of the processors [2,8,9,10] also have a global *ClockSuspend* (CS) state in which the clock is globally stopped to most peripheral cores such as SDRAM controller, DMA, LCD controller, UART, etc. The only cores that remain active are the Power Management unit, the Real-Time Clock and the Interrupt Controller unit. The logical state in the various cores is preserved. One of the drawbacks with exploiting this state is that it disables cores such as the LCD controller and the communications functions such as UART, USB and SSI interfaces which are asynchronous in nature. To keep these interfaces fully functional and to be able to either display a static image on an LCD or accept incoming data, they must be clocked. Furthermore, it takes longer to exit this state due to the fact that the PLL has to stabilize (100-200 us) upon wakeup. At least one processor [2] takes up to 250 ms to exit this state.

Quite recently, a couple of advanced high-speed processors have been emerged [5,10] which have a *PowerDown* state in which power is removed from the CPU core and where either the power is removed, or at least the clock is stopped, to most of the peripheral cores. The Power Management unit, the Real-Time Clock and the Interrupt Controller unit remain active to enable fast wake-up and to maintain time. The importance of this power state is that it offers significantly lower power consumption than the *ClockSuspend* state and that it is not sensitive to temperature. In addition there are signs that the *ClockSuspend* power is rising which is a tradeoff that is paid for faster and more complex circuits.

One of the drawbacks with exploiting the *PowerDown* state is that all processor logical state and cache content is lost when powering off the processor. So, CPU and peripheral state that is not available elsewhere in the system would have to be saved before entering the *PowerDown* state. It takes time to save and restore the processor context (250 us at minimum). But even more importantly, it is relatively expensive in terms of transition energy, which we shall discuss in more details in Section 6.

Table 1 shows a substantial difference in power consumption between the three states, though it varies between processors. When entering the *Idle* state, the peripheral bus frequency may be reduced to minimize switching power dissipation in the peripherals. This is however not possible in all processors. For example, the Cirrus EP7312 consumes 45 mW @ 18 MHz in the *Idle* state while the Intel PXA250 consumes 20 mW @ 20 MHz (scaled down from 400 mW @ 400 MHz [6].) Even though the bus frequency can be further reduced in some processors (e.g., Motorola's DragonBall MXL can go down to 0 Hz), the power consumption in the oscillator and PLL will limit power consumption to ~5 mW. In contrast, in the *ClockSuspend* state, the Cirrus EP7312, for example, consumes around 350 uW @ 25C [3]. The power consumption in this state is mainly due to DC leakage currents, which are strongly temperature dependent. At 70^{O}C the *ClockSuspend* power may go up by a factor of five. There is usually a small energy and delay associated with transitioning out of the *ClockSuspend* state, due to the time it takes the PLL to stabilize. In the *PowerDown* state, the power consumption is further reduced due to elimination of the leakage currents. The remaining 50 uW, or more, is due to the peripheral cores that must remain powered to enable the processor to be woken up and to maintain the real-time clock. As we shall see in Section 6, knowledge of the power drain and transition energies enables the WDT routine to select the optimal low power state.

2.2 Periodic Timing

The standard Linux kernel is implemented around the notion that it will get interrupted 100 times per second. This periodic interrupt is also known as the “tick”. In Linux, the variable *jiffies* counts the number of timer interrupts, or ticks, since kernel startup. *jiffies* in turn is used to update kernel time, process times, and to check expiration of callback timers. Periodic timer ticks are of course also well suited for multitasking environments when several tasks are actively running. This was certainly true when Unix was designed because a few computers were used by several people simultaneously. User tasks would be switched to provide a fair use of the system’s resources for all users. Moreover the timer ticks were used to manage software timers, check queues and lists, flush file buffers, swap out dirty memory pages, etc. One has to revisit this basic

approach today, since one user typically has several computers, most of which are typically idle for a good part of the day.

The disadvantages of a Periodic Timing scheme are discussed below.

2.2.1 Wasting energy in workless timer ticks

The periodic timer interrupt will periodically wake up the processor and subsequently the timer interrupt handler will be executed, even when the system is idling. However, whenever the system is idling, the queues and lists that need to be checked are empty and contain no expired callback timers and only the time variables get updated during such ticks. There is no fundamental reason why a periodic timer interrupt is needed just to maintain time.

2.2.2 State transition delays exceeding tick interval

There may be more power efficient low power states that cannot be exploited simply because the duration of the processor power state transitions exceed the tick period. For example, the most efficient low power states in the Cirrus Logic EP7312 [3] and in the Intel StrongARM 1110 [4] are the *ClockSuspend* state (Cirrus denotes it STANDBY) and the *PowerDown* state (Intel denotes it SLEEP), which may take up to 250 ms and 160 ms, respectively, to exit. With a periodic interrupt occurring every 10 ms, entering these low power states would result in timer ticks getting missed. This would make the OS timer callback service useless, and it would mess up time keeping. Besides, only 10 ms, or less, would be spent in the low power state before the next interrupt would occur which will cause the processor to start transitioning back out of the low power state again.

2.2.3 Excessive state transition energy consumption

In general, the more power efficient a low power state is, the more the energy required for state transition and the larger the transition delays in order to exploit 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 and on the transition energies.

2.2.4 Disabling of the system timer

Since the systems timer is typically disabled in the more efficient low power states, one would have to switch to another timer interrupt source before entering one of these states. Usually, it is possible to switch over to the real-time clock (RTC) timer. However, some processors do not offer fine grain resolution with the RTC timer. For example, the RTC timer in the Cirrus EP7312 has a resolution of only one second, which obviously cannot be used to generate a 100 Hz timer interrupt. Secondly, in case an RTC does offer fine grain resolution, there is more overhead associated with managing it to generate a periodic interrupt since RTCs were not designed for this purpose. Rather, RTCs are designed with large monotonically incrementing counter registers which, when compared against a match register, can generate in interrupt. Internal system timers, on the other hand, are typically smaller registers clocked at higher frequencies and which are initially populated with a load value corresponding to the timer interrupt interval. When the counter reaches zero, the initial load value is automatically reloaded on the next clock edge. Thus, an internal system timer has zero maintenance. In some cases an external timer source would be needed to generate the system timer interrupt.

3. Details for the WDT Scheme and Implementation

We now discuss the Work Dependent Timing (WDT) scheme with frequent references to the implementation within the Linux operating system for predominantly idle mobile devices. With the WDT scheme we are able to resolve all the limitations of the Periodic Timing (PT) scheme, presented in the previous section. Figure 1 shows a flow chart of the WDT scheme which is somewhat Linux specific. It also assumes a processor which has two low power states, namely the *Idle* and *ClockSuspend* states. In Figure 1, the gray boxes represent the WDT scheme. White boxes are the conventional functions. Boxes that are white/gray signify that their functions can operate in both conventional PT mode and in WDT mode.

3.1 Work Dependent Timing scheme

In an OS like Linux, whenever the current work item is suspended, the execution returns to the main, and infinite, idle loop, in which the first thing the WDT scheme aims to resolve is, "*Is there more work to be done?*". Usually, the answer is "No" which causes entry into the WDT mode of operation. In this mode the callback timer list is first parsed to extract 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 also selected and an associated timeout value calculated. The WDT routine then reprograms the selected hardware timer with the timeout value and passes control on to the Power State Transition (PST) routine.

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 Operation state (i.e., CPU running.) Typically, it involves preparing peripheral devices

and software drivers for the state change, saving/restoring system state, flushing/preparing data cache and TLB, and so forth. Description of the detailed operation of the PST is very system dependent. The impact on system software is discussed later.

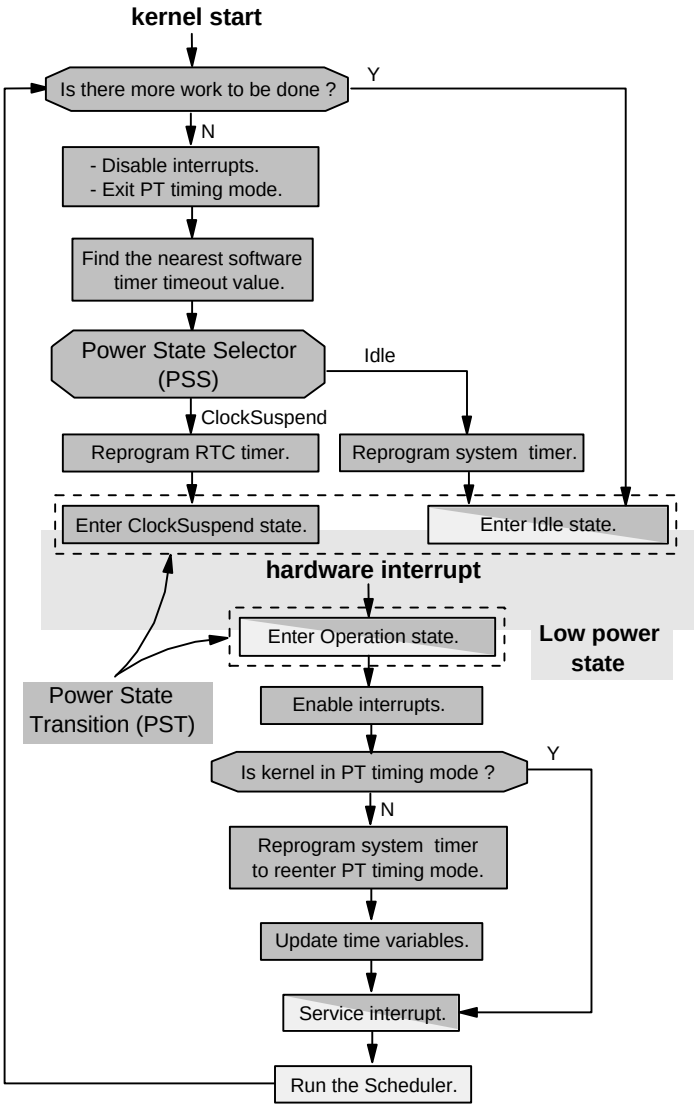

Figure 1. WDT flow chart. Gray boxes belong to the WDT routine. White boxes represent the conventional functions in the main idle loop. Boxes that are white/gray can operate in either WDT mode or conventional (PT) mode.

While in the low power state (the light-gray area in Figure 1), all execution has stopped and the processor will remain in this state until a hardware interrupt occurs. On exit from the low power state, the OS has to first determine which timing mode it is in, since it is possible that 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, the WDT routine 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/process variables is indeed required whenever there is work to be done. After reentry into PT mode, the WDT routine 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. Upon return from the interrupt handler, the scheduler is run. This completes the idle loop.

Figure 2 illustrates the effect of the PT and WDT schemes on the dynamic power consumption. As seen, the WDT scheme eliminates the execution of all the workless timer interrupts, creates extended idle periods, and the *ClockSuspend* state may be entered if the nearest timer callback timeout value is greater than 50 ms, which is the transition delay when exiting the *ClockSuspend* state (in this hypothetical example.)

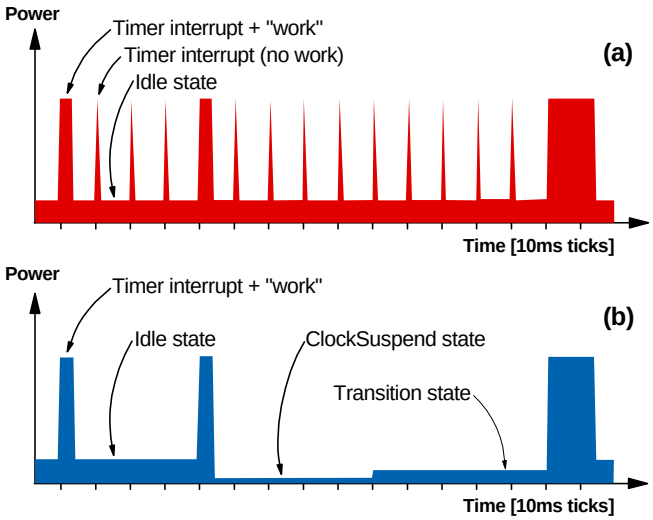

Figure 2. Illustration of the dynamic power consumption a) with the Periodic Timing scheme and b) with the WDT scheme. Note the 50 ms duration of the transition state. That's the reason why the *ClockSuspend* state cannot be exploited between the first two timers.

3.2 Keeping time and handling random interrupts

Clearly, the method for keeping time in the WDT mode of operation cannot rely on a timer tick that is no longer present. Rather, it depends entirely on the reading of a monotonically incrementing counter, such as a real-time clock (RTC) register. Processors such as [2,10] have an upper and a lower RTC register. The upper register is 32-bit wide and is clocked at 1 Hz. The lower register is 6-bit wide in [2] and 32-bit wide in [10]. In both cases, the resolution of the lower register is good enough for reading time accurately. So far we have decided not to deal with overflow of the upper RTC register since it would require 136 years before overflow occurred. Essentially, 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*.

Time is also updated on non-timer interrupts. The reason for this is that the interrupt handler may call a time function, or even attempt to set time. True time, or kernel time, is a fundamental parameter of any operating system, and it must be reliably and persistently updated regardless of changes to the timing scheme.

3.3 Selecting the optimal low power state

The Power State Selector (PSS) routine selects the optimal low power state to reduce overall energy consumption and 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 accommodates the exit delay, and thus guarantees timely execution of the callback timer. Furthermore, it must compare the exit delay to user or application specified demands to response times. For example, if the user has to press a touch-screen for more than 250 ms in order for the press to be registered, because the exit delay out of the *ClockSuspend* state is 250 ms, then that may be regarded as unreasonable. In another example, when considering to use the *PowerDown* state of a processor in favor of the *ClockSuspend* state, the PSS must know how long it takes to transition into and out of that state as well as know how much energy is consumed during the transition periods. If too much energy is spent entering and exiting the *PowerDown* state compared with the energy savings experienced once in it, it would be more optimal to remain in the less efficient *ClockSuspend* state. The PSS must also know the resolution of the various available hardware timer resources as well as their phase relationships in order to calculate when a timer will be able to generate an interrupt. For example the phase of the RTC interrupt cannot always be changed. In case of the Cirrus EP7211 [2], the RTC interrupt will always occur on a whole second boundary in absolute time. In other words, if time is right now 1234 s, and the next software timer expires in 1.5 s, then the RTC can not be manipulated to generate an interrupt at 1234s + 1.5s = 1235.5 s. In order to utilize the RTC, one would have to adjust the RTC to interrupt at 1235 s, and then transition into a less efficient low power state where the internal hardware timer can be used to generate the remaining 0.5 s sleep time. Typically, internal hardware timers have very fine grain resolution and their phases can be easily adjusted by modifying their load value. In general, all these timing details will be highly system dependent and in many cases would have to be measured and experimented with to get it right. For example, issues such as how much state the processor has, whether it is saved by hardware or by software, what type of memory the state is saved in, and bus and CPU frequencies, all affect the transition time and energy. A generic PSS algorithm is presented in Section 5.

3.4 Patch size and overhead

To implement the WDT scheme in ARM Linux 2.4.2-rmk1-bluemug7 requires about 800 new lines of code. In addition, roughly the same amount of code line have been deleted in the standard kernel version. The computational overhead, when running at 18 MHz on the TestDevice (see Section 4) is dependent on the timing mode. When the kernel is in the WDT mode, the time to service a timer interrupt has increased from 79 us in the conventional PT kernel to 170 us in the WDT kernel. When the kernel is already in PT mode, the time to service a timer interrupt has increased to 100 us in the WDT kernel. So there is some overhead, but when there is work to be done, the overhead is less significant and this is what we desire since we do not want to spend more time in handling timer interrupts.

3.5 Impact on system software

Some modifications to the system software are necessary to use the WDT scheme. When the system designer first brings up the WDT based kernel on his device, it may not work perfectly, i.e., it doesn't skip timer ticks as efficiently as expected. In this section we shall discuss the most important and illustrative obstacles that we have experienced when running the WDT scheme on the TestDevice.

Blinking cursor: The first issue is related to the user interface which often has blinking cursors 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 update is to register a timer function for callback every 500 ms. In the case where it takes 250 ms to exit the *ClockSuspend* state in the TestDevice, and where the RTC is only able to interrupt on whole second boundaries, this blinking effect completely disables the exploiting of the *ClockSuspend* state, and thus voids the chance of any significant battery life gains. The solution in these cases is to provide the tradeoff to the user and let him make the choice.

Keyboard tasklet: Some system interactions are harder to predict like the following example. We found that when we bring up X11 that it always opens up a virtual terminal in which it keeps looking for a keyboard to be attached, presumably because X11 designers assumed keyboards will be attached to all computers that use X11. A "tasklet" is put on a 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 TestDevice, we have no keyboard attached, and so the tasklet remains permanently on the queue. This causes the answer to the question, "Is there more work to be done?" in Figure 1 to be, "Yes" and therefore the WDT mode is never entered. This effect has a bigger impact on power

consumption than the blinking cursor. Not only is it not possible to exploit the *ClockSuspend* state, but not a single timer tick can be skipped. We resolved the issue by only queuing the tasklet if a keyboard is attached.

Device driver-kernel interactions: Some device drivers interact with peripheral cores such as the UART, the LCD controller, the synchronous serial interface (SSI), etc. These cores are disabled in the *ClockSuspend* state which means that the devices they are connected to may be rendered useless. For example, in the TestDevice a touch-screen device is connected to the SSI via an external A/D converter (ADC). When a user presses the touch-screen, the driver/SSI sends a request-for-data to the ADC. Next, the ADC takes a small but finite time to handle the request. This delay, however, is long enough for the Linux kernel to enter the *ClockSuspend* state. With SSI there is no way for the ADC to wake-up the processor. Thus, the touch coordinates never get sent to the processor, and so the physical touch action results in no GUI action. To resolve the issue we have introduced a generic mechanism through which device drivers can prevent the WDT scheme from exploiting the *ClockSuspend* state until the drivers decide that their associated device is no longer needed.

Persistent kernel daemons: There is a of number of daemons in the Linux kernel which are scheduled for execution with intervals of one second or more, but which can be 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. Again, this will cripple the operation of the WDT scheme, so we extended the interval to 30 s. We did the same for the *bdflush()* daemon which writes out dirty and aged file buffers to disk. When the system is active, the normal values have to be restored to preserve application level compatibility. So there is a concept of changing these intervals depending on whether the system is active or inactive.

RTC/software timer phase: In early experiments with the WDT based kernel on the TestDevice (see next section), the measured average power would vary significantly every time the kernel was rebooted. By examining the waveform of 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 running the software timer callback function. The root of the problem was the RTC phase which cannot be adjusted. So instead we adjust the phase of long-term (i.e., >1 s) timers 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. As it turns out, in the TestDevice, the vast majority of timers fall in the long-term category. This optimizes the use of the RTC timer interrupt, increases the time spent in *ClockSuspend* state and maximizes battery life.

As may be understood from the above, the WDT scheme in general is sensitive to the way system software is implemented and that a fair degree of tuning is required to get it to work efficiently.

4. Experimental Results

In this section we shall present measurement of average power consumption on an embedded platform with the WDT scheme and with the PT scheme and with varying computational loads. Our hardware test platform, called the TestDevice, 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 which is always on (it consumes 1.8 mW.) In order to enable the WDT scheme to perform optimally, we implemented all the kernel fixes discussed in Section 3.5. We made the same modifications to the conventional PT kernel for the fairest possible comparison even though they have a near zero impact on the power consumption of the PT kernel.

4.1 Experimental Setup

Figure 4 shows the experimental setup used for measuring average system power consumption on the TestDevice platform.

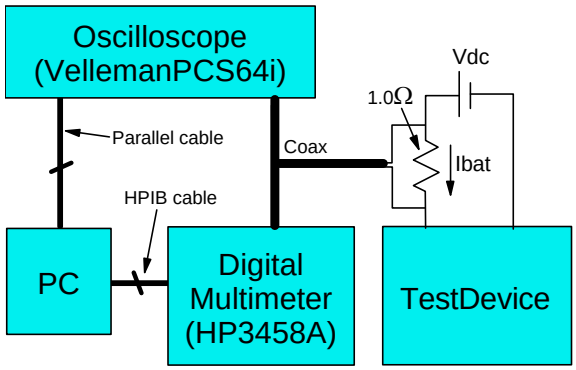

Figure 3 Experimental setup for measuring average power consumption of the TestDevice platform. The PC is used for collecting data from the multimeter and for monitoring power waveforms captured by the oscilloscope.

The current consumption, I_{bat}, is found by measuring the voltage across a 1 Ohm resistor inserted in series with the DC supply (Vdc=5V.) The digital multimeter (DMM) measures the voltage drop with a resolution of 10 nV, or equivalently 10 nA, which is much smaller than the minimum current draw of the TestDevice of around 500 uA. Further, the DMM samples I_{bat} every 1 ms. Even though 1 ms sampling time is not able to capture the instantaneous power consumption of every computing event, by virtue of sampling over an extended interval, the power fluctuations associated with various computing events will average out.

This is true if these events are repetitive events and if the events are not phase aligned with the sampling timer in the DMM. The latter is ensured by detuning the timer to 0.99 ms. There is no correlation between the sampling times in the DMM and computational events on the TestDevice. The computer (PC) is used to control and to collect data from the DMM, as described in [12]. The PC 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 TestDevice is operating as expected. This turned out to be an invaluable debugging tool.

On the TestDevice we run a program, *simm_load()*, to simulate a real compute 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 us, and Loop2 executes CPU bound instructions for 150 us. 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 simulated different types of repetitive work loads.

P_{active}: *active* power consumption	155mW
$P_{pm,PT}$: *pm* power consumption in PT case (*Idle* state)	23.3mW
$P_{pm,WDT}$: *pm* power consumption in WDT case (*ClockSuspend* state)	4.57mW
$t_{trans,PT}$: *trans* time in PT case (*Idle* state)	0
$t_{trans,WDT}$: *trans* time in WDT case (*ClockSuspend* state)	220 ms
$P_{trans,WDT}$: *trans* power consumption in WDT case (*ClockSuspend* state)	22.8mW
$N_{pops,0}$: Frequency of background timer pops	0.125 Hz

Table 2. Parameters measured on the TestDevice.

As part of examining the dynamic power consumption, the following parameters were measured as shown in Table 2. These parameters are typical parameters that a system designer should measure in order to determine if the WDT scheme can offer significant battery life improvements if adopted. We shall also use these parameters in Section 5.4 to evaluate the accuracy of the model of the battery life gain presented in Section 5.

4.2 Measurement Results

The experimental procedure is as follows. The Linux kernel and X11 are loaded onto the TestDevice. Then the load is set to one of the following values, {0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.25, 0.5}. The load routine is programmed to run either once every minute or once every three seconds. Say the load is 0.03, then the load routine will run for either 1.8 s once every 1 minute or 90 ms every 3 s. The PC in Figure 3 then collects data from the multimeter over a period of exactly 4 min. During this time, 242400 data points are collected of which the average is calculated. This gives us the average power consumed by the TestDevice. This measurement is repeated 4 times to optimize accuracy which is better than 1%. We go through the same steps with both the WDT and the PT based kernel.

Figure 4 shows the gain in battery life achievable with the WDT based kernel as a function of the computational load on the system and with the workload timer pop frequency as parameter, i.e., either 1, 20 or 60 pops per minute (i.e., 1/60, 1/3 or 1 Hz). In all cases the WDT scheme is skipping timer ticks between work items, and in between it returns to the *ClockSuspend* state. As the figure shows, the load timer pop frequency has a significant impact on the battery life gain, even for very light loads. The reason for this is that for a given load, the more this load is fragmented, or spread out, in time, proportionally the more the transition energy of exiting the *ClockSuspend* state is taking its toll on the total energy consumption. Even for an infinitely small load, the processor has to wake up and transition out of *ClockSuspend* state just to execute a couple of instructions which results in wasted transition energy.

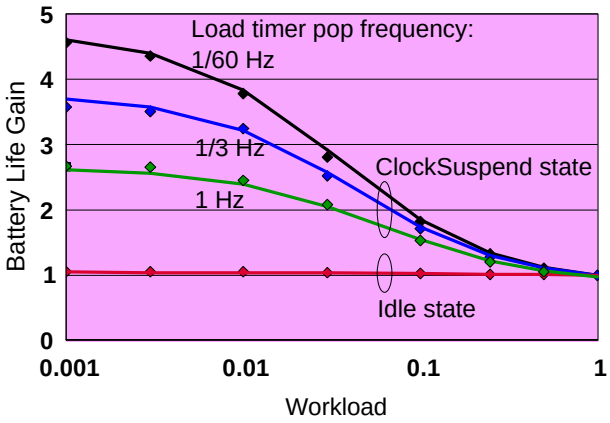

Figure 4. Measured battery life gain (markers) as function of workload achieved with the WDT scheme. Solid lines represent modeled results from Section 5.4.

As expected, the smaller the load, i.e., the more the system idles, the larger is the gain. As the load increases, the power contribution from the load starts to dominate, essentially washing out the much smaller contribution from the low power state. Still, gains >2 are achieved for loads of 4-8% and smaller. Also shown on the figure is the result of just skipping timer ticks and only using the *Idle* state (red markers), i.e., simulating the case where, say, a user informs the system that he cannot accept the 220 ms reaction time. This result is independent of the load frequency since there is no transition energy penalty when exploiting the *Idle*

state. As seen from the figure only minor gains are achievable by skipping ticks. For example, for workloads smaller than 0.1%, the gain is 4.4%.

5. Estimation of Battery Life Gain

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

5.1 Modeling Battery Life Gain

Figure 5 is an illustration of some of the variables that will be used in the following and their association to the dynamic power consumption of a computing system.

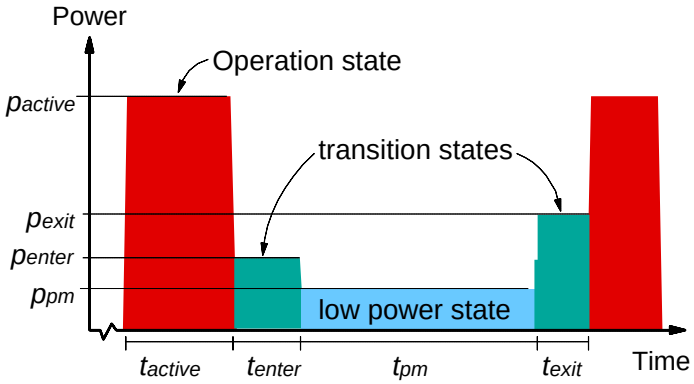

Figure 5. Illustration of the power and time variables associated with the dynamic power consumption of a low power embedded computing system.

The lifetime, T_{bat}, of an ideal battery with capacity, C_{bat}, and supplying an average power consumption of P_{avg} may be expressed as $T_{bat} = C_{bat}/P_{avg}$. In practice batteries are non-ideal [13] (i.e., the capacity, C_{bat}, is function of P_{avg}.) The impact of this non-linearity is mostly on the absolute value of T_{bat} and is most predominant for large values ofP_{avg}. In this paper, however, we are mostly concerned with idling devices which consume very small powers most of the time. P_{avg} is the average power consumption of the computer system over the lifetime of the battery and is expressed as

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

where

$$\tau_k = t_k/(t_{active} + t_{trans} + t_{pm}),\ k \in \{active, pm, trans\}$$

is the relative time spent in the various system states. *active* is the system state in which the processor is executing instructions (also sometimes called Operation state) and where memory may be accessed. *pm* is the system state in which the processor and memory are in a low power state. *trans* is the system state in which the processor is transitioning into and out of the *pm* state and where memory may be accessed. The system states, *pm* and *trans*, furthermore are broken into sub-states in the sense that there may be multiple low power *pm* states, and correspondingly multiple *trans* states. P_k and t_k are the average system power of and the average time spent in the various system states. In general, t_k will be composed of many smaller contributions as the system dynamically transitions between the system states. In addition P_{active} may vary during execution. We shall assume that any time a given system state is used, the power consumption is always the same.

The *trans* time and power are composed of two contributions, namely,

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

and

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

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.

Since we want to quantify the gain in battery life that may be achieved by using a WDT based timing scheme rather than a conventional PT scheme, the key parameter to calculate is the battery life gain, γ, which may be expressed as

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

The battery life gain is a key parameter that the mobile system designer has to determine to know if the WDT scheme will provide worthwhile battery life improvements.

5.2 Almost Zero Device Activity

In this subsection we will show the improvement in battery life that may be achieved with the WDT scheme over the PT scheme due to elimination of the active power consumption for servicing of timer interrupts. In the WDT case, it will be assumed that there is almost never any need for the device to wake up. In other words, the hardware timer is programmed to interrupt the processor at an infinite time, and therefore $\tau_{pm} = 1.000$. Even though a system does wake up every now and then due to background timer pops (OS daemons), the active contribution to the overall power consumption in the TestDevice can be safely ignored. In the

PT case, we shall assume that $\tau_{trans} = 0$ since only the processor *Idle* state is exploited; and as may be seen from Table 1 the transition time is so small that it may be ignored. At this point we shall make the analysis somewhat specific to the TestDevice. Basically, we shall use the measured value of t_{active} for an averaged timer interrupt. Next, realizing that the kernel behaves in a highly repetitive fashion in the PT case with a periodicity of 10 ms, we get τ_{active} = 0.079ms/10ms = 0.0079. Thus, $\tau_{pm} = 0.9921$ is spent in the *pm* state. Using these values in Equation 1 and 4, the battery life gain, γ, of the WDT scheme over the PT scheme may be calculated as

$$\gamma = (P_{active} \bullet 0.0079 + 0.9921 P_{pm,PT})/P_{pm,WDT}$$
$$= (P_{active}/P_{pm,PT} \bullet 0.0079 + 0.9921) \bullet (P_{pm,PT}/P_{pm,WDT}) \quad \text{(Eq. 5)}$$

First we shall look at the benefits of the WDT scheme by examining what happens in a system where the same *pm* state is used in both the WDT and PT case, i.e., when $P_{pm,WDT} = P_{pm,PT} = P_{pm}$. This reduces Equation 5 to

$$\gamma = P_{active}/P_{pm} \bullet 0.0079 + 0.9921 \quad \text{(Eq. 6)}$$

This will extract the active power savings due to elimination of all the energy consumed by handling the timer interrupt. Figure 6 shows γ as a function of the relative difference between *active* and *pm* power, P_{active}/P_{pm}.

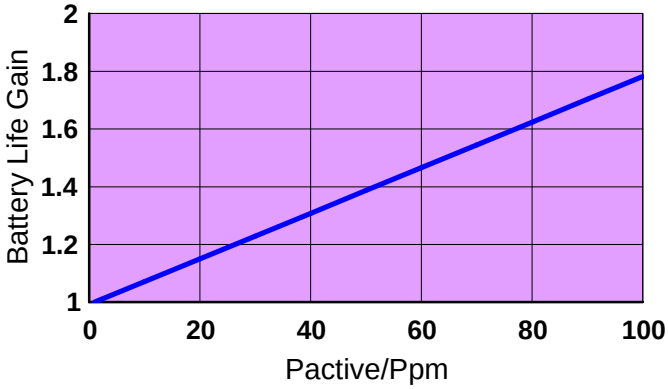

Figure 6. Battery life gain of the WDT scheme as function of the relative difference in power consumption between the *active* and *pm* states, P_{active}/P_{pm}.

As may be seen from Figure 6, and from Equation 6, the larger the ratio is between the *pm* and the *active* power, the larger is the WDT battery life gain. From Figure 6 one may also quickly determine the relative increase in battery life, and reduction in power consumption, achievable with the WDT scheme if the *active* and *pm* powers of a given target system are known. For example, we measured P_{active}/P_{pm} = 6.7 on the TestDevice which should yield a 4.5% gain in battery life. The measured gain was 4.4%.

Equation 6 must be used with caution. For example, by increasing the frequency of our system from 18 MHz to say 74 MHz P_{active} would increase almost four-fold. However, ,τ_{active} would not necessarily decrease four-fold correspondingly, due to cache misses and due to the slow memory used in the TestDevice. However, according to Equation 6 this is a good thing since the larger τ_{active} is, the higher the gain. Nevertheless, it would be a mistake to tune the processor frequency to 74 MHz just for the sole purpose of extending battery life. In fact, the absolute battery life would decrease in both the PT and the WDT case, except it would decrease more in the PT case. On the other hand, systems that do have high P_{active}/P_{pm} in their most power efficient mode, and which are slow to execute the timer interrupt, will benefit more from adopting the WDT scheme.

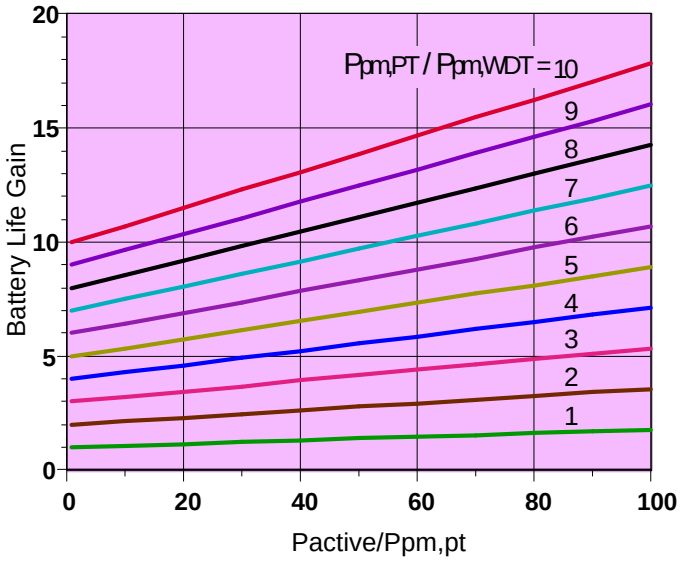

Figure 7. Battery life gain of the WDT scheme as function of $P_{active}/P_{pm,PT}$ with $P_{pm,PT}/P_{pm,WDT}$ as parameter.

Next, we shall consider what happens when the WDT scheme enables the OS to exploit a low power system state, which could not otherwise be exploited in the PT case, just like it was the case with the *ClockSuspend* state with the TestDevice in section 4. In our first scenario, we shall ignore the transition energy, and use Equation 5 in which the last factor, $P_{pm,PT}/P_{pm,WDT}$, represents the improvement due to the exposure of the untapped low power state. In fact, the gain in Equation 5 is really the product of two distinct gains; the gain due to elimination of the active switching energy in workless timer interrupts and the gain due to exposure of a more efficient low power state. Figure 7 shows the result of applying Equation 5 for the range of $1 \le P_{pm,PT}/P_{pm,WDT} \le 10$. As seen, the battery life improvement scales directly with the ratio between the *pm* powers in the two timing schemes, and that the gains may be quite significant, as was also seen in section 4.

5.3 Non-Zero Device Activity

In the previous section we assumed the scenario where the mobile device is idling nearly 100% of the time. In reality, mobile devices in the "instant on" mode will for various reasons wake up on occasion to perform work, such as on user interrupt, network interrupt, clock applications, slow changing screen savers, background timer pops (e.g., memory swap out daemon), etc. 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 the total relative time spent in the *active* state becomes $\tau_{active,PT} = \alpha + 0.0079$ and $\tau_{active,WDT} = \alpha'$ in the PT and WDT cases, respectively, where $\alpha' = \alpha \bullet (1 + \tau_{active,irq}) = \alpha \bullet (1 + 0.0079)$. The purpose of α' is to account for the energy in periodic timer interrupts that occur while there is work to be done in the WDT case. By incorporating the assumptions above and again using $\tau_{trans} = 0$, the battery life gain, γ, may be calculated from Equation 4 and 1 as

$$\gamma = (P_{active}/P_{pm,PT} \bullet (\alpha + 0.0079) + 0.9921 - \alpha) / (P_{active}/P_{pm,PT} \bullet \alpha' + P_{pm,WDT}/P_{pm,PT} \bullet (1 - \alpha')) \quad \text{(Eq. 7)}$$

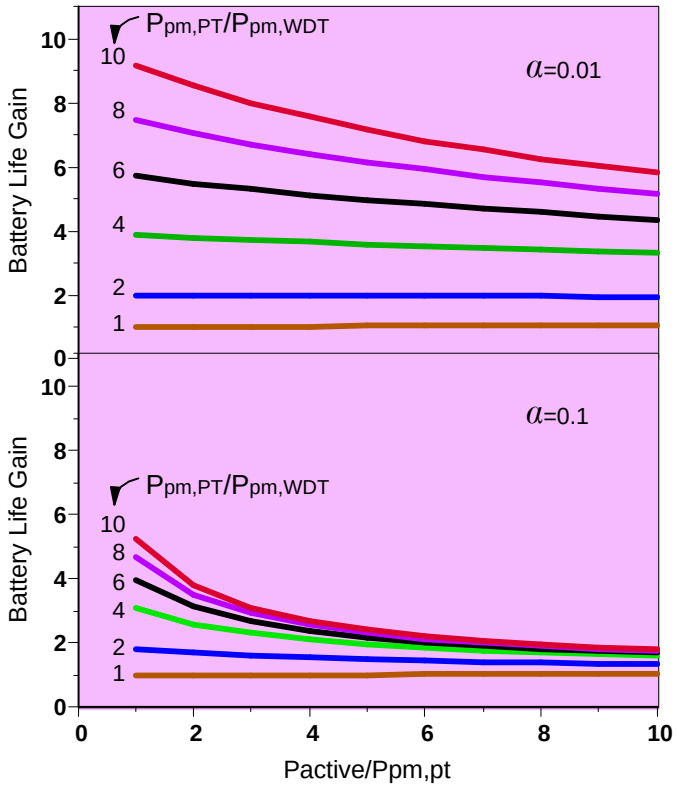

Figure 8. Battery life gain of the WDT scheme as function of $P_{active}/P_{pm,PT}$ for device workloads, α = {0.01, 0.1}, and with $P_{pm,PT}/P_{pm,WDT}$ = {1, 2, 4, 6, 8, 10} as parameter.

Figure 8 shows the relative increase in battery life for some select values of the device workload, α={0.01, 0.1} and with $P_{pm,PT}/P_{pm,WDT}$ as parameter. For workloads smaller than α = 0.001, the workload has insignificant impact and the gains approach the case shown in Figure 7. As in Figure 7 $P_{pm,PT}/P_{pm,WDT}$ has much more impact on the battery life gain than P_{active}/P_{pm}. Note, however, how the workload may significantly reduce the benefit of P_{active}/P_{pm}. (Note that the x-axis on Figure 7 goes to 100 while in Figure 8 it only goes to 10.) As may be seen, a workload of α = 0.01, which equals 14 min of device activity per day, the impact is clearly evident for large values of $P_{pm,PT}/P_{pm,WDT}$ and $P_{active}/P_{pm,PT}$. Though small, the battery life gain is still large enough to merit the incorporation of the WDT scheme. In fact, even when the workload increases to α = 0.1, which equals 2.4 hours of device activity per day, there is still a clear advantage of adopting the WDT scheme over the PT scheme, assuming that the WDT scheme does indeed enable the exploitation of a significantly more efficient power state.

5.4 Non-Zero Transition Energy

In this section we shall incorporate the transition energy into the analysis. The main complication is that the transition energy is dependent on the nature of the workload, α. For example, a device can have a task running which represents a 1% workload, i.e., α = 0.01. From an energy perspective, transition energy is consumed every time the workload causes the processor to transition out of a low power state. Say the workload equals 0.6 s of execution time every 60 s. Then if the workload is executed only once every 60 s, then only one unit of transition energy is consumed. If however the 1% workload is spread out over 10 executions every 60 s (i.e., 0.06s on every execution) then 10 units of transition energy is consumed. Thus, in essence, the transition time may be regarded as a non-computational and wasteful workload. To include transition energy into the model, we introduce the parameter, β, representing this transitional non-computational workload and which, for $\alpha > 0$, may be expressed as follows

$$\beta = t_{trans} \bullet (N_{pops,0} \bullet (1 - \alpha' - \beta) + N_{pops,load}) \quad \text{(Eq. 8)}$$

In Equation 8, $N_{pops,0}$ accounts for the average number of timer pops per second, i.e., the timer pop frequency, in the system contributed by background timers such as file buffer and virtual memory daemons and other smaller applications such as a clock application, which will always exist, and pop, regardless of the presence of an additional workload. When an additional workload is introduced, the number of timer pops per unit time will increase correspondingly by $N_{pops,load}$. As either of the workloads, α' or β, increase, the likelihood of a background timer popping during the

workloads increases. However, a timer that pops during the workload does not give rise to wasted transition energy since the processor is not in a *pm* state when the timer pop occurs. Therefore, the effective number of background timer pops, $N_{pops,0}$, that do give rise to wasted transition energy must be reduced as α' and β increase. To account for this effect, we approximate the reduction in background timer pops by multiplying the correction factor $(1-\alpha'-\beta)$ onto $N_{pops,0}$ in Equation 8. The assumption behind this correction is that background timers are independent of the workload and occur randomly in time. To give an example, say that $N_{pops,0}$= 0.125 Hz (i.e 7.5 timer pops per min.). Now, say the workload is 50%, this reduces the impact of the background timers on the transition energy to 0.0625 Hz.

In the general case, β must be introduced in both the PT and the WDT power consumption expressions. By isolating β' on the left side of Equation 8, Equation 7 may now be modified to

$$\gamma = (P_{active} \bullet (\alpha' + 0.0079) + P_{trans,PT} \bullet \beta + P_{pm,PT} \bullet (0.9921 - \alpha' - \beta)) / (P_{active} \bullet \alpha' + P_{trans,WDT} \bullet \beta + P_{pm,WDT} \bullet (1 - \alpha' - \beta)) \quad \text{(Eq. 9)}$$

Note that β is dependent on the particular *pm* state being exploited. For example, for the *Idle* state, β can be ignored. But for the *ClockSuspend* state, β could be significant, as was seen in Section 4. Thus in Equation 9, it is implied that β really is $\beta_{pm,PT}$ in the numerator and $\beta_{pm,WDT}$ in the denominator. Also note that we have ignored the time spent in the background timers, i.e., that they are much faster than the timer interrupt handler.

With Equation 9, it is now possible to determine the potential battery life gains of the TestDevice in Section 4. Beforehand, we measured a number of parameters on the devices, which are listed in Table 2 in Section 4. Then we inserted these parameters in Equation 9 and let the workload range from $\alpha \in \{0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.25, 0.5, 1\}$. We considered two values of the workload timer pop frequency, $N_{pops,load} \in \{0.0167, 0.3333\}$, which corresponds to 1 and 20 timer pops per minute, respectively.

The results are shown in Figure 4 (solid lines). As may be seen there is near perfect agreement between measurements and modeled results. In view of the fact that all the parameters in Equation 8 and 9 are based on measuring the parameters in the same hardware from which we obtained the experimental results, this is good news for the device designer. It indicates that the designer may be able to quite accurately estimate possible battery life gains with the WDT scheme using our modeling methodology.

Note that in principle, with the WDT scheme, it is possible to exploit two or more *pm* states, and thus two or more associated *trans* states as well. From a modeling perspective, this is difficult to account for as it is very application specific. Basically it would be a challenge to the device designer to estimate the average *pm* and *trans* power/times for his/her particular device for proper evaluation of the WDT scheme.

6. Selecting the Optimal Power State

In this section we shall present the Power State Selector (PSS) algorithms and then apply it to a hypothetical, yet realistic, scenario.

6.1 Power State Selection Algorithm

Let us denote the total transition time entering and exiting a power state *i* as $t_{trans,i}$, the total average transition power consumed during as $p_{trans,i}$, the time spent in the power down state as $t_{pm,i}$, and the average power consumed while in the power down state as $p_{pm,i}$. The PSS routine may then select the proper state *i* by first determining if it is even legal to enter a certain state *i* according to

$$\min\{t_{timeout}, t_{response}\} > t_{trans,i} \quad \text{(Eq. 10)}$$

where $t_{timeout}$ is the timeout value of the nearest callback timer and $t_{response}$ is the user/application specified maximum system response time. Equation 10 states that only if the timeout value and the response time both exceed the total transition time of power state *i*, then power state *i* is a legal state. Having now identified the legal power states, the PSS routine will next determine which of the legal power states has the lowest overall energy consumption, $e_{total,i}$, by calculating the potential overall energy consumption of each of the legal power states as

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

where the state, *i*, that satisfies this equation is the optimal state and where

$$E_{total,i} = E_{trans,i} + E_{pm,i} = P_{trans,i} \bullet t_{trans,i} + P_{pm,i} \bullet t_{pm,i} \quad \text{(Eq.12)}$$

$$t_{timeout} = t_{trans,i} + t_{pm,i} \quad \text{(Eq. 13)}$$

Note that $t_{timeout}$ is the maximum time the processor/system may be put to sleep. But as may be seen from Equation 13, in reality the sleep time is reduced by the state transition time.

6.2 PowerDown versus ClockSuspend State

In this section we shall determine how long a sleep interval, $t_{timeout}$, is required to make the *PowerDown* (PD) state the optimal state and assuming that the only other

"competing" state is the *ClockSuspend* state. In other words, we shall determine when $E_{total,PD} < E_{total,CS}$ in Equation 11 is true.

Table 1 lists the power ranges for the two states. Considering these ranges and applying them to Equation 11 and 12, the following approximations can be made. Firstly, the size of $t_{trans,CS}$ in recent processors is very small (100-200 us) thus making $E_{trans,CS}$ so small (<1 uJ) that it can be ignored. Secondly, since $t_{trans,CS}$ is so small, it may be ignored in Equation 13 thus making $t_{pm,CS} = t_{timeout}$. These assumptions leads to the following expression which determines when it pays off to use the PD state

$$t_{timeout} > (E_{trans,PD} - P_{pm,PD} \bullet t_{trans,PD})/(P_{pm,CS} - P_{pm,PD})$$

Recognizing now that the only difference between the power consumption in the CS and PD states is the processor's *pm* power consumption, all other system contributions to the denominator cancel out. And since from Table 1 it may be seen that $P_{pm,cpu,PD} << P_{pm,cpu,CS}$, the expression reduces to

$$t_{timeout} > (E_{trans,PD} - P_{pm,PD} \bullet t_{trans,PD})/P_{pm,cpu,CS} \quad \text{(Eq. 14)}$$

Figure 9 shows the value of $t_{timeout}$ which satisfies Equation 14 as a function of the processor's power consumption in the CS state and for some select values of the PD transition energy. The range we chose for the PD transition energy is intended to be representative of the total energy required to save and restore processor context to/from SDRAM. In the figure we have ignore the contribution from $P_{pm,PD} \bullet t_{trans,PD}$ which should be a good approximation for $P_{pm,PD}$ power levels smaller than 0.2 mW and for relatively fast transition times, $t_{trans,PD}$, smaller than 5 ms.

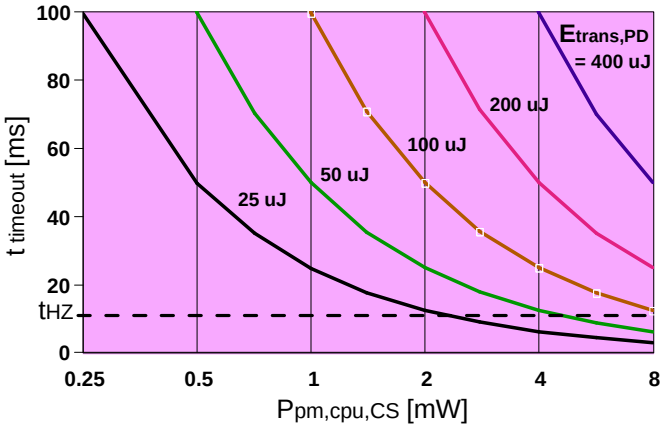

Figure 9. Sleep time, $t_{timeout}$, versus processor's CS power consumption. Curves indicate the sleep time at which the PD state becomes the optimal state.

The figure shows that the required sleep time, to make the PD state become the optimal state, decreases as $E_{trans,PD}$ decreases and as $P_{pm,cpu,CS}$ increases. Keeping in mind that the timer interrupt interval in conventional Linux OS is t_{HZ} = 10 ms, the figure also shows that for the predominant range of values considered in this example, it would not be possible to use the PD state with the conventional PT timing scheme. In other words, the WDT scheme enables the use of the PD state for increased battery life gains. We have not verified this yet. However, based on data published on IBM's most recent PowerPC405LP processor [10], power consumption in the ClockSuspend and PowerDown states are 300 uW and 54 uW, respectively. For a processor like the PowerPC405LP, which is based upon a PowerPC405 core, we estimate that the total transition energy (including SDRAM memory energy), required to save/restore the processor context and write out data-cache, may be as small as $E_{trans,PD}$ = 25 uJ in the most optimistic case. Consequently, from Figure 9, it may be concluded that it would be too "costly" to exploit the PD state with the conventional PT scheme as it requires sleep times >100 ms to make it worthwhile. On the other hand, with the WDT scheme it is very likely that such long sleep periods can be produced as from Table 2 it may be seen that the average sleep period is around 8 s.

We anticipate that the WDT scheme will enable the dynamic exploitation of the PowerDown, or PowerDown like, states of other processors as well, such as Intel's PXA 250/210 [6] and SA1110 [4] processors.

6.3 Elimination of Transition Energy

In view of the above, assume that indeed it is possible to exploit the *PowerDown* (PD) state between the 10 ms PT timer ticks. By then calculating the total system energy from Equation 12 for the PT and the WDT cases assuming $E_{trans,PD}$ = 25 uJ, $P_{pm,PD}$ = 50 uW, an average sleep time of 8 s (in the WDT case according to Table 2), and an average sleep time of 10 ms (in the PT case, where we ignore the transition time which we expect to be smaller than 1 ms), the following total average system power levels may be calculated

$$P_{avg,PT} = (25\text{uJ}+(50\text{uW}+P_{other}) \bullet 10\text{ms})/10\text{ms}$$
$$= 2.5\text{mW}+P_{other}$$

$$P_{avg,WDT} = (25\text{uJ}+(50\text{uW}+P_{other}) \bullet 8\text{s})/8\text{s}$$
$$= 50\text{uW}+P_{other}$$

where P_{other} is the average background power consumption due to SDRAM self-refresh, power supply loss, etc., and which is probably not smaller than 1 mW (in a system with say 16 MB SDRAM and the LCD and other peripherals turned off.) The above equations clearly demonstrate the impact of the transition energy on the total power

consumption in the PT case as well as the WDT schemes ability to eliminate these energies. For a very low power system (e.g. P_{other} = 1 mW), the average power in the WDT case is 3.5 times smaller than in the PT case.

7. Related Work

ACPI [14] is probably the most widely known power management approach in which the processor and connected devices, such as display, hard-drive and network interface, can register their PM capabilities with the OS to enable the OS to disable the devices when certain criteria are met, typically when a relatively long time of device inactivity has elapsed. Vahdat et al. [15] make a case for revisiting basic decisions in operating systems design by keeping power consumption in mind. And most recently, along the same line of thought, Zeng et al. [16] actually demonstrated a Linux system which uses energy consumption as a resource parameter to schedule processor time for applications. Weiser et al [17] and Grunwald et al. [18] consider several methods for varying the clock speed dynamically under control of the operating system. Lu et al. [19] investigate arranging the execution orders of tasks so that idle periods are clustered together instead of being scattered. Energy aware task scheduling, CPU instruction scheduling, wireless communication protocols, sensor networks, etc., are being studied by several researchers.

Researchers in mobile systems have also investigated how applications can be adapted to reduce energy consumption [12]. Some applications can adapt themselves to consume fewer energy resources when needed, such as lowering the video frame rate, using a smaller dictionary for voice recognition, multi-resolution rendering, etc. The idea of using devices in the environment that can help with computation to reduce the power consumed by the mobile device has also been investigated [20,21]. Lorch and Smith [22] describe several software strategies for energy management for portable devices.

The task of turning hardware resources on or off is best left to the operating system since it keeps track of all the activities in the system. So one question that arises is what other operating systems do. We examined the operation of Microsoft WinCE [23] and RTLinux [24]. WinCE is a non-real-time OS and uses a periodic timer tick. But it also allows the kernel to skip timer ticks according to the next scheduled computing task and if there are currently no active tasks/threads executing. This is no different than in our WDT scheme. It is impossible, however, to comment on other similarities and differences between WinCE and the WDT scheme since the WinCE source code is not available and the information in [23] is sparse. Based on [23], there is no indication that WinCE takes the transition time and energy into account when using the *Sleep* state. Nor is there any indication that WinCE has the capability to exploit more than one low power state, or that WinCE has the capability to exploit the most efficient low power state among two or more low power states.

One OS that does not rely on a periodic timer tick is RT Linux [20]. However, it is only the real-time part of it that works that way. The Linux kernel itself is treated as a preemptible non-real-time thread of lowest priority, and is interrupted periodically every 10 ms. The similarity between the real-time part of RT Linux, and other real-time kernels for that matter, and our WDT scheme is that every timer event is separately scheduled by programming a one shot timer, except we rely on the PT scheme when the system is not idling.

There are other Linux versions that do skip timer ticks, for example Schwidefsky [25]. However, his aim is not to save power but rather to reduce computational overhead in multi-image systems in which hundreds of Linux images may run simultaneously on just a few processors (e.g. in mainframe systems). Typically, many of these images will be idling and/or even if not idling the majority of the timer interrupts do nothing more than update kernel and process times. The accumulated computational load from servicing the excessively many timer interrupts starts to bog down the system performance. By only allowing timer interrupts associated with work items, the accumulated load can be significantly reduced. Timer ticks are skipped even while there is work to be done. *jiffies* and kernel reference time are updated either when referenced or on every entry into kernel space. A second hardware timer is needed to timeout the current process that is running, which is how multitasking is facilitated. It is possible that *jiffies* and kernel time do not get properly updated if too much time is spent in kernel space. There is also a complication in the sense that the current nearest timer may be modified, or an even shorter timer may be added, while the current process is running. The WDT scheme has none of these problems which arise simply because of the need to skip ticks during work. In embedded devices there is no incentive in skipping timer ticks during work, since these ticks present a negligible power drain and computational load.

OSs such as PalmOS, VxWorks, and REAL/IX use periodic timer interrupts, but some allow for variation in the frequency of the timer interrupt. Sometimes this interval is programmed to small numbers so that the operating system can respond at finer granularities such as 1 us as opposed to 10 ms. Presumably one could increase the periodic timer interrupt interval from 10 ms to larger values so that the device could be in a lower power state in between, however among other things the responsiveness of the system would suffer. In fact we attempted this as a first solution but quickly abandoned it because 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 hard coded into its code base.

8. Conclusion

We presented the details of implementation of the Work Dependent Timing scheme under Linux and corresponding experimental and modeling results. The WDT scheme was designed to skip timer ticks whenever the OS is idling and exploit the most optimal low power state with low complexity and computational overhead. We know of no other power management approach that offers the power efficiency and aggressiveness of the WDT scheme. The one particular property of the WDT scheme that makes it so efficient is the tight and seamless integration into the core kernel.

One of the main benefits of the scheme is that the skipping of many successive timer ticks increases the time the processor may go to sleep. In turn, this effect may expose a more efficient low power state. For a predominantly idling but "instant on" mobile device, this may potentially result in significant battery life gains. Our experiments with the TestDevice showed a battery life gain of 4.6 due to the exposure and exploitation of the *ClockSuspend* state. Secondly, in processors that have a *PowerDown* state, the scheme can essentially eliminate the fairly large energy associated with transitioning into and out of the state periodically with the PT scheme. Thirdly, in cases where there is a large differential between the system *pm* power and the system *active* power, i.e., greater than a factor of thirty, the WDT scheme may increase the battery life by more than 25%. 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 driven power management, etc.

We also introduced a simple analytical model that may be used by mobile device designers to determine whether the WDT scheme would provide a worthwhile increase in the battery life for the device they are building. The model showed good agreement with the experimental results. Finally, we presented a Power State Selection algorithm and combined it with the WDT scheme. With this algorithm we predict that the *PowerDown* state of advanced mobile processors can be much more efficiently exploited with the WDT scheme, and that in fact in some cases this state can only be exploited dynamically with the WDT scheme, and not at all with the PT based kernel.

We hope that more devices will use this technique to improve the battery life.

References

[1] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, C. Narayanaswamy, "*Energy trade-offs in the IBM Wristwatch computer*," Intl Symp. Wearable Computing , 2001.

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

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

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

[5] Intel, "*The Intel PXA250 Applications Procesor: White Paper*," Feb 2002.

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

[7] Motorola, "MC9328MXL/D," Advance information, Dec. 2002.

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

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

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

[11] Max Baron, *"Cool Performance for Handhelds*," Microprocessor Report, Aug 2002.

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

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

[14] Compaq, Intel, Microsoft, Phoenix, Toshiba Corporations,"Advanced Configuration and Power Interface Specification," Rev.2.0a, March 2002.

[15] A. Vahdat, A. Lebeck, C. S. Ellis, "*Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency*", 9th ACM SIGOPS European workshop, September 2000.

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

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

[18] 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.

[19] 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.

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

[21] R. Balan, J. Flinn, M. Satyanarayanan, S.

Sinnamohideen, "*The Case for Cyber Foraging*," Proc Tenth ACM SIGOPS European Workshop, Sep 2002.
[22] 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.
[23] http://msdn.microsoft.com/library. Search for "WinCE power management".
[24] V. Yodaiken, M. Barabanov, "*A Real-Time Linux*," Proc. of USENIX Annual Tech. Conf., 1997.
[25] Martin Schwidefsky, "*No 100 HZ timer!*", http://lwn.net/2001/0412/a/ibm-timer.php3.