System Shutdown Techniques for Energy Efficient Real Time Computation

by

Daniel Aaron Hartley

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

© Daniel Aaron Hartley. All rights reserved.
The author hereby grants MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author__________________________
Department of Electrical Engineering and Computer Science
May 19, 2000

Certified by_____________________
Roger Racine
Charles Stark Draper Laboratory
Thesis Supervisor

Certified by_____________________
Nancy Leveson
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by_____________________
Arthur C. Smith
Chairman, Department Committee on Graduate Students
System Shutdown Techniques for Energy Efficient Real-Time Computation

by

Daniel Aaron Hartley

Submitted to the Department of Electrical Engineering and Computer Science May 19, 2000, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract
This thesis presents a technique to reduce microprocessor power consumption in real-time systems. The technique is implemented in the system's task scheduler. The basic idea is when the microprocessor is idle it will be partially shutdown. Partial shutdown modes are standard features of currently available microprocessors. They allow the processor to consume a fraction of its normal mode power at the cost of disabling some of the processor's functionality. The other cost associated with partial shutdown modes is the time that it takes to transition in and out of these modes. This thesis will illustrate how to account for these costs while still providing the performance necessary for a real-time system.

Draper Laboratory Thesis Supervisor: Roger Racine
M.I.T. Thesis Supervisor: Nancy Leveson, Professor of Aeronautics and Astronautics
Acknowledgements

I would like to thank Roger Racine for his guidance, friendliness, patience, and knowledge, The Charles Stark Draper Laboratory for providing me with the resources to write and research this thesis. I would like to thank Professor Leveson for her time and guidance. Props to Dennis Gregorovic for hooking me up with some scanning action. I can’t forget my roommates Alex Kirshner and Louise Baker for the moral support and editing help that they provided. Thanks to my Mother and Father for providing me with some quality procrastination time on the phone. Thanks to Linus, my stepfather, for staying on my case. And finally to Bill, Ben, Chris, and Josh for holding down my city, Chicago, while I was away.
Contents

1. Introduction ................................................................. 5

2. Previous Work ............................................................ 6
  2.1 Minimizing Power at the Software Level ................... 6
  2.2 Minimizing Power at the Operating System Level ....... 7
  2.3 Extension of Previous Work to Real-Time Application ... 12

3. How To Integrate System Shutdown With Real-Time Scheduling 13
  3.1 Preemptive Versus Non-Preemptive Scheduling ............ 13
  3.2 Static Scheduling .................................................... 14
    3.2.1 Rate Monotonic Analysis ..................................... 14
    3.2.2 Power Saving In Statically Scheduled Systems ...... 15
  3.3 Dynamic Scheduling ................................................ 15
    3.3.1 Earliest Deadline First (EDF) .............................. 15
    3.3.2 Highest Priority First (HPF) ............................... 16
  3.4 Handling of Dynamic Events .................................... 17
    3.4.1 Full Immediate Handling .................................... 17
    3.4.2 Cyclic Polling .................................................. 18
    3.4.3 Deferred Periodic Handling ................................ 18
  3.5 Incorporating Power Saving Into Dynamic Scheduling .... 19

4. The PowerPC 604e Microprocessor .................................. 19
  4.1 Power Saving Modes of the PowerPC 604e .................. 20
  4.2 Doze Mode ......................................................... 21
  4.3 Nap Mode ........................................................... 22

5. Example of a Scheduler Implementation .......................... 22

6. Example Application Running Under the Scheduler .......... 23
  6.1 Example Specifications ............................................. 23
  6.2 Statically Scheduled Tasks ....................................... 24
  6.3 Filling in the Spaces ............................................... 25
  6.4 Dynamically Arriving Tasks ..................................... 26

7. Conclusions ............................................................ 27

8. Further Work .......................................................... 28
1. Introduction

As the demand for complex embedded systems, portable computing devices, and general computer usage increases, the power efficiency of computing systems has become as significant as the size and performance of these systems. Minimizing computer power consumption has both environmental and functional implications. From an environmental perspective, the proliferation of computers causes an increase in the consumption of electricity and places a greater burden on the earth's natural resources. From a functional perspective, users typically find portable devices more useful and more convenient to use as the time between recharge increases. Another application in which power is at a premium is in the embedded systems used to control space vehicles; this is due to the limited capacity that such vehicles have for the production of electricity. This thesis presents a technique to reduce microprocessor power consumption in real-time systems. The technique is implemented in the system's task scheduler. The basic idea is when the microprocessor is idle it will be partially shutdown. Partial shutdown modes are standard features of currently available microprocessors. They allow the processor to consume a fraction of its normal mode power at the cost of disabling some of the processor's functionality. The other cost associated with partial shutdown modes is the time that it takes to transition in and out of these modes. This thesis will illustrate how to account for these costs while still providing the performance necessary for a real-time system.
2. Previous Work

To completely minimize the power consumption of a system, it is important to optimize the efficiency of the system across all levels of system design. Recent efforts to reduce power consumption have focused primarily on designing reduced power memory and logic at the hardware level. Two good overviews of power minimization methods are given by Devedas and Malik and by Pedram [Devedas-95] [Pedram-96]. While hardware optimization is essential to the reduction of system power consumption, it is imperative that operating system and application software also be written with power consumption in mind if a truly optimal system is to be realized.

2.1 Minimizing Power at the Software Level

Flinn and Satyanarayanan give an excellent discussion of a system that works to reduce power consumption through operating system and application cooperation [Flinn-99]. The most distinctive aspect of their work is that the system allows the user to specify a desired battery life. Based upon this specification and its knowledge of the power remaining in the battery, the operating system will control the fidelity of the applications that are running in order to meet the user's desired computing time. The concept of fidelity exists due to the nature of the applications that are run on the system. The examples that are presented include a video player, a speech recognizer, a map viewer, and a web browser. By reducing the quality of the medium that an application deals with, it is possible to realize power savings. Examples of variations in fidelity are the use of lossy video or image compression, the varying of the size of the display window, or the use of smaller vocabulary sets and less complex acoustic models in the
case of the speech recognizer application. The main shortcoming of this method of power reduction is that the concept of fidelity does not necessarily apply to all applications. For example, a word processing program that only saved 90% of the text that was input (due to lossy compression) would not be very useful.

2.2 Minimizing Power at the Operating System Level

Operating systems can reduce power consumption when running on processors that provide software-controlled enabling of reduced-power modes. Such "sleep" or "idle" reduced-power modes are becoming standard features of most commercially available processors. Several issues surface in the design of operating systems that utilize such processor power saving modes. When should the processor be put into reduced-power mode? How long should it be placed in reduced-power mode? If several levels of power saving modes are available, which gradation is appropriate? Conventional approaches to utilizing reduced power modes are used in Apple's Macintosh PowerBook line of portable computer. The PowerBook uses three power saving modes: rest, sleep, and shutdown [Apple-92]. Power saving is initiated after the processor has been idle for two seconds. After two idle seconds the computer enters rest mode, at which point the processor begins power cycling. During power cycling, the processor registers are saved and the processor is powered down. After 1/60 of a second, power is restored to the processor. If no I/O activity has occurred, the cycle is repeated; otherwise, state is returned to the processor and computation begins. The computer will enter sleep mode if it has been idle for a user-specified amount of time. This time is usually on the order of minutes. Upon entering sleep mode, disk ports, I/O ports, and the sound circuitry are all
powered down. A significant power and time overhead is incurred by entering sleep mode, as the hard disk must be spun down to enter, and spun back up to exit sleep mode. In shutdown mode the computer is essentially off, with only a parameter RAM remaining powered. Shutdown is only entered upon the user's direct command. Srivastava et al. found that when running an X server on the PowerBook the enabling of rest mode power cycling resulted in power savings of x1.5-x2 [Srivastava-96]. It is harder to quantify the effects of sleep and shutdown modes on power consumption, because these are user dependent rather than application dependent. The power cycling method results in moderate power savings, but leaves room for improvement provided that information can be gathered about the length and frequency of idle processor periods.

Srivastava et al. propose a technique to determine whether or not to switch a processor into sleep mode based upon a predictive formula [Srivastava-96]. Upon entering an idle period, the formula is used to predict how long the processor will remain inactive, based upon previous active and idle time lengths. Stemming from a desire for simplicity and minimal resource usage, a quadratic regression model was used to predict idle period. In addition to the quadratic model, an even simpler formula was proposed. Based upon experimental analysis of X-Server traces it was found that large busy periods were followed by short idle periods, but that shorter busy periods were followed by a uniform distribution of idle periods. This observation led to a simple heuristic: if the preceding busy period was less than the delay overhead caused by transitioning to sleep mode and back, then the processor should be put into sleep mode; otherwise, it should remain in active mode. Experimental results were compiled for each formula and compared to the Mac PowerBook two-second strategy, which is used as the baseline.
The results are shown for delay overheads that span three orders of magnitude. Slowdown factor and power savings are used as metrics to evaluate each prediction formula. In terms of slowdown factor, the quadratic model performs very well when the delay overhead times are on the order of 10 milliseconds or less. The quadratic model starts to show large slowdown when delay overhead is 100 milliseconds or larger. In terms of energy reduction, the quadratic model far outperforms the non-predictive 2-second threshold method across all values of delay overhead.

**Energy Reduction Factor**

![Energy Reduction for several Shutdown Methods. Chart Compiled From [Srivastava-96].](image)

Figure 1: Energy Reduction for several Shutdown Methods. Chart Compiled From [Srivastava-96].
Figure 2: Slowdown for several Shutdown Methods. Chart Compiled From [Srivastava-96].

Hwang and Wu extend these predictive shutdown methods to be generally applicable for event-driven tasks [Hwang-97]. They experiment with the use of an exponential-average approach to predict the duration of idle periods on four different applications. They find promising results when applying the formula to X-server, Netscape, telnet, and tin applications. In addition to the exponential-average formula, Hwang and Wu also test a version of this formula that is modified to provide for a pre-wake up time margin, so as to minimize system slowdown by waking up the system far enough ahead of time to allow restoration of processor state. Although this formula helps combat system slowdown, it has a slightly adverse effect upon power reduction.
Figure 3: Energy Reduction for Several Shutdown Methods. Chart Compiled from [Hwang-97].

Figure 4: Slowdown for Several Shutdown Methods. Chart Compiled From [Hwang-97].
2.3 Extension of Previous Work to Real-Time Application

Herein, I examine how predictive shutdown techniques can be incorporated into a real-time scheduler responsible for delegating processor time between guidance, navigation, and control tasks. The majority of tasks running in guidance, navigation, or control systems are periodic rather than event-based. A completely periodic schedule of tasks more easily resolves issues such as when and for how long the processor ought to be shut down. If we envision a priority-based task scheduler, a simple solution is to run a pseudo-task at the lowest priority level that is responsible for switching the system into power saving mode whenever it is activated as discussed by Lorch [Lorch-97]. With only periodic tasks running, this pseudo-task would determine exactly how long the idle period would last, and would set a timer to wake the system back up with just enough time to allow for power-up overhead before the next task was to begin running. This method can also be extended to work for mixed event/periodic-based tasks. If event-based tasks require a very fast response time, it may be necessary to limit the processor to modes of operation with little wake up overhead. For example, let’s say the system’s processor has two power saving modes, light sleep and heavy sleep. The processor needs 10 time units to wake up from a heavy sleep, but only 5 time units to wake up from a light sleep. If an event-based task was required to begin computation within 7 time units after an interrupt was asserted, then it would be necessary to ban the processor from heavy sleeps unless we could be sure that the event-based task was not running at that time.
3. How To Integrate System Shutdown With Real-Time Scheduling

The problem of scheduling tasks on a real-time system can be divided along several lines. Preemptive versus non-preemptive, dynamically versus statically scheduled, and uni-processor versus multi-processor systems are all possible. Although my work focuses upon single processor systems, the technique that I present could be extended to systems with multiple processors.

3.1 Preemptive Versus Non-Preemptive Scheduling

One of the first considerations when designing a real-time scheduler is whether the system will allow a higher priority job to interrupt a lower priority job. If the scheduler provides for this, it is referred to as preemptive. In a non-preemptive scheduler, upon gaining control of the processor, all jobs are allowed to run until completion. This means that, at times, a higher priority job may have to wait for a lower priority job to finish before it can execute. Non-preemptive schedulers are easier to implement but more difficult to analyze [Briand-99]. The implementation is easier, because the scheduler only needs to worry about invoking a new job upon the completion of a previous job or while the processor is idle. Non-preemptive schedulers are more difficult to analyze, because rate monotonic analysis cannot be used. Since guidance, navigation, and control systems inherently involve the timely execution of certain critical tasks, preemptive schedulers are usually necessary for these types of systems.
3.2 Static Scheduling

The simplest form of real-time scheduling is done off-line, when all task deadlines and execution times are known before the system is deployed. In this situation, an optimal solution to task scheduling is given by the rate monotonic scheduling algorithm [Briand-99].

3.2.1 Rate Monotonic Analysis

Rate monotonic scheduling refers to the method of assigning higher priorities to tasks with shorter periods. Liu and Layland assert that a set of n tasks will be able to meet their specific deadlines if the following equation holds true [Lui-73].

\[
\frac{\sum_{i=1}^{n} C_i}{T_i} \leq U(n) = n(2^n - 1)
\]

(Where \(C_i\) refers to the execution time of task \(i\), and \(T_i\) refers to the period of task \(i\).) In addition, the scheduler must be preemptive, no interdependencies can exist between tasks, task deadlines must be equal to their periods, and scheduling must be done according to the rate monotonic algorithm (tasks of shorter period get higher priority). The Liu and Layland theorem can prove the feasibility of scheduling a set of tasks, but it cannot prove that a set of tasks is unschedulable.

When the schedulability of a set of tasks cannot be proved by the Liu and Layland theorem, it may still be possible to prove schedulability using another theorem. Lehoczky, Sha, and Ding proved that, “If a set of independent, periodic tasks is started synchronously and each task meets its first deadline, then all future deadlines will be met” [Lehoczky-87]. An implicit assumption of this theorem is that all tasks have fixed execution times. This theorem is less conservative than the Liu and Layland theorem,
because it allows for the case of partial or fully harmonic tasks that may actually be schedulable while having a processor utilization of up to 100% (for a fully harmonic task set) [Briand-99].

3.2.2 Power Saving In Statically Scheduled Systems

Introducing power saving into a statically scheduled system is conceptually simple. Once all tasks have been successfully scheduled, the remaining spaces, idle processor time, can be filled in by putting the processor into a power saving mode. The processor should be put into the deepest sleep mode that the idle period and processor overhead will allow for. It is crucial that the scheduler perform the necessary tasks to ensure a timely wake up for the processor, such as setting a timer to reactivate the processor in time for its next job.

3.3 Dynamic Scheduling

Dynamic scheduling introduces a much greater degree of complexity than static scheduling, but also allows for the flexibility of being able to add tasks to the system after it has been deployed. Dynamic scheduling algorithms can use pre-computed priorities or assign priorities to tasks dynamically.

3.3.1 Earliest Deadline First (EDF)

The earliest deadline first algorithm assigns priorities based on the proximity of a task’s deadline. The highest priority is given to the task with the nearest deadline, the second highest priority to the task with the second nearest deadline, and so on. EDF is an
optimal priority assignment algorithm for dynamic scheduling assuming that it is implemented on a single processor system that is not overloaded. The optimality of EDF was proved by Dertouzos [Dertouzos-74]. EDF makes the scheduler responsible for choosing task priority levels. Power saving capabilities could be added to an EDF scheduler by creating a pseudo-task with a deadline that is always later than every other task. The problem with using this method in a dynamic task environment is that the processor must be woken up regularly to see if any dynamic tasks have arrived, or if a periodic task needs to be scheduled again. One way to address this problem is to provide dynamic tasks with the capability to wake the processor from power saving mode through the assertion of an interrupt. I will defer a full discussion of dynamic event handling to section 3.4.

3.3.2 Highest Priority First (HPF)

On the opposite side of the spectrum, highest priority first puts the user in charge of setting the priority level for each task. This scheduler is much easier to implement. It allows the user a much greater degree of flexibility, and is useful when there is a known ranking of precedence among the tasks. Power saving could be incorporated in an HPF scheduler by creating a pseudo-process that is assigned the lowest possible priority. This pseudo-process would put the processor into sleep mode. Just as with an EDF scheduler, the processor would need to be woken up periodically to see if any tasks were waiting to be scheduled.
3.4 Handling of Dynamic Events

To implement a scheduler capable of handling dynamically arriving tasks, some thought must be given to the mechanism that will recognize that a dynamic task has arrived and the mechanism that will schedule the incoming task. There are several ways that the responsibility for these two functions can be divided.

3.4.1 Full Immediate Handling

With full immediate handling, incoming tasks are handled at the hardware interrupt level or at the software level provided that interrupts are disabled during critical sections. If dynamic task arrivals are handled at the hardware interrupt level, then the processor will trap to supervisor mode, whereupon the new task will be scheduled. This may prove to be detrimental to the performance of the real-time system if the time it takes to switch in and out of supervisor mode is significant. The advantages of this method are that it is simple to implement, easy to model using Rate Monotonic Analysis, and consumes the minimum possible amount of processor time. The main disadvantage is that a burst of incoming tasks could lead to continuous execution of the interrupt service routine, potentially causing other tasks to miss their deadlines [Briand-99]. An alternate approach is to implement full immediate handling at the software level. Instead of switching to supervisor mode, the processor will remain in user mode, but interrupts will be disallowed for a short time while shared data structures of the scheduler are modified to include the new task. Software implementation of full immediate handling significantly reduces processor overhead. At the same time, it is crucial to ensure that
interrupts are disabled for the shortest amount of time possible, so that system responsiveness is not compromised.

### 3.4.2 Cyclic Polling

Under this implementation, a polling task that runs periodically, checks to see if any new tasks have arrived since it ran last. If so, the new task is processed by the scheduler. The advantages of cyclic polling include ease of implementation and avoidance of using hardware interrupts (which have higher priority than software level tasks). The main drawback of this implementation is that it can only handle one incoming task per period. It is also important to consider the overhead introduced by the polling task, as it must execute partially even when no new task has arrived [Briand-99].

### 3.4.3 Deferred Periodic Handling

Deferred handling breaks the processing of incoming tasks into two parts. The first part of the processing occurs at the hardware interrupt level, and consists only of the processing necessary to allow for the arrival of another dynamic task. The remainder of the processing is passed on to a deferred handler that executes periodically only if a dynamic task has arrived within the last period. This approach minimizes the amount of processor time required by the interrupt service routine, thereby reducing the chance that a burst of incoming tasks will cause tasks that are already within the system to miss deadlines. The advantage of deferred handling is that it can process several incoming tasks in one cycle, while only spending a short time at the hardware priority level. The disadvantage of this approach is that it is necessary to determine a worst case inter-arrival
time for both the hardware level handler and the software level handler in order to use Rate Monotonic Analysis [Briand-99].

3.5 Incorporating Power Saving Into Dynamic Scheduling

In each of the above methods of implementing dynamic scheduling, it is necessary to consider what response time is required for incoming tasks. The most extreme possibility is that a task arrives with a deadline and computation time that require it to be executed immediately. The only dynamic scheduling implementation that can ensure that the incoming task will be executed as quickly as possible is full immediate handling. In this case, it is important that the amount of time required to bring the processor out of sleep mode and then schedule the dynamic task to execute immediately, does not cause the desired response time to be violated. Degradation of dynamic response time due to wake up overhead is an issue only in systems where dynamic tasks are given the highest priority level. At any other priority level, there is always a chance that the dynamic task will have to wait for another task to complete before it will be allowed to execute. In general, the time it takes to execute a task will be longer than the time required to bring the processor out of sleep mode.

4. The PowerPC 604e Microprocessor

As an example, I will discuss how to implement power saving control in a real-time operating system running on a PowerPC 604e microprocessor based system.
4.1 Power Saving Modes of the PowerPC 604e

In addition to its Dynamic Power Management (DPM) feature, which automatically turns chip execution units on and off based upon the instruction stream, the PowerPC 604e supports two levels of static power management [PowerPC-98]. Doze mode provides for the least amount of power savings, but maintains cache coherency, snoop logic, and a wake up timer. In Nap mode all logic is disabled except the wake up timer. Since the processor will not be executing during Doze or Nap mode, system memory may be powered down when the processor enters either mode [Gary-94]. Although I could not obtain power consumption figures for the PowerPC 604e, the chart below for the PowerPC 603 approximates the factor of savings that can be achieved in Doze and Nap modes. The PowerPC 604e and PowerPC 603 share a common architecture.

<table>
<thead>
<tr>
<th>Power Mode</th>
<th>50 MHz</th>
<th>66 MHz</th>
<th>80 MHz</th>
<th>Average Power Consumption Ratio (compared to DPM only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPM only</td>
<td>1.52 W</td>
<td>1.89 W</td>
<td>2.54 W</td>
<td>1</td>
</tr>
<tr>
<td>Doze Mode</td>
<td>0.242 W</td>
<td>0.307 W</td>
<td>0.366 W</td>
<td>0.155 (factor of 6.44 savings)</td>
</tr>
<tr>
<td>Nap Mode</td>
<td>0.089 W</td>
<td>0.113 W</td>
<td>0.135 W</td>
<td>0.057 (factor of 17.49 savings)</td>
</tr>
</tbody>
</table>

Table 1: PowerPC 603 Typical Power Consumption [Suessmith-94]

Switching between Normal, Doze, and Nap modes require external system support. Logic must be implemented to control the RUN signal and respond to the HALTED signal, both of which are external interface signals of the PowerPC 604e. The following diagram illustrates which state transitions are possible between the three modes of operation.
4.2 Doze Mode

While the PowerPC 604e is in Doze mode, data cache, snoop logic, and time base/decrementer all remain active. Keeping the data cache active allows the processor to maintain cache coherency while in Doze mode [Gary-94]. Time base/decrementer logic is used to set the processor to wake up after a specified amount of time. In order to switch the 604e from normal mode to doze mode, the system must assert the RUN signal for at least 10 bus cycles. At this point, the following software power management sequence can begin.

Sync
Mtmr (set the POW bit in the Machine State Register only)
Isync
Branch back to sync instruction [PowerPC-98]

Before invoking this sequence of instructions, it is necessary to ensure that external interrupts have been enabled, so that the system will be able to wake up the processor when it is necessary. Upon setting of the POW bit, the processor waits for its internal state to become idle and then asserts the HALTED signal. The system must hold the
RUN signal asserted for at least 10 bus cycles after the HALTED signal has been asserted by the processor [PowerPC-98].

Assuming the external interrupt bit has been set in the Machine State Register, the processor will return from doze mode to normal if either an external interrupt or a decrementer interrupt occurs. In order to exit doze mode, it is essential that the exception handler return to instructions outside of the power management loop, whereupon processing will continue normally [PowerPC-98].

4.3 Nap Mode

The only logic that remains active in Nap mode is the time base/decrementer [Gary-94]. Nap mode can only be entered once the processor is already in doze mode. Once the processor is in doze mode, the system must negate the RUN signal and prevent bus grants for at least 10 more bus cycles [PowerPC-98].

From nap mode the processor can transition to doze mode or all the way back to normal mode. Since either transition takes the same amount of time, my scheduler will only allow transitions from nap mode to normal mode. In other words, based on the amount of idle time available, my scheduler will put the processor into the deepest sleep possible. In the case of a dynamic task arrival, doze or nap mode will yield equivalent wake up times. External interrupts and decrementer interrupts will both cause the processor to wake up from nap mode to normal mode [PowerPC-98].
5. Example of a Scheduler Implementation

As an example, I will focus on incorporating power-saving functionality into a real-time software scheduler that could be used in a space vehicle. While it may be desirable to dynamically add tasks to the system’s processor, I will assume that the most time-critical tasks such as those required for the guidance, navigation, and control of the vehicle will be completely specified before the vehicle ever leaves the ground. A mixture of static and dynamic scheduling is suitable for this scenario. Static tasks will be scheduled using a preemptive rate monotonic scheduler, as it is simple to implement and its optimality has been shown. Cyclic polling will be used to manage the scheduling of tasks that arrive dynamically. The creator of the new task will be able to set its priority in comparison to tasks that are already present in the system. While it is expected that dynamic tasks will be given lower priorities than those assigned statically to the guidance, navigation, and control tasks, it will be possible for the user to assign a new task a higher priority in the event that something unexpected occurs.

6. Example Application Running Under the Scheduler

The following will provide an extremely simplified example to demonstrate how an application would run with the above real-time scheduler. Although the example is much less complex than most real-time systems, my aim is to show how all the important issues could be handled if one were to implement the scheduler.
6.1 Example Specifications

Imagine that the system is designed so that it is known before deployment that three tasks must run periodically. These tasks are critical to the operation of the system. A missed deadline could be fatal to the system (vehicle crashes, etc.). All three tasks have deadlines that are equal to their periods. While this is not a necessary condition for a real system, it makes our example much easier to analyze. Task 1 has an execution time of 5 time units and a period of 100. Task 2 takes 20 time units to execute and has a period of 200. Finally, task 3 executes in 100 time units and has a period of 500. In addition there is a cyclic polling task to handle dynamically arriving tasks. The polling task has a period of 50 time units and an execution time of 2 units when no dynamic tasks have arrived, or 10 units if a dynamic task has arrived. The overhead incurred by switching between any of the three tasks or the polling task is 1 time unit. Although I have specified exact execution times for each task, it is unusual to have this much information in a real system. In general, maximum processing times are known for each task. Maximum processing times are generated by analyzing instruction sequences, input/output dependencies, and inter-process dependencies. While it is necessary for the user to know the maximum execution times of all tasks to avoid overloading the processor, the scheduler does not need to know this information. All scheduling is done according to priority level. Two levels of power saving are provided by the microprocessor. Light power-saving mode takes 2 time units to enter and reduces the power consumption of the processor by half. Heavy sleep mode reduces the power consumption of the processor by a factor of ten. In order to enter the heavy power-saving mode, the microprocessor must already be in light mode. Once the processor is in light
mode, another 2 time units are needed to enter heavy mode. A transition from normal operation to heavy mode will, therefore, take 4 time units. Waking up the processor from light or heavy power-saving mode requires only 2 time units.

6.2 Statically Scheduled Tasks

The Polling task and tasks 1, 2, and 3 are scheduled off-line using the rate-monotonic approach. The polling task has the shortest period, and is therefore given the highest priority. Task 1 gets the second highest priority, then 2. Task 3 has the longest period, and therefore the lowest priority. Assuming all three tasks are begun at time 0, the processor utilization will follow the diagram below for the first 500 time units. While the period between 500 time units and one thousand time units will look slightly different (due to the fact that only tasks, 1, 3, and the polling task will need to be scheduled at the 500 time units mark; task 2 waits until the 600 time unit mark), all tasks are guaranteed to meet their deadlines. The schedulability is proved by the fact that every task has met it first deadline under the condition that all tasks were begun synchronously [Lehoczky-87].

![Diagram of task scheduling](image.png)

Figure 6: Example Timeline

In the diagram above, P denotes the polling task, while 1, 2, and 3 refer to tasks 1, 2, and 3 respectively. The start and end times of each task are also shown.
6.3 Filling in the Spaces

The next step of the scheduling process involves determining when the processor should be switched to a reduced power mode, which mode to switch to, and when it needs to awaken. The following diagram illustrates the time intervals during which the processor will be powered down.

![Diagram showing time intervals for power saving modes.]

**Figure 7: Example Timeline with Power Saving**

H and L refer to the heavy and light power saving modes respectively. Light sleep mode will only be chosen over heavy sleep mode when the processor is idle for 6 time units or less. This is due to the fact that heavy sleep mode has a total overhead of 6 time units, whereas light sleep mode has a total overhead of only 4 time units. If the processor is idle for 4 time units or less, then the scheduler will keep the processor in normal mode.

In this example, only one idle interval was small enough to require that the processor be powered down to light sleep mode instead of fully entering heavy sleep mode. Processor power consumption is reduced to \((243 \times 1) + (2 \times 0.5) + (255 \times 0.1) / 500 = 0.539\), or 53.9% of what it would be if no power saving had been used.
6.4 Dynamically Arriving Tasks

Upon submission of a dynamic task, the user can specify the priority level of the new task in relation to the existing tasks. It is the user's responsibility to ensure that tasks submitted with higher priority than the existing tasks will not prevent the existing tasks from meeting their deadlines. Due to this fact, it is recommended that the user assign dynamic tasks lower priority than the original, mission-critical tasks. Since the polling task runs every 50 time units, the system can only handle incoming dynamic tasks with a minimum inter-arrival time of 50 time units. It is important to realize that this minimum inter-arrival time would have to be doubled if the polling task were not the highest priority task [Briand-99]. If it is necessary to provide for shorter inter-arrival times for dynamic tasks, several things can be done. The period of the polling task can be reduced. Another approach is to provide full immediate handling or deferred periodic handling of dynamic task arrivals. Under these approaches, smaller or no inter-arrival time is imposed upon incoming dynamic tasks.

![Figure 8: Example Timeline with Power Saving and a Dynamic Task Arrival](image)

The diagram above shows the arrival of a dynamic task at the 185 time unit mark. The dynamic task has a lower priority than tasks 1, 2, and 3. The polling task, which executes every 50 time units runs from between the 200 and 210 time unit marks. During
this time the dynamic task is scheduled, and the hardware that notifies the polling task of new arrivals is reset. The execution time of the dynamic task is 100 time units, and the task is done executing 171 time units after its arrival. Processor power consumption is reduced to $(338*1)+(2*0.5)+(160*0.1)/500 = 0.71$, or 71% of what it would be if no power saving had been used.

7. Conclusions

By scheduling microprocessors to partially shutdown during idle periods, real-time systems can reduce power consumption while maintaining system specified levels of performance. The key is to understand the response time necessary for dynamic tasks, and implement a system that will be able to wake up quickly enough to meet the deadline. In cases where relatively short dynamic response times are necessary, an immediate handling approach may be required. By powering down other system logic, such as memory circuitry, in conjunction with the processor, real-time systems will be able to realize significant power savings.

8. Further Work

It had been my original intention, had time permitted, to implement a power-saving real-time scheduler, and compare the power consumption of typical guidance, navigation, and control system to the power consumption of the same application run without power-saving capabilities. It would be extremely interesting to test these ideas on a number of different microprocessors and also for a number of different real-time
applications. It would also be very interesting to see how this technique could be applied to multi-processor real-time systems.
References


Programming in a Hard Real-Time Environment,” Journal of the 
Association for Computing Machinery (ACM) Vol. 20, No. 1, 


[Pedram-96] Pedram, M., "Power Management in IC Design: Principles and 
Applications," ACM Transactions on Design Automation of 
Electronic Systems, volume 1, number 1, Pages 3-56, Jan., 1996.


Computing: Scheduling and Resource Management, Kluwer 

[Silberschatz-98] Silberschatz, Abraham, Galvin, Peter Baer, Operating System 

[Srivastava-96] Srivastava, M. B., Chandrakasan, A. P., Brodersen, R. W., 
"Predictive System Shutdown and Other Architectural Techniques 
for Energy Efficient Programmable Computation," IEEE 
Transactions on VLSI Systems Vol. 4 No. 1 March, 1996. Pages 
42-55.

[Stankovic-88] Stankovic, John A., Ramamritham, Krithi, Hard Real-Time 

[Stankovic-93] Stankovic, John A., Ramamritham, Krithi, Advances In Real-Time 
Systems, Computer Society of IEEE Press, Los Alamitos, CA, 
1993.

[Suessmith-94] Suessmith, Brad “PowerPC 603 Microprocessor Power 
Management,” Communications of The ACM, Vol.37, No.6, June 
1994.

[Tanenbaum-92] Tanenbaum, Andrew S., Modern Operating Systems, Prentice Hall, 