Low-Power Task Scheduling for Multiple Devices

Yung-Hsiang Lu, †*Luca Benini, Giovanni De Micheli*
CSL, Stanford University, USA. {luyung, nanni}@stanford.edu
† DEIS, Università di Bologna, Italy. lbenini@deis.unibo.it

Abstract

Power management saves power by shutting down idle devices. These devices often serve requests from concurrently running tasks. Ordering task execution can adjust the lengths of idle periods and exploit better opportunities for power management. This paper presents an on-line low-power scheduling algorithm for multiple devices. Simulations show that it can save up to 33% power and reduce 40% state-transition delays. This algorithm is robust under imperfect knowledge of future requests and timing constraints; therefore, it is applicable to interactive systems.

1. Introduction

Dynamic power management (DPM) shuts down unused devices to save power [3]. When serving requests (busy), a device must be in a high-power *working state*. When a device is not serving any requests (idle), it can be shut down and put into a *sleeping state* to save power. Studies show that more than 50% power can be saved by power management [9]. Power state changes are decided by a *power manager* (PM); PM wakes up a device to serve requests and shuts it down to save power. State changes take time and energy; consequently, a device should be shut down only if it can sleep long enough to compensate the performance and energy overhead.

In modern computers, requests are often generated by concurrently running tasks. For instance, hard disk IO's can come from a compiler, a text editor, or a file transfer program (`ftp`). Similarly, network transmission requests can be generated by an Internet browser or a `telnet` session.

Traditional power management focuses on predicting the lengths of idle periods and implicitly assumes that request arrival time cannot be changed [3] [9]. In reality, however, the lengths of idle periods can be adjusted by ordering task execution, i.e. by *scheduling* tasks. Even though scheduling is a standard feature in operating systems (OS), task scheduling for power management has not been well studied for OS-based power management (OSPM) [3].

Intuitively, scheduling for power management is to make idle periods clustered and long, instead of scattered and short, so that power management is applicable. Previous scheduling techniques focus on processors [8] [10] [14] or real-time systems [4] [12]. These algorithms deal with only one service provider— the processor; it is unclear how to extend them for multiple devices. The authors do not explain how to integrate the algorithms into existing systems. Furthermore, they unrealistically assume perfect knowledge of future requests.

This paper presents a greedy on-line scheduling algorithm to facilitate power management for multiple devices. It orders task execution such that devices can have continuous long idle periods to be shut down. We also show how to integrate this algorithm into existing systems. In addition to saving power, task scheduling has another benefit: clustered idle periods reduce the numbers of shutdowns, hence state-transition delays. Compared to a traditional scheduling algorithm which does not consider power management, simulations show that this algorithm can save 33% power and reduce 40% transition delays. The algorithm is robust under timing constraints and with imperfect knowledge of future requests. Therefore, it is applicable to interactive systems.

2. Background

2.1. Traditional Task Scheduling

Traditional scheduling algorithms do not consider power management. Instead, they focus on performance, fairness, and so on [13]. Figure 1 shows the flow of a typical OS scheduler, specifically the scheduler in Linux [1]. When the scheduler is invoked, it checks whether any queued task needs to run. The task queue is a mechanism for device drivers to request future execution, such as polling a device [11]. Then the scheduler executes interrupt handlers; after checking interrupts, it signals tasks whose timers expire. Afterwards, it considers task-specific requirements, such as timing constraints. The last two steps in the scheduler are to select a task with higher priority or with the largest unfinished time slice.

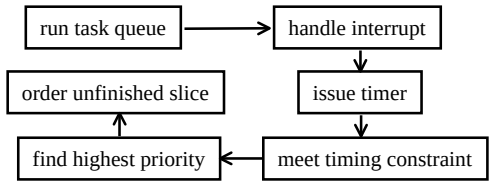

Figure 1: typical task scheduler

Symbol	Meaning
T	time slice
T_{be}	break-even time
T_o / E_o	transition time / energy overhead
P_w / P_s	power in working / sleeping state
Ψ	required device set (RDS)
Ψ_c	current RDS
k_Ψ	length of Ψ
$I_d(\tau)$	length of idle period for d at time τ
$E_d(l)$	minimum energy in duration l for d

Table 1: symbols and meanings

Time slice (also called *time quantum*) is the time unit allocated to each task [13]. A task may stop execution before using up its slice by, for example, issuing a system call. If no task can execute, the *idle process* is chosen. This paper focuses on scheduling for interactive systems without hard timing constraints. In contrast, real-time scheduling is more tightly constrained because it must meet hard deadlines [5].

2.2. Break-Even Time

Since changing power states takes time and extra energy, a device should be shut down only when the length of an idle period is long enough. The minimum length to save power by entering the sleeping state is call the *break-even time* (T_{be}). Let P_w and P_s be the power consumption in the working and the sleeping states ($P_w > P_s$). T_o and E_o are the time and energy overhead to shut down and wake up the device. T_{be} can be obtained by this formula: $P_w \cdot T_{be} = E_o + P_s \cdot (T_{be} - T_o)$; also, T_{be} must be larger than T_o. Consequently,

$$T_{be} = \max(\frac{E_o - P_s \cdot T_o}{P_w - P_s}, T_o) \quad (1)$$

T_{be} is a device characteristic unaffected by requests. We use subscripts to distinguish multiple devices; for instance, T_{be,d_1} is the break-even time of device d_1.

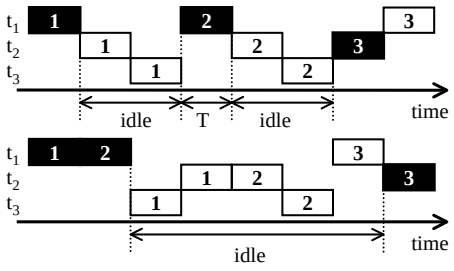

Figure 2: two schedules of three independent tasks. The second schedule reorders execution to make a long, continuous idle period.

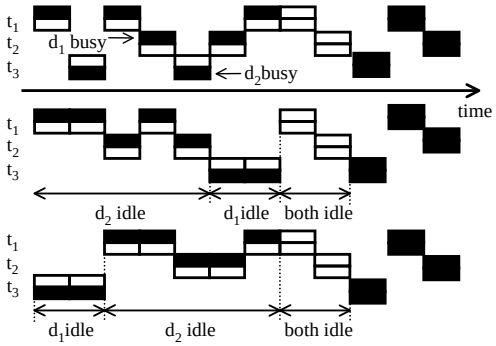

Figure 3: scheduling for multiple devices

2.3. Execution Order and Power Management

Figure 2 is an example of three independent tasks requiring service from a device; T is a time slice. A block indicates that a task is running. If the task generates requests, the block is filled; an unfilled block indicates that the task does not generate requests. In this figure, each task has multiple slices (labeled as 1,2, and 3); the scheduler cannot rearrange the slices *within* each task. When $2T < T_{be} < 5T$, the device can be shut down only in the second schedule because the idle periods are too short in the first schedule. Even if $T_{be} < 2T$, the second schedule is still advantageous. When $T_{be} < 2T$, the device will be shut down twice in the first schedule causing delay (T_o) and wasting energy (E_o) two times. In contrast, it is shut down only once in the second schedule. This example shows that, compared to short scattered idle periods, a long continuous idle period can save power and reduce delays.

In a system with multiple devices, scheduling becomes more complex. Figure 3 shows three schedules for three tasks and two devices. In the first schedule, idle periods are not continuous. The second schedule makes d_2 idle first and the third schedule makes d_1 idle first. If $5T < T_{be,d_2} < 7T$, d_2 can be shut down only in the third schedule. On the other hand, if $3T < T_{be,d_1} < 5T$, d_1 can be shut down only in the second schedule. This example shows that scheduling may cause one device to shut down while keeping another in the working state.

3. Problem Formulation

The scheduling problem for power management is to *arrange execution orders so that idle periods are clustered instead of scattered*. We first assume that the scheduler can perfectly predict whether a device is used by a task in the future (Ψ, defined below). Later, we will show how prediction accuracy affects power saving.

3.1. Required Device Sets

We define $\Psi(t, n)$ as the *required device set* (RDS) for running task t during its n-th time slice; $\Psi(t, n) = \{d : t \text{ uses } d \text{ at the } n\text{-th slice}\}$. In Figure 3, $\Psi(t_1, 1) =$

$\{d_1\}$, $\Psi(t_2, 3) = \phi$, $\Psi(t_3, 2) = \{d_2\}$, and $\Psi(t_2, 4) = \{d_1, d_2\}$. We call the current RDS Ψ_c; it is the RDS of the latest running task. Let $I_d(\tau)$ be the length of the idle period for device d up to time τ. $\Psi(\tau)$ is the RDS of the running task at τ. Obviously, $I_d(\tau) = 0$ if $d \in \Psi(\tau)$ since this device is used and cannot be idle. Table 2 shows the relationship between $I_d(\tau)$ and $I_d(\tau + 1)$.

3.2. Device Energy

$E(l)$ is the minimum energy of a device during an idle period of length l. If the idle period is long enough ($l > T_{be}$), the device is shut down; otherwise, it remains in the working state. $E(l)$ is the minimum energy during l; it can be achieved by an "oracle" power manager, such as off-line analysis of requests [6]. An oracle power manager has full knowledge of future requests and shuts down a device for all idle periods longer than T_{be}.

$$E(l) = \begin{cases} E_o + P_s \cdot (l - T_o) & \text{if } l > T_{be} \\ P_w \cdot l & \text{if } l \leq T_{be} \end{cases} \quad (2)$$

We add subscripts, $E_d(l)$, to distinguish different devices when necessary. Consider a sequence of N tasks to execute and their RDS's are Ψ_1, Ψ_2, ..., Ψ_N. These RDS's will create a series of idle and busy periods for each device. Let $(L_d[1], B_d[1], L_d[2], B_d[2] \ldots L_d[n_d], B_d[n_d])$ be the length of the series for device d; $L_d[1]$ and $B_d[1]$ are the lengths of the first idle and busy periods respectively. $L_d[0]$ and $B_d[0]$ are defined as zero. For example, in the third schedule of Figure 3, $(L_1[1], B_1[1], L_1[2]) = (2, 5, 2)$ for d_1 and $(L_2[1], B_2[1], L_2[2]) = (0, 2, 7)$ for d_2. The energy of these devices during the N slices is

$$E = \sum_d \sum_{i=1}^{n_d} (E_d(L_d[i]) + P_{w,d} \cdot B_d[i]) \quad (3)$$

The two terms express the energy during those idle and busy periods.

3.3. Scheduling for Energy Minimization

The goal of scheduling for power management is to find a sequence $(\Psi_1, \Psi_2, \ldots, \Psi_N)$ to minimize $\frac{E}{N}$. N is called *look-ahead*; it is the number of slices the scheduler considers in advance.

$d \in (\Psi(\tau), \Psi(\tau + 1))$?	$I_d(\tau)$ and $I_d(\tau + 1)$
(Y, Y)	$I_d(\tau) = I_d(\tau + 1) = 0$
(Y, N)	$I_d(\tau) = 0, I_d(\tau + 1) = 1$
(N, Y)	$I_d(\tau + 1) = 0$
(N, N)	$I_d(\tau + 1) = I_d(\tau) + 1$

Table 2: $\Psi(\tau)$, $\Psi(\tau+1)$ and $I_d(\tau)$ determines $I_d(\tau+1)$.

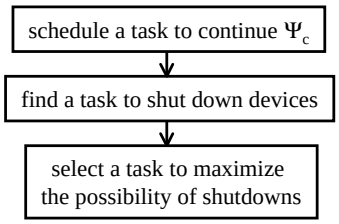

Figure 4: steps of selecting tasks

Theorem Optimal scheduling cannot be obtained by looking ahead a finite number of slices.

Due to space limit, we omit the proof in this paper. This theorem implies that we cannot find a globally optimal schedule without considering all slices. Some tasks, such as `tcsh`, may execute arbitrarily long; therefore, it is impossible to consider all slices in advance.

4. Scheduling for Power Management

4.1. Scheduling Boundaries

Since optimal scheduling is impossible by looking ahead a finite number of slices, we need to determine the number of slices to look ahead. We use a heuristic way for finding the number of slices; our algorithm finds the *scheduling boundary* of each task. It is the boundary when the task starts generating requests for a device which could have been idle previously. It is the largest m such that $\Psi(t, m-1) \supseteq \Psi(t, m)$ and $\Psi(t, m) \cap \Psi(t, m+1) \neq \Psi(t, m+1)$ for task t. In other words, $\Psi(t, m)$ is a subset of $\Psi(t, m-1)$ while $\Psi(t, m+1)$ is not a subset of $\Psi(t, m)$. A limit, M, can be set for the scheduling boundary so that $m < M$ to reduce the number of slices considered. For dependent tasks, M can cause one task to wait until the other is scheduled. These boundaries create a group of Ψ's to schedule.

4.2. Task Selection

Figure 4 shows the steps to select tasks. First, it selects a task whose RDS is the same as Ψ_c; then, it finds a task that can cause some devices to be shut down. If neither step succeeds, it selects a task with the best potential to save power in the near future. These steps follow the procedure in Figure 1, so certain properties in the original systems such as priorities can still hold. Whenever a task is selected, Ψ_c is updated accordingly.

The scheduler first tries to find a task whose RDS is the same as Ψ_c to avoid possible state transitions. If Ψ_c cannot continue because all remaining tasks have Ψ's different from Ψ_c, the scheduler finds a task that can shut down some devices that were busy previously. Because the scheduler always tries to continue Ψ_c, this step will find a set of tasks with the same Ψ. Suppose there are k_Ψ slices of tasks with the same Ψ and the current time is τ.

$I_d(\tau + k_\Psi)$ is updated by the rules in Table 2. This step tries to minimize the average power during the k slices by choosing Ψ:

$$\min_{\Psi} \sum_{d} \frac{E_d(I_d(\tau + k_\Psi))}{k_\Psi} \quad (4)$$

If no device can be be shut down, (4) is the same for all Ψ's. The scheduler finds a task with the best "potential" to save the most power. This potential is calculated by

$$\max_{\Psi} \sum_{d} (\frac{P_{w,d} - P_{s,d}}{T_{be,d} - I_d(\tau + k_\Psi)}) \quad (5)$$

It finds a Ψ that has the best chance in the future (small $T_{be,d} - I_d(\tau + k_\Psi)$) to save the most power (large $P_{w,d} - P_{s,d}$). If a Ψ can cause any device to be shut down, it will be selected by (4). Consequently, when the scheduler reaches (5), $T_{be,d} > I_d(\tau + k_\Psi)$ for all device and the denominator is always positive.

This algorithm takes a "greedy" strategy in selecting tasks; its complexity is $O(n \log n)$ where n is the number of Ψ's determined by the scheduling boundaries.

4.3. Example

In Figure 3, all tasks need both devices after $\tau = 9$. The scheduling boundaries for these tasks are 4, 3, and 2. The scheduler can select $\Psi_x = \{d_1\}$ or $\Psi_y = \{d_2\}$; their lengths are $k_x = 5$ and $k_y = 2$. Also, $I_{d_1}(k_y) = 2$, $I_{d_2}(k_x) = 5$, $I_{d_2}(k_y) = I_{d_1}(k_x) = 0$, and $\Psi_c = \phi$.

For simplicity, we assume that these devices consume the same power in either state ($P_{w,d_1} = P_{w,d_2}$ and $P_{s,d_1} = P_{s,d_2}$). Suppose $T_{be,d_1} = 3T$ and $T_{be,d_2} = 7T$. For Ψ_x, formula (4) gets $\frac{E_{d_1}(I_{d_1}(k_x))}{k_x} + \frac{E_{d_2}(I_{d_2}(k_x))}{k_x} = \frac{P_w \cdot 0}{5} + \frac{P_w \cdot 5}{5} = P_w$. The formula produces the same result for Ψ_y. Neither device can be shut down immediately; the scheduler moves to the third step. For Ψ_x, (5) is $(P_w - P_s) \cdot (\frac{1}{3} + \frac{1}{7-5})$; for Ψ_y, it is $(P_w - P_s) \cdot (\frac{1}{3-2} + \frac{1}{7})$. Ψ_y has better potential to save power; consequently, the algorithm selects Ψ_y and updates $\Psi_c = \{d_2\}$. The scheduler continues Ψ_c, so the second slice is also occupied by t_3. Now, due to the sequence inside t_1 and t_2, the only choice the scheduler has is to select tasks whose RDS's are $\{d_1\}$. Ψ_c is updated to $\{d_1\}$ and this RDS continues up to five slices. Finally, there are two slices that use neither devices. The result is shown at the bottom of Figure 3.

5. Experiments

Evaluating scheduling algorithms can be achieved by mathematical analysis, simulation, or implementation [13]. We use a Linux-based scheduling simulator for deterministic analysis of different workloads.

5.1. Timing Constraints

We define timing constraints as the maximum numbers of slices between two executions of a task. For example, if a slice is 5 millisecond and the timing constraint is 200 slices, the task will execute at least once every second. Timing constraints are essential for interactive systems to maintain responsiveness, such as reacting to mouse movement. We start with a constraint of 1000 slices and reduce it to 100 slices. The constraints limit the scheduler's choices; meanwhile, they provide shorter response time and improves interactivity.

5.2. Device Parameters and Task Generation

Four hypothetical devices are shown in Table 3. The system have five tasks generating requests. Studies show that requests are often bursty [2]; bursty requests are simulated by clusters using cluster-interval and cluster-length distributions in Table 4. Each distribution has two parameters: mean and standard deviation. For an exponential distribution, the standard deviation is determined by the mean, so "-" is shown in the table.

5.3. Power Saving and Overhead Reduction

Three scheduling algorithms are compared: base scheduling, task grouping, and task scheduling. The comparisons start by assuming that Ψ's are perfectly predicted; later, we show how imperfect prediction affects power saving. The base scheduling implements Figure 1 except interrupt handling. The task grouping algorithm improves the base algorithm by including the first step in Figure 4; the task scheduling algorithm uses all three steps. After the execution orders are determined, a 2-competitive power manager (2CPM) decides power states. A 2CPM is an on-line power management algorithm using $T_{be,d}$ as the timeout value; it consumes at most twice of power compared to an oracle power manager [7]. Table 5 summarizes the simulation results. These devices consume totally 30 W in their working states. Approximately 10% power can be saved when applying power management to the base scheduling.

Compared to the base scheduling, additional 20% and 33% power can be saved by the grouping and the scheduling algorithms. Because the grouping algorithm does not consider which Ψ follows Ψ_c, it can reduce only 10% state changes. The scheduling algorithm can reduce the number of state changes by more than 40%. Since state changes cause delay and consume energy, fewer changes reduce state-transition overhead (T_o and E_o). In other words, task scheduling can *save power*

Device	P_w	P_s	T_o	E_o	T_{be}
d_1	8	2	5.5	88	12.8
d_2	10	1	4.4	91	9.6
d_3	5	0.5	0.8	21	4.6
d_3	7	1.5	10	62	8.5

Table 3: hardware parameters. time unit: T

task	device			
	1	2	3	4
1	(N, 40, 20) (U, 10, 5)	(U, 40, 30) (N, 10, 5)	(E, 60, -) (U, 10, 5)	(E, 50, -) (E, 20, -)
2	(E, 50, -) (E, 10, -)	(N, 50, 40) (E, 20, -)	(U, 40, 20) (N, 30, 20)	(N, 50, 20) (U, 15, 10)
3	(E, 60, -) (N, 12, 20)	(U, 20, 6) (U, 10, 5)	(N, 50, 20) (N, 20, 15)	(N, 30, 10) (U, 20, 15)
4	(E, 80, -) (E, 10, -)	(E, 90, -) (E, 15, -)	(N, 70, 40) (N, 20, 20)	(U, 90, 30) (U, 10, 10)
5	(U, 90, 60) (E, 15, -)	(N, 50, 20) (N, 20, 15)	(U, 60, 30) (E, 12, -)	(E, 100, -) (N, 15, 10)

Table 4: cluster-interval and cluster-length distributions. Distribution: U– uniform; E– exponential; N– normal.

and reduce overhead. When timing constraints become tighter, the scheduler has fewer choices in selecting tasks. Our simulations show that the scheduling algorithm can save 20 % power when the constraint is 10 times tighter. Finally, we consider inaccurate prediction of Ψ's because an on-line algorithm unlikely has perfect knowledge of Ψ's in advance. A prediction is inaccurate if an actual RDS is different from the predicted one. Inaccurate prediction may make idle periods shorter than expected and wake up devices earlier. Figure 5 shows power ratio compared to base scheduling when the prediction accuracy changes. While less power can be saved when accuracy deteriorates, the algorithm can still save nearly 20% power when the accuracy reduces by 10%. Because of its robustness under timing constraints and imperfect knowledge of future requests, this algorithm can be applied to interactive systems.

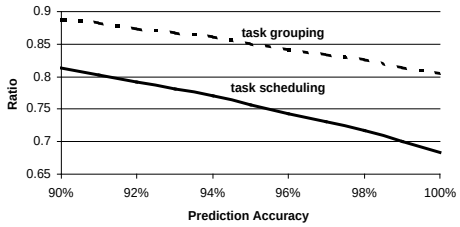

Figure 5: ratio of power consumption for different prediction accuracy when timing constraint is 500-slice.

6. Conclusions

We present a scheduling algorithm that controls the lengths of idle periods to exploit the opportunities of power management. This algorithm saves power and reduces state-transition overhead. Simulations show that it can save 33% power and reduce 40% state changes. It is robust under timing constraints and with imperfect knowledge of future requests.

timing constraint	power			change ratio	
	P_b	P_g	P_t	R_g %	R_t %
1000	27.0	21.8	18.0	90.8	57.0
500	27.0	21.8	18.5	90.9	61.3
100	27.0	21.9	21.5	91.7	84.9

Table 5: power consumption and ratio of state changes. P_b: base; P_g: grouping; P_t scheduling. R_g, R_t: ratio of numbers of state changes to the base scheduling.

7. Acknowledgments

This work was supported in part by MARCO/DARPA Gigascale Silicon Research Center and in part by NSF under contract CCR-9901190.

8. References

[1] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals.* Addison-Wesley, 1997.

[2] L. Benini, A. Bogliolo, S. Cavallucci, and B. Ricco. Monitoring System Activity for OS-Directed Dynamic Power Management. In *International Symposium on Low Power Electronics and Design*, pages 185–190, 1998.

[3] L. Benini, A. Bogliolo, and G. D. Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Transactions on VLSI Systems*, March 2000.

[4] J. J. Brown, D. Z. Chen, G. W. Greenwood, X. Hu, and R. W. Taylor. Scheduling for Power Reduction in a Real-Time System. In *International Symposium on Low Power Electronics and Design*, pages 84–87, 1997.

[5] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer, 1997.

[6] E.-Y. Chung, L. Benini, A. Bogliolo, and G. D. Micheli. Dynamic Power Management for Non-Stationary Service Requests. In *Design Automation and Test in Europe*, pages 77–81, 1999.

[7] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Nonuniform Problems. *Algorithmica*, 11(6):542–571, June 1994.

[8] J. R. Lorch and A. J. Smith. Scheduling Techniques for Reducing Processor Energy Use in MacOS. *Wireless Networks*, 3(5):311–324, 1997.

[9] Y.-H. Lu, E.-Y. Chung, T. Šimunić, L. Benini, and G. D. Micheli. Quantitative Comparison of Power Management Algorithms. In *Design Automation and Test in Europe*, 2000.

[10] G. Qu and M. Potkonjak. Power Minimization using System-Level Partitioning of Applications with Quality of Service Requirements. In *ICCAD*, pages 343–346, 1999.

[11] A. Rubini. *Linux Device Drivers*. O'reilly, 1998.

[12] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *DAC*, pages 134–139, 1999.

[13] A. Silberschatz and P. B. Galvin. *Operating System Concepts.* Addison-Wesley, 1994.

[14] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.