Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems

Youngsoo Shin and Kiyoung Choi
School of Electrical Engineering
Seoul National University
Seoul 151-742, Korea

Abstract

Power efficient design of real-time systems based on programmable processors becomes more important as system functionality is increasingly realized through software. This paper presents a power-efficient version of a widely used fixed priority scheduling method. The method yields a power reduction by exploiting slack times, both those inherent in the system schedule and those arising from variations of execution times. The proposed run-time mechanism is simple enough to be implemented in most kernels. Experimental results show that the proposed scheduling method obtains a significant power reduction across several kinds of applications.

1 Introduction

Recently, power consumption has been a critical design constraint in the design of digital systems due to widely used portable systems such as cellular phones and PDAs, which require low power consumption with high speed and complex functionality. The design of such systems often involves reprogrammable processors such as microprocessors, microcontrollers, and DSPs in the form of off-the-shelf components or cores. Furthermore, an increasing amount of system functionality tends to be realized through software, which is leveraged by the high performance of modern processors. As a consequence, reduction of the power consumption of processors is important for the power-efficient design of such systems.

Broadly, there are two kinds of methods to reduce power consumption of processors. The first is to bring a processor into a power-down mode, where only certain parts of the processor such as the clock generation and timer circuits are kept running when the processor is in an idle state. Most power-down modes have a trade-off between the amount of power saving and the latency incurred during mode change. Therefore, for an application where latency cannot be tolerated, such as for a real-time system, the applicability of power-down may be restricted.

Another method is to dynamically change the speed of a processor by varying the clock frequency along with the supply voltage when the required performance on the processor is lower than the maximum performance. A significant power reduction can be obtained by this method because the dynamic power of a CMOS circuit, which is a dominant source of power dissipation in a digital CMOS circuit, is quadratically dependent on the supply voltage. Since there is a delay overhead along with an area requirement on the processor and a power overhead in dynamically changing the speed of the processor, great care must be taken when employing this method in the design of a real-time system.

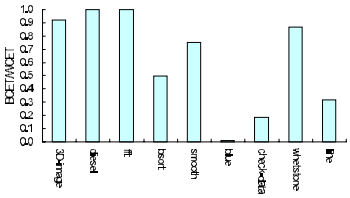

Figure 1: The ratio between BCET and WCET for a number of applications.

In this paper, we investigate power-conscious scheduling of hard real-time systems. In particular, we focus our attention on fixed priority scheduling and propose its power-efficient version, which we call *Low Power Fixed Priority Scheduling* (LPFPS). Our approach is built upon two observations regarding the behavior of a real-time system. The first is that the dynamics of a hard real-time system vary from time to time. Specifically, we need a handful of timing parameters for each of the tasks making up the system, to analyze the system for its schedulability [1, 2, 3, 4]. One of those parameters is the *worst-case execution time* (WCET), which can be obtained through static analysis [5, 6, 7], profiling, or direct measurement. However, during operation of the system, the execution time of each task frequently deviates from its WCET, sometimes by a large amount. This is because the possibility of a task running at its WCET is usually very low, even though a real-time system designer must use WCET to guarantee the temporal requirements. As examples of this variation in execution time, Figure 1 shows the ratio between the best-case execution time (BCET) and WCET obtained from [8] for a number of applications.

The second observation is that, in fixed priority scheduling, there are usually some idle time intervals even when the system just meets the schedulability and tasks run at their WCETs [1, 2, 3]. The actual number and length of these idle time intervals increase when some of the tasks run faster than their WCET, which was our first observation.

In LPFPS, we exploit both execution time variation and idle time intervals to obtain a power saving for a processor while ensuring that all tasks adhere to their timing constraints. To obtain the maximum power saving, we dynamically vary the speed of the processor whenever possible, and bring the processor to a power-down mode when it is predicted to be idle for a sufficiently long interval. Specifically, if there is only one task eligible for execution and its required execution time is less than its allowable time frame, the clock frequency of the processor along with the supply voltage is lowered. If it is detected that there is no task eligible for execution until the next arrival of a task, the processor enters power-down mode. Both these mechanisms are made possible by a slight modification of the conventional fixed priority scheduler.

The remainder of the paper is organized as follows. In the next section, we briefly review related work, which focuses on the re-

DAC 99, New Orleans, Louisiana

duction of power consumption of processors, and then discuss the motivation of LPFPS. In section 3, we introduce LPFPS and explain the advantages of the proposed scheme. In section 4, we present experimental results for a number of real-time system examples, and draw conclusions in section 5.

2 Related Work and Motivation

2.1 Power Down Modes

In most embedded systems, a processor often waits for some events from its environment, wasting its power. To reduce the waste, modern processors are often equipped with various levels of power modes. In the case of the PowerPC 603 processor [9], there are four power modes, which can be selected by setting the appropriate control bits in a register. Each mode is associated with a level of power saving and delay overhead. For example, in *sleep mode*, where only the PLL and clock are kept running, power consumption drops to 5% of full power mode with about 10 clock cycles delay to return to full power mode.

In the conventional approach employed in most portable computers, a processor enters power-down mode after it stays in an idle state for a predefined time interval. Since the processor still wastes its energy while in the idle state, this approach fails to obtain a large reduction in energy when the idle interval occurs intermittently and its length is short. In [10, 11], the length of the next idle period is predicted based on a history of processor usage. The predicted value becomes the metric to determine whether it is beneficial to enter power-down modes or not. This method focuses on event-driven applications such as user-interfaces because latency, which arises when the predicted value does not match the actual value, can be tolerated. However, we need an exact value instead of a predicted value for the next idle period when we are to apply the power-down modes in a hard real-time system, which is possible in the LPFPS.

2.2 Scheduling on a Variable Speed Processor

A scheduling method to reduce power consumption by adjusting the clock speed along with the supply voltage of a processor was first proposed in [12] and was later extended in [13]. The basic method is that short-term processor usage is predicted from a history of processor utilization. From the predicted value, the speed of the processor is set to the appropriate value. Because latency exists when the prediction fails, these methods cannot be applied to real-time systems.

Static scheduling methods for real-time systems were proposed in [14, 15, 16]. The underlying model of their approaches is a set of tasks with a single period. When periods of tasks are different from each other, which is the conventional model employed in real-time system design, we can transform a problem by taking the LCM (Least Common Multiple) of tasks' periods as a single period and treating each instance of the same task occurring within the LCM as a different task. This can cause a practical problem because we require excessively large memory space to save a statically computed schedule, whereas the size of memory is one of the design constraints in a typical embedded system. Furthermore, LCM becomes excessively large when periods of tasks are mutually prime. Another problem is that a schedule is computed based on the assumption that a fixed amount of execution time is required for each task. As a result, the full potential of power saving cannot be obtained when variations of execution time exist.

A dynamic scheduling method, called Average Rate Heuristic (AVR), was also proposed in [14] with the same model as in the static version. Associated with each task is its *average-rate requirement*, which is defined by dividing its required number of cycles by its time frame (deadline − arrival time). At any time t, the AVR sets the speed of a processor to the sum of average-rate requirements of tasks whose time frame includes t. Among available tasks, AVR resorts to the earliest deadline policy [1] to choose a task. Because average-rate requirements are computed statically with fixed numbers of execution cycles, the same problem occurs when variations of execution time exist.

Table 1: An example task set

	T_i	D_i	C_i	Priority
τ_1	50	50	10	1
τ_2	80	80	20	2
τ_3	100	100	40	3

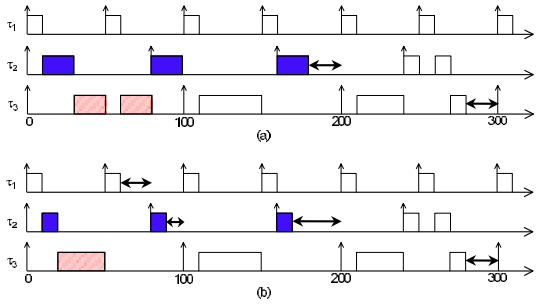

Figure 2: A schedule for the example task set. (a) When tasks always run at their WCET. (b) When the execution times of the first three instances of τ_2 and the first instance of τ_3 are smaller than their WCETs, respectively.

2.3 Motivation

Consider the three tasks given in Table 1. Rate monotonic priority assignment is a natural choice because periods (T_i) are equal to deadlines (D_i). Priorities are assigned in row order as shown in the fifth column of the table[1]. Assume all tasks are released simultaneously at time 0. A typical schedule, which assumes that tasks run at their WCETs (C_i), is shown in Figure 2(a). Note that this system just meets its schedulability. For example, if τ_2 were to take a little longer to complete, τ_3 would miss its deadline at time 100. Even though the system is tightly constructed, there are still some idle time intervals, as can be seen in the figure. At time 160 in Figure 2(a), when the request for τ_2 arrives, the run-time scheduler knows that there will be no requests for any tasks until time 200, which is the time when requests for τ_1 and τ_3 will arrive. This knowledge can be derived by examining run-time queues. We will elaborate on the details in the next section. As a consequence, we can save power by reducing the speed of the processor by lowering the clock frequency then lowering the supply voltage. When some task instances are completed earlier than their WCET, we have more chances to apply the same mechanism. For the example of Figure 2(b), we can slow down the processor at time 50 because the first instances of τ_2 and τ_3 complete their execution before the second request for τ_1 arrives. Because the execution time of each task frequently deviates from its WCET during the operation of the system, we have many chances to slow down the processor as shown in the figure.

The second possibility for power saving occurs when there are no tasks eligible for execution. At time 80 in Figure 2(a), we should

[1] We assume that a priority is higher when the value of the priority is lower, a convention usually adopted in real-time scheduling.

maintain the processor at its full speed because there will be requests for τ_1 and τ_3 at time 100, which is the same time when τ_2 will complete its execution at its WCET. If τ_2 completes its execution earlier at time 90 as shown in Figure 2(b), the processor can enter the power-down mode with timer set to the time 100. This is again possible because the run-time scheduler has exact knowledge that the processor will be idle until time 100. Another chance for applying power-down modes occurs in a slightly different situation. At time 160 in Figure 2(a), we can reduce the speed of the processor by half[2] because the available time for τ_2 is twice as large as its WCET. Even with the lowered speed, if τ_2 completes its execution earlier, meaning that it runs faster than its WCET, the processor can enter the power-down mode.

3 Low Power Fixed Priority Scheduling

3.1 Fixed Priority Preemptive Scheduling

In a typical real-time system, there are many periodic tasks that share hardware resources. To ensure that each task satisfies its timing constraints, the execution of tasks should be coordinated in a controlled manner. This is often done via fixed priority scheduling. Fixed priority scheduling has several advantages over other scheduling schemes. It is quite simple to implement in most kernels. Also, many analytical methods are available to determine whether the system is schedulable. Rate monotonic scheduling (RMS) [1] is the first scheduling scheme that falls into this category. It assigns a higher priority to a task with a shorter period or with a higher execution rate. It is proved to be optimal in the sense that if a given task set fails to be scheduled by RMS, it cannot be scheduled by any fixed priority scheduling. Although RMS is constrained by a set of assumptions [1], recent research has relaxed these constraints in several ways. For example, deadline monotonic priority assignment [4] can be used when the deadlines are different from the periods. Earliest deadline first (EDF) scheduling [1], which is an optimal dynamic priority scheduling, has an apparent dominance over RMS because it can schedule a task set if and only if the processor utilization is lower than or equal to 1, meaning that a schedule with zero slack time is possible. However, RMS by itself is of great practical importance [2].

Once the priorities are assigned to each task, the scheduler ensures that higher priority tasks always take the processor over lower priority ones. This is maintained by preempting lower priority tasks when higher priority ones request the processor, which is called a context switch.

The basic mechanism of the scheduler in the kernel proposed in this paper is based on the implementation model in [17, 18]. The scheduler maintains two queues, one called *run queue* and the other called *delay queue*. The run queue holds tasks that are waiting to run and the tasks in the queue are ordered by priority. The task that is running on the processor is called the *active task*. The delay queue holds tasks that have already run in their period and are waiting for their next period to start again. They are ordered by the time their release is due. When the scheduler is invoked, it searches the delay queue to see if any tasks should be moved to the run queue. If some of the tasks in the delay queue are moved to the run queue, the scheduler compares the active task to the head of the run queue. If the priority of the active task is lower, a context switch occurs. The process is illustrated in the following example using the task set in Table 1.

Example 1 *At time 0, when the requests for all tasks arrive, tasks are put in the run queue in priority order. Because τ_1 has the highest priority, it becomes the active task and immediately starts execution. Figure 3(a) shows the status of the queues. At time 50, when the second request for τ_1 arrives, τ_3 is preempted because it has a lower priority than τ_1 (Figure 2(a)). It goes to the run queue and τ_1 starts execution as the active task. Figure 3(b) shows the status of the queues.*

[2] At this moment, we ignore the delay to vary the speed of the processor for simplicity.

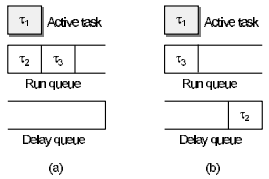

Figure 3: The status of queues for the task set example (a) at time 0 and (b) at time 50.

3.2 Overview

As described in the previous subsection, the fixed priority preemptive scheduler in the kernel can be implemented easily using run-time queues. Because most information about the tasks is available through queues and LPFPS depends on this information, the scheduler for LPFPS can be implemented with a slight modification of the conventional scheduler.

Figure 4 shows pseudo code for the LPFPS scheduler. The code between L5 and L11 conforms to the behavior of the conventional scheduler explained in the previous subsection. LPFPS works when the run queue is empty (L12). This is further divided into two cases: one when all tasks have completed their executions in each of their periods and are waiting for their next arrival times while residing in the delay queue (L13) and the other when all tasks *except* the active task have completed their execution (L16). In the first case, we can bring the processor into a power-down mode because there are no tasks that need it. Furthermore, we know how long the processor will be idle because the task at the head of the delay queue is the first one that will require the processor (recall that the delay queue is ordered by the tasks' release times). This is the key ingredient of LPFPS. Thus, we set a timer to expire at the next release time of the head of the delay queue and then put the processor into power-down mode. Because, there is a delay overhead to wake up from power-down mode, the timer actually should be set to expire earlier by that amount of delay (L14).

In the second case, we can control the speed of the processor because there is just one task (the active task) to execute and the processor will be available solely for that task until the release time of the task at the head of the delay queue. Note that instead of changing the speed of the processor to adopt to the computational requirements imposed on the processor, we can keep the proceesor at the maximum speed and then bring it into a power-down mode. However, it can be shown that the former method obtains a more power saving because the dynamic power of a CMOS circuit is quadratically dependent on the supply voltage. The amount of time that will be needed by the active task equals its WCET less its already executed time[3]. Note that we assume that the execution of the whole task takes its WCET because at the time of scheduling we have no information whether it will take less than WCET or not. When the active task completes its execution, the processor should return to the full speed to prepare for the next arrival of tasks (L1 through L4). This involves a delay for raising the supply voltage and subsequently the clock frequency. Thus, the active task actually should complete its execution ahead by an amount equal to this delay. Considering all these factors, we obtain the ratio of the

[3] In preemptive scheduling, a task is preempted when a request for a task with higher priority arrives during its execution (L8). When this occurs, we get the executed time of the task from the timer (L9), which is supplied by most processors used in real-time systems.

```
L1:   if current_frequency < maximum_frequency then
L2:       increase the clock frequency and the supply voltage
          to the maximum value;
L3:       exit;
L4:   end if
L5:   while delay_queue.head.release_time ≤ current_time do
L6:       move delay_queue.head to the run_queue;
L7:   end do
L8:   if run_queue.head.priority > active_task.priority then
L9:       set the active_task.executed_time;
L10:      context switch;
L11:  end if
L12:  if run_queue is empty then
L13:      if active_task is null then
L14:          set timer to (delay_queue.head.release_time − wakeup_delay);
L15:          enter power down mode;
L16:      else
L17:          speed_ratio = Compute_speed_ratio();
L18:          find a minimum allowable
              clock frequency ≥ speed_ratio · max_frequency;
L19:          adjust the clock frequency along with the supply voltage;
L20:      end if
L21:  end if
```

Figure 4: Pseudo code of the LPFPS scheduler.

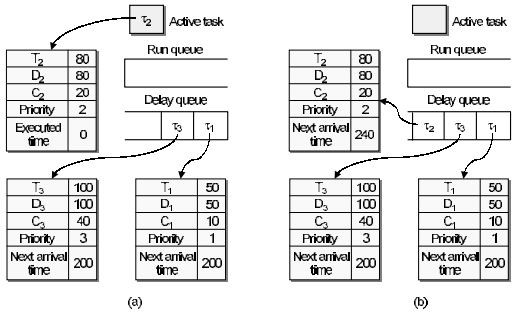

Figure 5: The status of queues and the information associated with each task (a) at time 160 and (b) at time 180.

processor speed needed for the active task to the full speed (L17), which we will elaborate in detail in the next subsection. From the computed ratio, we find an appropriate clock frequency (L18). In practice, only discrete levels of frequency are available, and among them we should select a frequency larger than or equal to the computed one to guarantee the timing constraints. All these processes are illustrated in the following example with the same task set as in Example 1.

Example 2 *At time 160 in Figure 2(a), when a request for τ_2 arrives, the status of queues and the information associated with each task are as shown in Figure 5(a). For simplicity of illustration, assume that the delay required to wake up from the power-down mode and that required to change the speed of a processor are all 0. Because the run queue is empty with the active task of τ_2, the scheduler computes the desired ratio of speed that yields $\frac{20-0}{200-160} = 0.5$ (see L17 of Figure 4). Thus, we can slow down the processor by half. Now, assume that the instance of τ_2 started at time 160 executes at the lowered speed, but completes its execution at time 180 instead of 200, meaning that it executes in half its WCET. At this time, the status of queues becomes that of Figure 5(b). Because all tasks reside in the delay queue, the scheduler brings the processor into a power-down mode (see L14 and L15 of Figure 4) with the timer set to the next arrival time of τ_1 (200).*

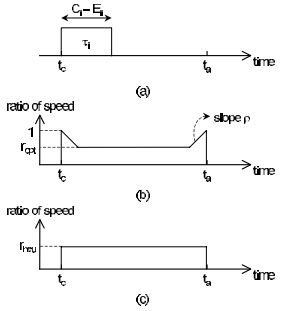

Figure 6: Computation of the speed ratio. (a) An instance when the processor's speed can be changed, (b) Optimal solution, and (c) Heuristic solution.

3.3 Computation of the Ratio of Processor's Speed

Because it takes time to change the clock frequency and the supply voltage, we should take this delay into account when computing the processor's speed ratio. We present two methods to compute the ratio, an optimal but complex solution and a heuristic but simple solution, and show that the latter one is always safe and is accurate enough for many practical situations. Figure 6(a) shows an instance when we can change the processor's speed, that is, the active task alone is eligible for execution. Before we explain the solutions in detail, we introduce the notations we use in the solutions.

- The active task is denoted by τ_i. C_i is its WCET and E_i denotes the time for which it has already executed.
- t_a is the next arrival time of the task at the head of the delay queue and t_c is the current time.
- ρ is the rate of changing the speed ratio of the processor. For example, if the clock frequency can be raised from 30 MHz to 100 MHz (full speed) in 10 μs (including the delay to raise the supply voltage), $\rho = 0.07/\mu$s.

The optimal (or exact) desired ratio of speeds, denoted by r_{opt}, can be computed with the help of Figure 6(b) and with the knowledge that the processor can still execute operations while its speed is being changed. Because the area under the curve should be equal to the required execution time, $C_i - E_i$, we have

$$(t_a - t_c) r_{opt} + \frac{(1 - r_{opt})^2}{\rho} = C_i - E_i. \tag{1}$$

Solving for r_{opt} gives

$$r_{opt} = \frac{-\rho(t_a - t_c) + 2 + \sqrt{\rho^2 (t_a - t_c)^2 - 4\rho(t_a - t_c - C_i + E_i)}}{2}. \tag{2}$$

The equation (2) gives an accurate ratio provided that the speed is changed linearly with time. However, it has some practical problems. It is computationally expensive (compared to the execution time of the conventional scheduler, see L5 through L11 of Figure 4), which adds a burden to the run-time scheduler. Note that the overhead of the scheduler should be kept as small as possible so as not to violate the schedulability of the system [17, 18]. Furthermore, an increase in the execution time of the scheduler translates into increased power consumption.

To overcome the problems, we resort to a straightforward heuristic solution, given by

$$r_{heu} = \frac{C_i - E_i}{t_a - t_c}, \tag{3}$$

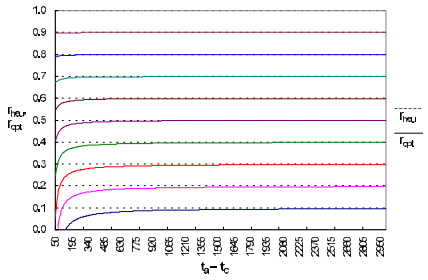

Figure 7: Optimal ratio versus heuristic ratio over time intervals.

which is simply the solution built upon the assumption that the delay is negligible (see Figure 6(c)). To use r_{heu} in practice, it should be guaranteed that it has a *safeness* property in the sense that r_{heu} is always larger than or equal to r_{opt}, so that the active task (τ_i) can complete its execution before t_a. It should also have *accuracy* in that it should be close to r_{opt} in practical situations[4]. The safeness is guaranteed by the following theorem. The proof can be found in the Appendix.

Theorem 1 *r_{heu} is always larger than or equal to r_{opt} provided that $t_a > t_c$ and $t_a - t_c > C_i - E_i$.*

We compute r_{opt} with $\rho = 0.07/\mu s$ while we vary $t_a - t_c$ from 50 μs to 3000 μs for each of r_{heu} from 0.1 to 0.9. As can be seen in Figure 7, r_{heu} closely matches r_{opt} except for small values of $t_a - t_c$ and for low r_{heu}. Thus, we can obtain a sufficient power reduction while guaranteeing real-time constraints using equation (3) instead of equation (2) in a broad range of situations.

4 Experimental Results

To evaluate the LPFPS, we simulate several examples and compare the average power consumed with LPFPS against that consumed with fixed priority scheduling (FPS). In FPS, we assume that the processor executes a busy wait loop, which consists of NOP instructions, when it is not being occupied by any tasks. The average power consumed by a NOP instruction is assumed to be 20% of that consumed by a typical instruction [19]. The delay overhead to vary the clock frequency and the supply voltage is assumed to follow the model in [20], where the clock is generated by a ring oscillator driven by the operating voltage resulting in the worst-case delay of 10 μs. The maximum clock frequency and the supply voltage of the processor, which is based on the ARM8 microprocessor core, is 100 MHz and 3.3 V, respectively. The clock frequency can be varied from 100 MHz down to 8 MHz with a step size of 1 MHz. We assume that the average power consumed by the processor when it is in power-down mode is 5% of the full power mode and that it takes 10 clock cycles to return from the power-down mode to the full power mode [19]. We make all these assumptions in order to reflect implementation issues thereby enabling a fair comparison between FPS and LPFPS.

We collected four applications for experiments: an Avionics task set [21], an INS (Inertial Navigation System) [18], a flight control system [22], and a CNC (Computerized Numerical Control) machine controller [23]. The first three examples are mission critical applications and the last one is a digital controller for a CNC machine, which is an automatic machining tool that is used to produce user-defined workpieces. All the examples are summarized in Table 2 where we show the number of tasks in each application and the range of WCETs in the unit of μs. Note that the worst-case

[4]Safeness is a mandatory condition in a hard-real time system whereas accuracy is not. We simply obtain a smaller power reduction with a larger r_{heu}.

Table 2: Task sets for experiments

Applications	# tasks	Range of WCETs (μs)
Avionics	17	1,000 ∼ 9,000
INS	6	1,180 ∼ 100,280
Flight control	6	10,000 ∼ 60,000
CNC	8	35 ∼ 720

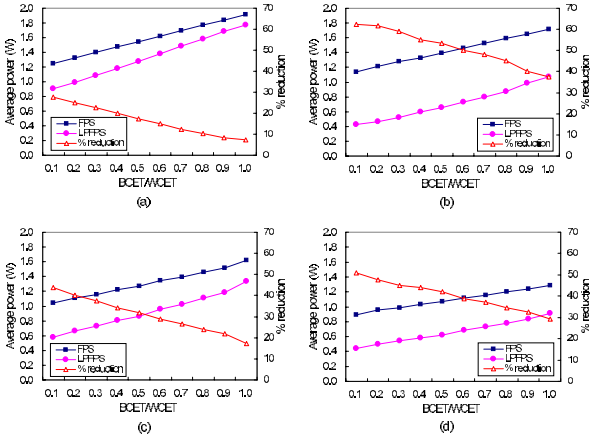

Figure 8: Simulation results of (a) Avionics, (b) INS, (c) Flight control, and (d) CNC.

delay to vary the clock frequency and the supply voltage (10 μs) is negligible compared to the WCETs except for CNC. We use the heuristic solution (equation (3)) to compute the ratio of processor's speed. Because the statistics of the actual execution times of instances of the tasks comprising each application are not available, we assume that the execution time of each instance of a task is drawn from a random Gaussian distribution with mean, denoted by m, and standard deviation, denoted by σ, given by[5]

$$m = \frac{\text{BCET} + \text{WCET}}{2}, \tag{4}$$

$$\sigma = \frac{\text{WCET} - \text{BCET}}{6}. \tag{5}$$

Figure 8 shows the simulation results when we vary the BCET from 10% to 100% of the WCET for each application. Even when the BCET equals the WCET, which is the case when tasks always execute in their WCET, LPFPS obtains a higher power reduction than FPS. This is the result of dynamically varying the clock frequency and the supply voltage when the active task alone is eligible for execution. We can observe from the figure that the power gain increases as the BCET gets smaller. This matches the motivation of this paper illustrated in section 1 and 2: the chance both for dynamically varying the clock frequency and the supply voltage and for bringing the processor into a power-down mode increases as the variation of execution times increases.

Among the applications, the LPFPS obtains the most power gain (up to 62% power reduction) for INS, as shown in Figure 8. This is another interesting fact observed with LPFPS. For FPS, the average power consumption is proportional to processor utilization, $U = \sum_i \frac{C_i}{T_i}$. However, it is not true for LPFPS. This is evident from

[5]In a random Gaussian distribution, the probability that a random variable x takes on a value in the interval $[m-3\sigma, m+3\sigma]$ is approximately 99.7%. Thus, if we set WCET to be equal to $m+3\sigma$, almost all generated values fall between BCET and WCET. Let $m+3\sigma = \text{WCET}$ and solving for σ with the help of equation (4), we get equation (5). After the generation of execution times, we apply clamping operation so that the generated value does not exceed WCET.

Figure 8 where INS with the second largest processor utilization consumes relatively low average power when LPFPS is used. Investigation of the application reveals the reason. In INS, the processor utilization (0.736) is occupied mostly by one task (0.472) and the remaining utilization is spread over other tasks (in the range between 0.02 and 0.1). Furthermore, the period of that task (2500) is the shortest and much shorter than those of other tasks (in the range between 40000 and 1250000), meaning that it has the highest rate and thus has the highest priority under rate monotonic priority assignment. Therefore, in INS, the run queue is empty for most of the time and the processor has many chances to run at lowered clock frequency and supply voltage for a heavily loaded task thereby obtaining a larger power gain with LPFPS than other applications, where the utilization is more equally distributed.

5 Conclusion

In this paper, we propose a power-efficient version of fixed priority scheduling, which is widely used in hard real-time system design. Our method obtains a power reduction for a processor by exploiting the slack times inherent in the system and those arising from variations of execution times of task instances. We present a run-time mechanism to use these slack times efficiently for power reduction for a processor that supports a power-down mode and can change the clock frequency and the supply voltage dynamically. For computation of the ratio of the processor's speed, two solutions are proposed and compared. The heuristic solution, which is simple and amenable to implementation issues, is shown to be always safe and accurate enough to be used in a broad range of applications. Experimental results show that the proposed method obtains a power reduction across several applications.

The heuristic solution to compute the processor's speed ratio may fail to obtain the full potential of power saving when the timing parameters associated with the system are comparable to the delay exhibited when the processor's speed is changed (see Figure 7), though it still guarantees safeness. In this case, we can use the optimal solution at the cost of increased execution time and power consumption of the scheduler; this approach needs a trade-off analysis, which is included in our future work.

Appendix

Here we present the proof to Theorem 1. Let $C_i - E_i = R_i$ and $t_a - t_c = t_I$. For $r_{hue} \geq r_{opt}$, we need to prove

$$\frac{R_i}{t_I} \geq \frac{-\rho t_I + 2 + \sqrt{\rho^2 t_I^2 - 4\rho(t_I - R_i)}}{2}, \tag{6}$$

provided that $r_{opt} > 0$. It follows that

$$\frac{R_i}{t_I} + \frac{\rho t_I}{2} - 1 \geq \frac{\sqrt{\rho^2 t_I^2 - 4\rho(t_I - R_i)}}{2}, \tag{7}$$

and squaring both sides gives

$$\frac{(R_i - t_I)^2}{t_I^2} \geq 0, \tag{8}$$

which is true. □

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.

[2] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proc. IEEE Real-Time Systems Symposium*, pp. 166–171, Dec. 1989.

[3] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer J.*, vol. 29, pp. 390–395, Oct. 1986.

[4] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard real-time scheduling: The deadline-monotonic approach," in *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pp. 133–137, May 1991.

[5] C. Park and A. C. Shaw, "Experiments with a program timing tool based on source-level timing schema," *IEEE Computer*, pp. 48–57, May 1991.

[6] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim, "An accurate worst case timing analysis for RISC processors," in *Proc. IEEE Real-Time Systems Symposium*, pp. 97–108, Dec. 1994.

[7] Y. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," in *Proc. Int'l Conf. on Computer Aided Design*, pp. 380–387, Nov. 1995.

[8] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," in *Proc. Int'l Conf. on Computer Aided Design*, pp. 598–604, Nov. 1997.

[9] S. Gary, "PowerPC: A microprocessor for portable computers," *IEEE Design & Test of Computers*, pp. 14–23, Dec. 1994.

[10] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Trans. on VLSI Systems*, vol. 4, pp. 42–55, Mar. 1996.

[11] C. Hwang and A. Wu, "A predictive system shutdown method for energy saving of event-driven computation," in *Proc. Int'l Conf. on Computer Aided Design*, pp. 28–32, Nov. 1997.

[12] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pp. 13–23, 1994.

[13] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithms for dynamic speed-setting of a low-power CPU," in *Proc. ACM Int'l Conf. on Mobile Computing and Networking*, pp. 13–25, Nov. 1995.

[14] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Proc. IEEE Annual Foundations of Computer Science*, pp. 374–382, 1995.

[15] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable voltage core-based systems," in *Proc. Design Automat. Conf.*, pp. 176–181, June 1998.

[16] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proc. Int'l Symposium on Low Power Electronics and Design*, pp. 197–202, Aug. 1998.

[17] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Trans. on Software Eng.*, vol. 19, pp. 920–934, Sept. 1993.

[18] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Trans. on Software Eng.*, vol. 21, pp. 475–480, May 1995.

[19] T. Burd and R. Brodersen, "Processor design for portable systems," *Journal of VLSI Signal Processing*, vol. 13, pp. 203–222, Aug. 1996.

[20] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int'l Symposium on Low Power Electronics and Design*, pp. 76–81, Aug. 1998.

[21] C. Locke, D. Vogel, and T. Mesler, "Building a predictable avionics platform in Ada: a case study," in *Proc. IEEE Real-Time Systems Symposium*, Dec. 1991.

[22] J. Liu, J. Redondo, Z. Deng, T. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. Shih, "PERTS: A prototyping environment for real-time systems," Tech. Rep. UIUCDCS-R-93-1802, University of Illinois, 1993.

[23] N. Kim, M. Ryu, S. Hong, M. Saksena, C. Choi, and H. Shin, "Visual assessment of a real-time system design: a case study on a CNC controller," in *Proc. IEEE Real-Time Systems Symposium*, Dec. 1996.