Chapter 7

POWER MANAGEMENT POINTS IN POWER-AWARE REAL-TIME SYSTEMS

Rami Melhem, Nevine AbouGhazaleh, Hakan Aydin, Daniel Mossé
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260[*]
{melhem, mosse}@cs.pitt.edu

Abstract Managing power consumption while simultaneously delivering acceptable levels of performance is becoming a critical issue in several application domains such as wireless computing. We integrate compiler-assisted techniques with power-aware operating system services and present scheduling techniques to reduce energy consumption of applications that have deadlines. We show by simulation that our dynamic power management schemes dramatically decrease energy consumption.

Keywords: Processor Power, Dynamic Speed Setting, Speculation, Voltage Scaling

1. Introduction

As the remarkable advances in VLSI and communication technologies have culminated in a proliferation of mobile, embedded and wireless computers in the last decade, system designers are faced with a relatively new and difficult resource management problem. Most of these devices usually have to rely on *battery* power, which is usually rather scarce. Moreover, many popular and emerging applications designed for these platforms, such as wireless communication, and image/audio/video processing, tend to consume considerably higher power than more traditional tasks. Some recent studies even advocate the replacement of CPU-centric operating system design view by the energy-centric view [37]. In short, the *power bottleneck* has to be addressed in an

[*]This work has been partially supported by the Defense Advanced Research Projects Agency under contract F33615-00-C-1736

efficient way to guarantee the functionality in the upcoming *pervasive computing* era.

The Engineering and Computer Science communities at large confronted the low power system design problems with a multi-dimensional effort [12, 28]. Hardware and software manufacturers have agreed to introduce standards such as the ACPI (Advanced Configuration and Power Interface) [16] for power management of laptop computers that allows several modes of operation, such as predictive system shutdown [34]. Such on-going low-power research has important implications for real-time systems design, simply because most of the applications running on power-limited systems inherently impose temporal constraints on the response time (such as real-time communication in satellites).

An increasingly popular technique for saving power, *Dynamic Voltage Scaling* (DVS) [38], is based on exploiting the convex (usually quadratic) relation between the supply voltage and the CPU power consumption. In addition, it has been long recognized that the CPU clock frequency (hence, the speed) should be reduced in parallel with the supply voltage [8]. In this case, it is possible to obtain striking (quadratic) energy savings at the expense of roughly linearly increased response time. One aspect that needs to be carefully taken into consideration is the energy and delay overhead associated with speed/voltage changes. Some studies are optimistic about the overhead imposed by DVS schemes [30]; we examine this issue in more detail in Section 4.

The DVS framework aims at stretching out task executions through speed and voltage reduction. For real-time systems, the proposed DVS schemes focus on minimizing energy consumption in the system, while still meeting the deadlines. The extensive literature on traditional real-time scheduling theory [22, 7] deals with settings where the CPU speed is constant, and hence can not be directly applied.

The principle of slowing down the processor can and should be applied in multiple dimensions. The natural starting point is the *static* DVS dimension, where the aim is to compute the optimal speed assignments for a given real-time task set and a (worst-case) workload.

In one of the earliest studies in this line, Yao et. al [40] provided an optimal static off-line scheduling algorithm to minimize the total energy consumption while meeting all the deadlines, assuming independent aperiodic tasks with release times and timing constraints. The algorithm has $O(n \log^2 n)$ time complexity for a system of n tasks. Heuristics for on-line scheduling of aperiodic tasks while not hurting the feasibility of off-line periodic requests are proposed in [14], which also suggested assigning a uniform speed value to all periodic tasks using the *total task utilization* as a basis. Non-preemptive power aware scheduling is investigated in [13]. Concentrating on a periodic task set with identical periods, the effects of having an upper bound on the voltage change

rate are examined in [15]. The authors show that the problem is intractable even with a linear change rate and propose a heuristic to tackle the problem. The static solution for the general periodic model where tasks have potentially different power consumption characteristics is provided in [4]. Aydin et. al recently showed [5] the equivalence of the *static* dynamic voltage scaling problem to the reward-based scheduling problem [3].

Designing a real-time system with worst-case workload assumption in mind is common and often necessary. However, the *actual* workload may be much lower than the worst-case assumption in many real-time applications. Most of the scheduling schemes presented in the above studies, while using exclusively worst-case execution time (WCET) to guarantee the timeliness of the system, lack the ability to dynamically take advantage of unused computation time. In fact, applications usually exhibit a large variation in *actual* execution times; for example, [9] reports that the ratio of the worst-case execution time to the best-case execution time can be as high as 10 in typical applications. Consequently, *dynamically monitoring and reclaiming the 'unused' computation* can be (and, as we show below, is in fact) a powerful approach to obtain considerable power savings and to minimize the effects of conservative predictions of the *actual* execution time by the WCET information. In this line of research, the aim is to dynamically reduce the CPU speed of running task(s) by taking into account the early completions in the history of task executions. The main problem, naturally, is to determine the speed reduction amount that would not compromise any timing constraints, in addition to choosing the tasks that will be executed at the low speed.

One technique for dynamic reclaiming relies on slowing down the processor whenever there is only a single task eligible for execution [33], where a set of periodic tasks scheduled by Rate Monotonic Scheduling [20] is considered. A more general dynamic reclaiming approach, based on comparing the worst-case execution schedule to the actual schedule is proposed in [17]. In that study, a detailed analysis is provided for frame-based tasks; the extension to general Earliest Deadline First (EDF) scheduling of periodic tasks is sketched. One assumption of that work is that there are (only) two discrete speed levels. However, systems which are able to operate on a (more or less) continuous voltage spectrum are rapidly becoming a reality thanks to advances in power-supply electronics and CPU design [10, 27]. For example, the Crusoe processor is able to dynamically adjust clock frequency in 33 MHz steps [36].

Despite the gains due to *static* and *dynamic reclaiming* schemes, there is still room for additional savings provided that we have access to the *statistical* workload information; in this chapter, we present also aggressive schemes where we *anticipate* the early completions of *future* executions and *speculatively* reduce the CPU speed. This approach immediately raises two intertwined questions, namely, (a) the *level* of aggressiveness that justifies specula-

tive speed reductions under a given probability distribution of actual workload; and (b) the issue of guaranteeing the timing constraints.

Note that both dynamic reclaiming and aggressive scheduling techniques can be adopted at the *task level* **and** at the *system level*. In the former, user- or compiler-inserted Power Management Points (PMPs) allow intra-task monitoring of the execution and controlling speed of a given task to improve the energy savings [1, 32]. In the latter, the operating system invokes the PMPs at context switch times, taking advantage of the global knowledge (i.e., system-wide workload). Finally, it is worth mentioning that a whole new line of research that tolerates (and tries to minimize) deadline misses for the sake of energy savings has recently emerged [23, 18, 19, 31, 11]. These *soft* real-time scheduling techniques also make use of the *statistical* workload information while determining the CPU speed assignments.

In this chapter, we summarize the results of our multi-layered research effort in power-aware scheduling for real-time systems. Our solution is based on a three-dimensional approach that can be applied at the task level or at the system level, while taking into account energy and time overheads. Hence, we present:

1 A *static* (off-line) solution to compute the optimal speed, assuming *worst-case workload*,

2 A *dynamic* (on-line) speed adjustment mechanism based on the *actual workload*, used to reclaim unused time/energy, when executions fall short of their worst-case workload, and

3 An adaptive and *speculative* speed adjustment mechanism based on *statistical information about the workload*, used to anticipate and compensate probable early completions of future executions.

We emphasize once again that, in the context of real-time systems, all these components should be designed not to cause any deadlines to be missed even under the worst-case scenario: the aim is to **meet the timing constraints while simultaneously and dynamically reducing power consumption as much as possible**.

2. Real-time task and system models

Typical real-time research assumes that a task, τ_i, has a deadline, D_i, which is derived from the system's real-time requirements. If we assume that a task is ready at time 0, then D_i can be seen as the length of the time interval within which τ_i is allowed to execute. Given that variable voltage CPUs are available, the time to execute a program, P_i, depends on the processor speed. We will characterize a task (we will use task τ_i and program P_i to denote the same entity) by the worst case *number of CPU cycles*, C_i, needed to execute the program.

In order to simplify the analysis and to allow for the derivation of analytical formulas, we would like to assume that C_i is independent of the CPU speed for a given processor architecture. This assumption, however, does not hold if the speed of the memory system is independent of the speed of the CPU, since memory references will consume larger number of cycles when the processor speed is high, thus increasing the total number of cycles needed to execute the program. For this reason, we assume that C_i is the worst case number of CPU cycles needed to execute a program at the maximum processor speed.

We have conducted a number of simulation experiments using SimpleScalar 3.0 [6] (a micro architectural simulator) to determine the degree of pessimism in the definition of C_i. These experiments show that, with on-chip caches and low cache miss rates, C_i does not change substantially with the processor speed. For the Li, Perl, Go and Compress programs from the SPEC benchmarks [35], changing the processor speed from 700 MHz to 300 MHz changed the number of CPU cycles needed to execute the benchmarks by 0.01%, 1.2%, 1.9% and 0.6%, respectively. In all the experiments, the default SimpleScalar configurations for the L1 and L2 caches are used and no disk I/O is performed (typical assumption for real-time systems).

In this chapter, we normalize the units of C_i such that the maximum processor speed is 1. That is, if the maximum processor speed is s CPU cycles per second, then we define a *hypercycle* to consist of s CPU cycles and express C_i in terms of the number of hypercycles. The maximum processor speed is thus normalized to $S_{max} = 1$ hypercycles per second. We will simply refer to hypercycles by "cycles", and thus, at S_{max}, the time for executing C_i cycles is C_i seconds.

Modeling control flow

We consider a general form of program execution in which a program P_i is divided into n_i segments, $\tau_{i,(j)}, 1 \leq j \leq n_i$, where a segment is a loop, a procedure call or, in general, any subgraph of the control flow graph of P_i (see Figure 7.1). We assume that each segment $\tau_{i,(j)}$ executes a maximum of $C_{i,(j)}$ cycles. Each segment is represented by a circle. A segment that is composed of a loop is represented by a square and a number representing the maximum loop index. Note that a "segment flow graph" is a compact version of the control flow graph of the program, in which subgraphs are replaced by single nodes. Each execution of the program will follow a specific path from the start node to an end node.

For any given node, j, in the segment flow graph, let $\Pi\_wc_{i,(j)}$ denote the maximum number of cycles to complete the execution of the program, starting at the beginning of segment j. Clearly, $\Pi\_wc_{i,(j)}$, $1 \leq j \leq n_i$ can be computed recursively from

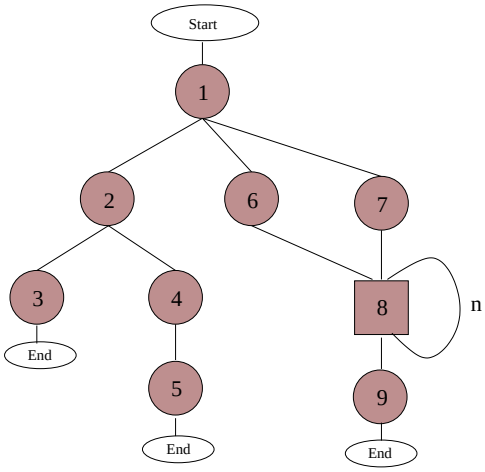

Figure 7.1. Schematic of a segment flow graph.

$$\Pi\_wc_{i,(j)} = C_{i,(j)} + \max_{k \in B(j)} \{\Pi\_wc_{i,(k)}\} \tag{7.1}$$

where $B(j)$ is the set of children of node j in the segment flow graph. If node 1 represents the first segment in the program, then $C_i = C_{i,(1)}$ represents the worst case execution of the entire program P_i.

Although knowing the worst case execution scenario of a task is essential to guarantee that a task meets its deadline, usually a program, P_i, executes for much less than its worst case estimate, C_i [9]. This is because the input data and system architecture (e.g., the amount of cache in the system) determine not only the actual number of cycles the segment executes (typically less than C_i), but also determines the execution path of P_i. If we denote by $C\_avg_{i,(j)}$ the average number of cycles consumed in the execution of segment j of P_i, then the average number of cycles to complete the execution of P_i starting at the beginning of segment j, denoted by $\Pi\_avg_{i,(j)}$, can be computed recursively from

$$\Pi\_avg_{i,(j)} = C\_avg_{i,(j)} + \sum_{k \in B(j)} Prob_k \cdot \Pi\_avg_{i,(k)} \tag{7.2}$$

where $Prob_k$ is the probability that execution proceeds from segment j to segment k. Clearly, execution should proceed from segment j to one of the segments in $B(j)$, and thus $\sum_{k \in B(j)} Prob_k = 1$.

Periodic task model

Typically, real-time systems execute periodic tasks, where each task τ_i has associated with it a period, T_i, which represents the interarrival time of consecutive instances of the task. We will assume that a task τ_i is ready for execution at the beginning of its period and should complete execution by the end of its period. *Frame-based systems* are special periodic real-time systems in which a sequence of frames is repeatedly executed, and all tasks in a frame have the same period, T, and the same initial phasing. In practice, many real-time systems are frame-based, since designing and verifying the correctness for such systems is much simpler than for more general real-time systems.

Given a set of tasks, $\{\tau_1, \ldots, \tau_N\}$, let $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$ be the total utilization of the task set under the maximum processor speed (recall that we normalized $S_{max} = 1$). The utilization U can also be seen as the load imposed on the system by the task set under consideration. It is well known that if $U \leq 1$, $T_i = D_i$, and EDF scheduling is used, then each instance of every task will meet its deadline [20]. In this chapter, we will assume that there are no precedence constraints among the N periodic tasks and that EDF is used to schedule these tasks.

Power consumption model

Variable-voltage CPUs can reduce power consumption *quadratically* or *cubically* at the expense of *linearly* increased delay (reduced speed) [15]. Thus, any effective DVS scheme should be able to vary the voltage fed to the system component and the frequency of the system clock. The power consumption of the processor under the speed S is given by $g(S)$, which is assumed to be a strictly increasing convex function, represented by a polynomial of at least the second degree [15]. If task τ_i occupies the processor during the time interval $[t_1, t_2]$, then the *energy* consumed during this interval is $\int_{t_1}^{t_2} g(S(t))dt$, which is equal to $g(S)(t_2 - t_1)$ if S is constant during the period $[t_1, t_2]$. Unless stated otherwise, we assume that the CPU speed can be changed continuously between a minimum speed, S_{min} (minimum supply voltage necessary to keep the system functional), and the maximum speed S_{max}. The idle processor power consumption, g_{idle}, is usually less than $g(S_{min})$ but larger than $g(0)$.

The importance of the speed management is derived from the convexity of the power function $g(S)$. Specifically, if τ_i is allotted Y_i time units to execute, then $S_i = \frac{C_i}{Y_i}$ is the speed that will execute C_i in exactly Y_i time units. Because the function g is convex, we have

$$g(S_i)Y_i \leq g(S')x + g(S'')(Y_i - x)$$

for any $S' \neq S''$ and x such that $S'x + S''(Y_i - x) = C_i$. This means that the total energy consumption is reduced when the processor speed is uniform during

the Y_i time units. Note that in case of a single task, $Y_i = D$, but when there are several tasks, their allocations typically fall short of their deadline.

3. PMPs: Power Management Points

In the context of power management through CPU speed adjustment, a *power management point*, PMP, is an abstraction that lets us reason about and influence power management at specific points in time. Typically, a PMP will have associated with it a piece of code that manages information about the execution of the tasks in the system and decides about changing the CPU speed. A PMP code can be part of the user's program (executes in user space) or can be part of the operating system (executes in kernel space). In either case, after making a decision to change the CPU speed, a PMP typically makes the appropriate system calls to change both the CPU clock frequency and CPU voltage. We distinguish between two types of PMPs.

- A *task-level PMP* is invoked during the execution of a given task, τ_i, and uses information only about τ_i to make decisions about τ_i's execution speed. Task-level PMPs may be inserted by the user or by the compiler in the program. For instance, the user or the compiler may insert a PMP at the beginning of each segment of the program.
- A *system-level PMP* takes a more global view of the system and uses information about all the tasks in the system to make speed adjustment decisions. For instance, after determining the next task to be dispatched, the scheduler in an operating system may execute a PMP to determine the execution speed of the next task.

To make a decision, a PMP uses *task profile information*, such as worst case executions, average case executions and timing or performance constraints, and *execution progress information* such as CPU time consumption and early termination of tasks. The task profile information can be precomputed and stored, while the execution progress information must be collected at run time with help from the hardware and/or the operating system. Different speed adjustment schemes can be designed based on how a PMP computes the slack existing in the system and on how to use this slack to carry out power (via CPU speed) management.

3.1 Static (off-line) power management

In static power management, a PMP computes the processor speed based on the assumption that each task, τ_i, will execute the maximum number of cycles, C_i. If executing C_i cycles at speed S_{max} does not consume the entire time allocated to τ_i, then it is possible to reduce the speed of executing τ_i thus reducing the consumed energy, while still meeting the timing constraints.

This power management scheme is called *static* because only worst-case task profile information and no execution progress information is used in speed adjustments.

For task-level power management, if D_i is the time allocated to execute τ_i, then the speed during the execution of τ_i can be safely set to $S_i = \frac{C_i}{D_i}$. Clearly, if $S_i > S_{max}$, then it is not possible to guarantee completion of τ_i's execution within the given time constraints. Also, if $S_i < S_{min}$, we should set $S_i = S_{min}$. A single PMP at the beginning of task's execution is needed to calculate and set the processor speed to S_i.

For system-level power management of N periodic tasks scheduled using EDF, the convexity of the power function, $g(S)$, implies that all deadlines can be met and that the total energy is minimized when the speed of the processor is the same for all the tasks. Hence, when executing a periodic task set with utilization $U \leq 1$, the energy consumption is minimized if the CPU speed is set uniformly to $max\{S_{min}, US_{max}\}$.

In order to appreciate the energy savings resulting from static power management, assume conservatively that the idle power, g_{idle}, is equal to zero and assume that $g(S) = \alpha S^3$, for some constant α. If T_{lcm} is the least common multiple of the periods $T_1, \ldots, T_N$, then, executing at $S_{max} = 1$ during T_{lcm} will result in an energy consumption equal to $\sum_{i=1}^{N} g_i(S_{max}) \frac{C_i}{S_{max}} \frac{T_{lcm}}{T_i} = \alpha U T_{lcm}$. If the speed is set to US_{max}, then the energy consumption during T_{lcm} reduces to $\sum_{i=1}^{N} g_i(U) \frac{C_i}{US_{max}} \frac{T_{lcm}}{T_i} = \alpha U^3 T_{lcm}$, which is a factor of U^2 lower than $\alpha U T_{lcm}$. For example, if $U = 0.5$, then static power management consumes only 25% of the energy that is consumed without power management.

The optimality of the uniform speed based on the load assumes that the power functions, $g_i(S)$, are the same for all tasks. However, due to the fact that different tasks may use different hardware units and have different patterns of memory and cache usage, we may have a different power consumption function, $g_i()$, for each task τ_i. (The different power functions could refer to different tasks or different segments of the same task.) In this case, the energy consumption is minimized when each task τ_i executes at a speed S_i derived from solving the following optimization problem:

$$minimize\{\sum_{i=1}^{N} g_i(S_i) \frac{C_i}{T_i S_i}\}$$

such that

$$\sum_{i=1}^{N} \frac{C_i}{T_i S_i} \leq 1$$

$$S_{min} \leq S_i \leq S_{max}$$

The above formulation is obtained by noting that when the speed of executing τ_i is set to S_i, then each instance of τ_i executes for a time $\frac{C_i}{S_i}$, and thus the fraction of time allocated to τ_i (the time utilization) is increased from $\frac{C_i}{T_i}$ to $\frac{C_i}{T_i S_i}$. EDF can always meet the deadlines if the sum of utilizations of the tasks in the system is less than one.

After solving the above minimization problem (see [4] for solution techniques), the speed S_i of each task, τ_i, can be stored in its process control block. Before the scheduler dispatches τ_i, it calls a PMP to set the processor speed to S_i. That is, the processor speed change becomes part of the context switch operation.

In summary, static power management for both task-level and system-level aim at the same goal, namely fully utilizing the CPU in the system, assuming worst case execution scenarios. When a single task is involved, the task-level management is sufficient to determine the uniform speed of that task, but when several tasks are involved, the operating system must intervene because it is the only entity that knows about all tasks and their characteristics.

3.2 Dynamic (on-line) power management

Dynamic power management is based on the observation that tasks usually do not execute their worst case scenarios, and thus by using execution progress information, the processor speed can be adjusted during the execution to increase the energy saving beyond that achieved by static power management. To simplify the discussion, we assume that the power consumption function is the same for all the tasks, and that S_s is the optimal speed obtained by static power management.

When tasks do not execute their worst case scenario, slack (unused computation time) is generated dynamically in the system; the first task of a PMP is to compute this slack. One way of estimating this slack is to determine whether a task or task segment is running earlier than predicted by the worst-case scenario. For instance, if the worst case scenario indicates that a PMP should be reached at time t_{wc} and this PMP is actually reached at time t_{ac} ($t_{ac} \leq t_{wc}$), then the difference of $slack_{early} = t_{wc} - t_{ac}$ can be considered as an *earliness slack* which can be used to slow down the processor.

An alternate way of looking at the *earliness slack* is to consider the work (in terms of number of CPU cycles) that remains to be executed until the next deadline, D. Let Π_{wc} and Π_{av} be the worst case estimate and the average case estimate of that work, respectively. Given that static power management fully utilizes the processor assuming worst case execution scenarios, the speed S_s guarantees that the period from t_{wc} to the deadline D is exactly equal to $\frac{\Pi_{wc}}{S_s}$, which is the time needed to execute Π_{wc} at speed S_s. In other words, the slack

$D - t_{ac} - \frac{\Pi_{wc}}{S_s} = D - t_{ac} - (D - t_{wc}) = t_{wc} - t_{ac}$ is the slack time between the current time and the deadline, after accounting for the execution of Π_{wc} at speed S_s.

Given that the worst case execution scenario occurs very rarely, it is reasonable to assume that a more useful estimate of the slack is obtained by assuming the average case scenario for the remaining work, rather than the worst case scenario. For example, one may speculate that the usable slack is equal to $slack_{speculate} = slack_{early} + \frac{\Pi_{wc} - \Pi_{av}}{S_s}$. This speculation is based on the assumption that only the average case scenario for the remaining work will be executed, and that this work will execute at speed S_s. The rationale behind using $slack_{speculate}$ is to consider the average case behavior, which is more common than the worst case behavior.

After a PMP computes *slack*, it uses this slack to slowdown the execution of the next task or task segment. Assuming that the worst case estimate of the number of cycles in this next task or task segment is C, and that S_s is the execution speed calculated from static power management, then the time allocated to the execution of the C cycles is $\frac{C}{S_s}$. Adding the slack to this time, the new execution speed is computed as:

$$S_{next} = \frac{C}{\frac{C}{S_s} + slack} \tag{7.3}$$

However, it should be noted that there is a lower bound on the processor speed in order to guarantee that, in the worst case, the remaining work will be completed by the deadline D. Specifically, if the speed for executing the next task or task segment is S_{next}, then this execution will consume at most a time equal to $\frac{C}{S_{next}}$. Hence, the remaining time until the deadline, $D - t_{ac} - \frac{C}{S_{next}}$ should be at least large enough to guarantee that the worst case scenario of the remaining work, $\Pi_{wc} - C$, can finish by the deadline. Given that the maximum speed is S_{max}, this remaining work will require at least $\frac{\Pi_{wc} - C}{S_{max}}$ to execute. Hence,

$$D - t_{ac} - \frac{C}{S_{next}} \geq \frac{\Pi_{wc} - C}{S_{max}}$$

should always hold, which means that

$$S_{next} \geq \frac{C}{D - t_{ac} - \frac{\Pi_{wc} - C}{S_{max}}} = S_{feasible} \tag{7.4}$$

From Equations (7.3) and (7.4), we can put an upper bound on the amount of slack that can be used to compute the next speed. Namely,

$$slack \leq slack_{max} = (D - t_{ac} - \frac{\Pi_{wc} - C)}{S_{max}}) - \frac{C}{S_s} \tag{7.5}$$

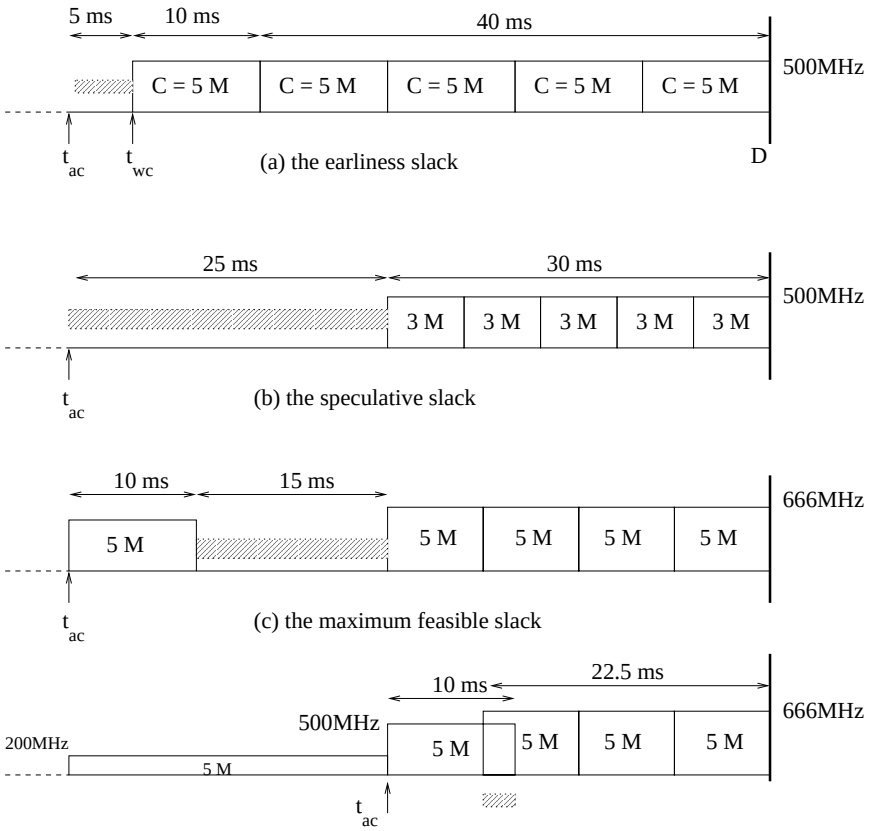

Figure 7.2. An example of slack computation; height of box is CPU speed for task.

In Figure 7.2 we show an example to clarify the above slacks. The deadline D is $50ms$ later than t_{wc} and the remaining Π_{wc} = 25 million cycles should execute at a speed $S_s = 500$ MHz to meet the deadline (in the figure, M denotes a million cycles). It is assumed that a PMP is placed every 5 million cycles of the remaining work. In the figure, the execution of C at speed S cycles is represented by a rectangle whose area is C. The height of the rectangle is S and its width, $\frac{C}{S}$, is the execution time. Assume that a PMP is reached at time $t_{ac} = t_{wc} - 5ms$ (recall that t_{wc} is reached in the worst case scenario). Figure 7.2(a) shows the earliness slack, while Figure 7.2(b) shows the speculative slack assuming that Π_{av} = 15 million cycles. Figure 7.2(c) shows the maximum slack assuming that the next task consumes 5 million cycles and that $S_{max} = 666MHz$.

In general, the PMP can select any value for the *slack* between zero and $slack_{max}$. In fact, a zero slack means that the PMP does not perform any dy-

namic speed adjustment and selects the speed computed by static power management. A slack equal to $slack_{max}$ is a very aggressive scheme which gives all the available slack to the next task or task segment. This scheme may slow down the next task or task segment too much without taking into consideration the remaining computations in the system, perhaps causing the CPU speed to be raised to S_{max}.

If a PMP is very aggressive in speculatively reducing the processor speed, the next PMP may be reached at a time, t'_{ac}, which is later than the time, t'_{wc}, predicted by the worst case scenario, and the maximum slack, $slack_{max}$, may be negative. The consequence of negative slack (too much aggressiveness) is the setting of speed above S_s in order to meet the deadlines. As shown in Figure 7.2(d), if $slack_{max} = 15ms$ were used, $S_{next} = 200$ MHz; if the next task executes its worst case scenario, then at the next PMP, $slack_{max}$ is equal to $-2.5ms$, which means that the speed has to be raised to 666 MHz in order to guarantee that the deadline will not be missed.

4. The overhead of speed management

Changing the speed of a CPU takes time and energy, an overhead that was ignored in the above estimation of the usable slack. In this section we study how long and how much energy is spent in each PMP, and later compare this overhead with the actual gains from DVS.

4.1 Time overhead

At every PMP, a time overhead is incurred for *computing* the new speed, S_{next}, and for *changing* the speed from the current speed, $S_{current}$, to S_{next} through a voltage transition in the processor's DC-DC switching regulator (resulting in a processor frequency change) and the clock generator. We denote changing both voltage and frequency by the term *speed change*. In order to guarantee that the deadlines are met, the above time overheads must to be considered.

When a PMP is added at the beginning of a segment $\tau_{i,(j)}$, the estimates $C_{i,(j)}$ and $C\_avg_{i,(j)}$ should be modified to include the number of cycles, F_c, needed to run the PMP code. From experiments with SimpleScalar 3.0, where we implemented speed setting and inserted PMPs in different applications, we observed that the overhead, F_c, of computing the new speed varied between 280 and 320 CPU cycles. Therefore, we consider F_c to be constant.

To change voltage, a DC-DC switching regulator is employed. This regulator cannot make an instantaneous transition from one voltage to another [15]. When setting a new speed, both the CPU clock and the voltage fed to the CPU need to be changed, incurring a wide range of delays. For example, the *Strong Arm SA-1100* is capable of on-the-fly clock frequency changes in the range of

59MHz to 206MHz where each speed and voltage change incurs a latency of up to 150 μsec [25], while the *lpARM* processor [29] (a low-power implementation of the *ARM*8 architecture) takes 15 μs for a full swing from 10 MHz to 100 MHz. Another example is the Transmeta TM5400, which is specifically designed for DVS [36]. Some systems can continue operation while changing speed and voltage [29, 15], but the frequency continues to vary during the transition period. A conservative approach, which we adopt, is to assume that the processor cannot execute application code during this period.

Hence, we assume that a fixed time, F_t, is needed for each speed step transition. That is, the time overhead for speed changes is $F_t \cdot d(S_{current}, S_{next})$, where $d(S_i, S_j)$ is a function that returns the number of speed steps needed to make a transition between S_i and S_j. In the Transmeta model, this function returns how many multiples of 33MHz is the difference between S_i and S_j.

The overhead of changing the speed should be accounted for in order to guarantee that deadlines are met. Specifically, the maximum feasible slack, $slack_{max}$, should be adjusted by subtracting this overhead before computing the new speed. Moreover, because $slack_{max}$ is computed assuming that the processor runs at S_{max}, we should allow the time for a future PMP to switch the processor speed to S_{max} to meet the deadline. This will require $F_t d(S_{next}, S_{max})$ to switch the speed to S_{max}. In other words, $slack_{max}$ computed by Equation (7.5) should be reduced by $F_t d(S_{current}, S_{next}) + F_t d(S_{next}, S_{max})$ to guarantee that deadlines are met.

4.2 Energy overhead

Given that a PMP executes code which is not part of the application code, the energy consumed during the execution of a PMP is an overhead that exists only because of the power management. In addition to the energy consumed in executing a PMP, there is an energy overhead associated with the change in voltage and frequency to change the CPU speed from $S_{current}$ to S_{next}. This overhead is proportional to the number of speed steps, $d(S_{current}, S_{next})$. In the simulations presented below, we will assume that the energy overhead for changing the speed is equal to $F_t \cdot d(S_{current}, S_{next}) \cdot g(S_{current})$. This means that during the speed change, the CPU consumes power at a rate equal to its consumption, while executing at speed $S_{current}$.

In the next two sections, we compare different power management schemes that use different methods for estimating the slack and different schemes for using the slack. These methods can take into consideration the energy and time overheads, as discussed above.

5. Task-level dynamic power management

Consider a task, τ_i, which is allotted a time D_i to execute, and assume that a PMP is executed at the beginning of segment j of this task. Moreover, assume that the PMP is actually invoked at time t_{ac}, even though this PMP is supposed to be called at time t_{wc}, according to the worst case scenario executing at speed S_s. In order to compute the speed, $S_{i,(j)}$, for executing segment $\tau_{i,(j)}$, the PMP needs to estimate the slack that it will use for speed adjustment. This slack can vary between zero and $slack_{max}$, which, due to the time and energy overheads should be modified from Equation (7.5) to

$$slack_{max} = \frac{C}{S_{feasible}} - \frac{C}{S_s} - F_t \cdot (d(S_{current}, S_{i,(j)}) + d(S_{i,(j)}, S_{max}))$$

where $S_{current}$ is the currently executing speed. Given that $S_{i,(j)}$ is not known, it can be safely and conservatively approximated by S_{min} to obtain

$$slack_{max} = \frac{C}{S_{feasible}} - \frac{C}{S_s} - F_t (d(S_{current}, S_{min}) + d(S_{min}, S_{max}))$$

As described in Section 3, a PMP can use all or part of $slack_{max}$ to compute the speed of the segment. Then, for $0 \leq slack \leq slack_{max}$,

$$S_{i,(j)} = \frac{C_{i,(j)}}{\frac{C_{i,(j)}}{S_s} + slack} \tag{7.6}$$

In [26], three specific schemes were described for the computation of the slack. The first scheme, called **Greedy**, uses $slack = slack_{early}$. That is, it uses the *earliness slack* to adjust the speed of the next task segment. The second scheme, called **Proportional**, uses $slack = slack_{early} \frac{C_{i,(j)}}{\Pi\_wc_{i,(j)}}$. That is, it distributes the earliness slack to all the future segments and gives to the next segment only a proportional amount of that slack. The third scheme, called **Statistical**, uses $slack = min\{slack_{max}, slack_{speculate} \frac{C_{i,(j)}}{\Pi\_avg_{i,(j)}}\}$. That is, it assumes an average case scenario for the computation of the slack and distributes this slack to all the remaining segments in a proportional fashion.

We implemented a simulator to experiment with the different power management schemes. Inputs to the simulator are a segment flow graph, the ratio of the worst case execution time to the best case execution time (β), and the system utilization or load (U). In the graphs below, the energy consumption values are normalized to the energy consumed by the Static scheme. We ran experiments using a synthetic program that has a segment flow graph similar to one shown in Figure 7.1 where C_i values, the loop indexes, and the actual execution times for each segment are drawn from a normal distribution.

From our experiments with SimpleScalar, we extracted $F_c = 300$ cycles and $F_t = 320$ cycles for a single step of 33 MHz. These values are used in the calculation of the total overhead associated with each PMP, and consequently augmented the C_i of each task i.

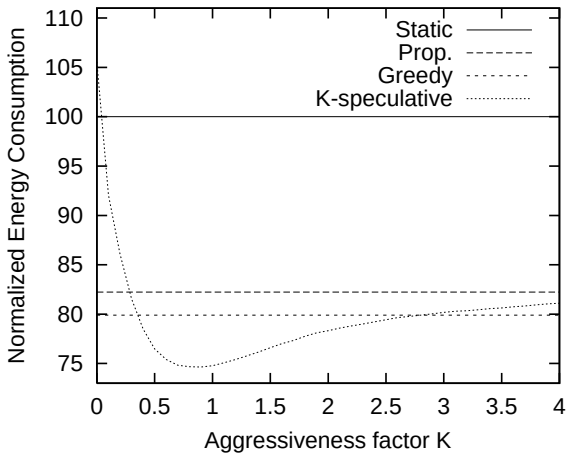

Figure 7.3. Simulation results for $\beta = 3$ and load = 0.5.

In Figure 7.3, we show the results of simulating task-level dynamic power management for the Greedy and the Proportional schemes mentioned above, as well as for a scheme which uses $slack = min\{slack_{max}, Kslack_{speculate}\frac{C_{i,(j)}}{\Pi\_avg_{i,(j)}}\}$, where K represents the aggressiveness in using the slack for slowing down the next segment. We call this scheme, which is more flexible, **K-speculative**. Note that the Statistical scheme is the K-speculative scheme with $K = 1$. As shown in the figure, Proportional and Greedy consume less energy than Static, because of the dynamic slack reclaiming. In addition, varying the aggressiveness factor K dramatically affects the K-speculative scheme: for $K = 0$, Static outperforms K-speculative since the latter does not take any advantage of the reclaimed slack, *and* it has to pay the overhead cost at each PMP, while Static has overhead only once, at the beginning of execution. At the other end of the spectrum, it is clear that K-speculative tends to Greedy when K increases. K-speculative reaches its minimum consumption around $K = 1$, that is, approximately the average behavior of the system.

To study the effect of the variability of the workload, we experimented with the same program at different values of $\beta = \frac{C_i}{C_i - 2(C_i - C\_avg_i)}$ (see Figure 7.4). We found that the larger variability, the more dynamic slack to reclaim, and thus the smaller energy consumption. However, the pattern is very similar for all values of β.

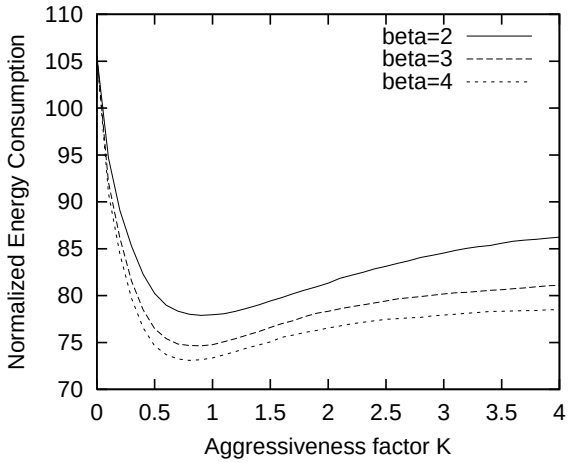

Figure 7.4. Simulation results for different values of β with load = 0.5.

6. System-level dynamic power management

The first observation that we make in this section is that the speed adjustment schemes used in system-level dynamic power management for frame-based periodic tasks are similar to the schemes described in the previous section for task-level power management. The difference is that task-level management adjusts the speed for n segments of the same task with a common deadline, D, while system-level management adjusts the speed for N frame-based tasks with a common deadline, T.

The second observation is that it is a good idea to combine task-level and system-level dynamic management, since the former carries out adjustments within the task's allotted time, while the latter attempts to adjust speeds between tasks (when context switches occur). This way, system-level complements the task-level dynamic management, allowing for benefits stemming from both compilers and operating systems.

Thirdly, we note that the case of general periodic tasks is more complicated than frame-based systems because each task has a different deadline. For simplicity of presentation, we do not take the PMP overhead (of computing or changing speeds) into account in this section. The principles are the same as outlined in Section 4. To detect earliness slack, we perform comparisons between the actual execution history and the canonical schedule S^{static}, which is the static optimal schedule on which every instance presents its worst-case workload to the processor and instances run at the constant speed S_s. The CPU speed is adjusted (i.e., a PMP is executed) only at task dispatch times.

Due to the periodic nature of the tasks we consider, it is impractical to produce and keep the entire static optimal schedule S^{static} (length of S^{static} could be T_{lcm}) during the execution. To address feasibility and efficiency while tasks execute, complete, and re-arrive dynamically, we choose to construct and update a data structure (called **α-queue)** that helps to compute the earliness slack at dispatch time. At any time t during actual execution, the α-queue contains information about the (up to) N tasks that would be active (i.e., running or ready) at time t in the worst-case static optimal schedule S^{static}. That is, α-queue is the ready queue of S^{static} at time t and it is constructed and maintained so that the remaining execution time, $rem_i(t)$, of τ_i at time t in S^{static}, under the static optimal speed S_s, is available (note that $rem_i(t) > 0$).

In this chapter, we assume that tasks are scheduled and dispatched according to EDF* policy, which is the same as EDF [20], but if deadlines are the same, the task with the earliest arrival time and then lowest task id has the highest priority. This EDF* *priority ordering* is essential in our approach because it provides a total order on the priorities. (Any scheduling policy that provides total order on priorities will work as well.) We denote the EDF* priority-level of the task i by d_i^*; low values denote high priorities.

To relate the α-queue with the computation of earliness slack, let $w_i^S(t)$ denote the remaining worst-case execution time of task τ_i under the speed S at time t. Note that when task τ_x is being dispatched, tasks with higher priority that are still in the α-queue must be already finished in the actual schedule (since τ_x currently has the highest EDF* priority), but they would have not yet finished in S^{static}.

Therefore, for any task τ_x which is about to execute, any unused computation time (slack) of any task in the α-queue having strictly higher priority than τ_x will contribute to the earliness of τ_x along with the already-finished portion of τ_x in the actual schedule. That is, *total earliness* of τ_x is no less than $\varepsilon_x(t) = \sum_{i|d_i^* < d_x^*} rem_i(t) + rem_x(t) - w_x^{S_s}(t) = \sum_{i|d_i^* \leq d_x^*} rem_i(t) - w_x^{S_s}(t)$.

The α-queue can be easily implemented using the following rules:

R1. Initially the α-queue is empty.

R2. Upon arrival, each task τ_i "pushes" its worst-case execution time under speed S_s to the α-queue in the correct EDF* priority position (this happens only once for each arrival, no re-push at 'return from preemptions').

R3. As time elapses, the elements in the α-queue are updated (consumed) accordingly: the rem_i field at the head of α-queue is decreased with a rate equal to that of the passage of time. Whenever the rem_i field of the head reaches zero, that element is removed from α-queue and the update continues with the next element. No update is done when the α-queue is empty.

Note that, *at time t, the α-queue, updated according to the rules R1, R2 and R3, contains only the tasks that would be ready at time t in the static optimal schedule* S^{static}. This observation stems from the following: (a) α-queue is ordered according to EDF* order, (b) every arriving task pushes its remaining worst-case execution time according to S_s into the α-queue only once, (c) the queue is updated only at the head, reflecting the fact that only the task with the highest EDF* priority would be running in S^{static}, and (d) a task that would have finished in S^{static} (i.e., $rem = 0$) is removed from the α-queue. This effectively yields *a dynamic image* of the ready queue in S^{static} at time t.

Note that the dynamic reduction of rem_i in R3 above does not need to be performed at every clock cycle; instead, for efficiency, we perform the reduction whenever a task is preempted or completes, by taking into account the time elapsed since the last update. The above approach relies on two facts. First, the system-level speed adjustment decision will be taken only at arrival/preemption and completion times, and it is necessary to have an accurate α-queue only at these points (if speeds are to be changed at other points like the task-level PMPs, the update of rem_i must reflect that). Second, between these points, each task is effectively executed non-preemptively.

As mentioned in Section 3 any specific algorithm should specify the *exact* amount of earliness slack. One natural choice is to use $\varepsilon_x(t)$, that is, to reduce the speed so as to profit from the full earliness. We call this simply *Dynamic Reclaiming Algorithm* (DRA).

6.1 Speculative Speed Reduction

Another dimension of the reclaiming process is based on the fact that we may speculate on how early tasks will finish. This speculative move is similar to the one for frame-based systems, but the periodic model has different deadlines and periods per task. With tasks arriving and departing dynamically, we need to make sure that no deadlines are violated while speculatively reducing the speed of the current task.

To simplify this speculative slack management, we restrict the speculative power management to occur only when we can limit their effects upto the next event (NE) that corresponds to the arrival/deadline of any task. When we can ensure that a ready task τ_x will not finish beyond NE, we can guarantee that it will complete before its deadline (because, by definition, it is not later than NE). Even then, we may need to increase the speed of other tasks $\tau_{x+1}, \ldots, \tau_r$ that also complete before NE to guarantee timeliness, since these tasks may be delayed if a worst-case scenario occurs for τ_x. Clearly, all these speed adjustments should adhere to S_{min} and S_{max}.

In addition to tasks that finish before NE, even the highest priority ready task that completes after NE, τ_{r+1}, may provide a portion of its time allocation

under certain conditions. For simplicity of presentation, we do not describe this more complex scheme (for details, see [5]).

As in frame-based systems, after computing $slack_{speculate}$ and $slack_{early}$, we can assign all or some of this slack to the next task to be executed. This is controlled by an aggressiveness parameter that controls how much of the slack will be used by the next task and how much will be "saved" for future tasks. The computation of these slacks is very similar to the one in Section 5, where the deadline D is replaced by the next event NE. In fact, if $t_{wc} - t_{ac} > 0$, we will be able to slow down task τ_x. Further, if there are other ready tasks that will complete before NE, we need change the speed settings for these tasks.

6.2 Evaluation of the Dynamic Schemes

In order to experimentally evaluate the performance of DRA, we implemented a periodic scheduling simulator for EDF* policy. We implemented the following schemes: (a) **Static** uses constant speed S_s, and switches to power-down mode (i.e., $S = S_{min}$) whenever there is no ready task; (b) **DRA**, which reflects only using $slack_{early}$, and (c) **SPECULATE**, which uses $slack_{speculate}$. All graphs shown here are normalized to the Static scheme, and use the best available aggressiveness parameter (which, similar to frame-based systems, is around 1).

In our experiments, we investigated the average performance of the schemes over a large spectrum of worst-case utilization U (or load) and variability in actual workload (β). The periods of the tasks were chosen randomly in the interval [1000, 32000], S_{min} is set to 0.1, and S_s is set to U. The results shown here focus on the average energy consumption of task sets containing 30 tasks each and random values from a normal probability distribution function; results with different number of tasks and uniform distribution are rather similar [2]. The mean and the standard deviation for any task, τ_i, are set to C_avg_i and $\frac{C_i - C\_avg_i}{3}$ respectively, for a given β, as suggested in [33]. These choices ensure that, on the average, 99.7% of the execution times fall in the interval $[C_i - 2(C_i - C\_avg_i), C_i]$. For each task set, we measured the energy consumption using a cubic power/speed function [15].

Effect of Utilization. We observed that the energy consumption has very little variation when the utilization of the task set (i.e., U) is changed. This is because the use of optimal speed S_s results in having very similar *effective* utilization, for any value of U. In other words, when the utilization decreases, the speed decreases making the CPU fully utilized.

Effect of β. The simulation results confirmed our prediction that the energy consumption would be highly dependent on the variability of actual

workload. The average energy consumption of the task sets, as a function of β, with $U = 0.6$, is shown in Figure 7.5.

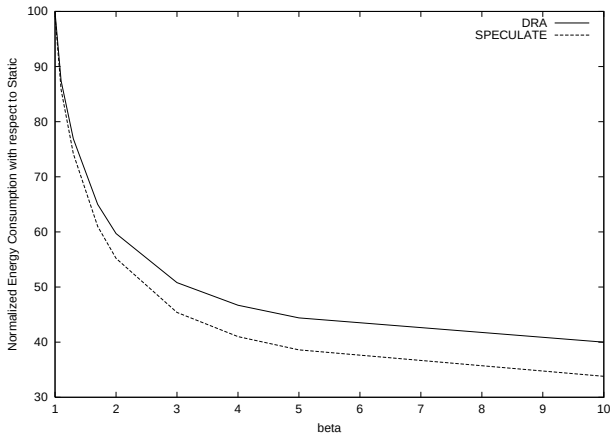

Figure 7.5. Effect of variability in actual workload (load = 60%)

- When $\beta = 1$, there is no CPU time to reclaim dynamically, and thus the energy consumption is the same for all three techniques, as expected. However, once the actual workload starts decreasing (that is, increasing β), DRA and SPECULATE are able to reclaim unused computation time and are able to save additional energy.
- Once we increase the β beyond 4, power savings of DRA and SPECULATE continue to increase, but the improvement is not as impressive as the case where that ratio is ≤ 4. This is because the expected workload converges rapidly to 50% of the worst-case workload with increasing β (remember that the mean of our probability distribution is C_avg_i.)

7. Maximizing reward while satisfying time and energy constraints

So far, we have assumed that the main goal of power management is to minimize the energy consumption while meeting timing constraints. In this section, we will consider a different model in which the goal of the power management is to maximize the system *value* while meeting both timing and energy constraints. In this *reward based* model, each task τ_i has a certain "value", v_i. Given timing and energy constraints, the goal is to select the tasks to execute and the speed at which these tasks should execute such that the total value of the system is maximized. In other words, the goal is to find a subset M of $\{1,\ldots,N\}$, such that when the tasks τ_i, $i \in M$, execute at speed S_i, the timing and energy constraints are satisfied, and the sum of the values of the

tasks in M is maximized. To formalize the problem assume that T_{lcm} is the least common multiple of the N periods, $T_1, \ldots, T_N$, and that E_{lcm} is the energy that is available for consumption during one T_{lcm}. Then, the problem is to find the subset of tasks (the set M) and the execution speed S_i for each task τ_i in this subset to

$$maximize \quad \sum_{i \in M} v_i \tag{7.7}$$

subject to

$$\sum_{i \in M} \frac{C_i}{S_i T_i} \leq 1 \tag{7.8}$$

$$\sum_{i \in M} g_i(S_i) \frac{C_i}{S_i T_i} T_{lcm} \leq E_{lcm} \tag{7.9}$$

$$S_{min} \leq S_i \leq S_{max} \tag{7.10}$$

Inequality (7.8) guarantees temporal feasibility (i.e., all deadlines will be met) if EDF is used, inequality (7.9) guarantees that the energy budget will not be exceeded, and inequality (7.10) guarantees that the selected speeds are within the allowable speed bounds. The knapsack problem, which is shown to be NP-hard [24], is a special case of the above problem in which only the first inequality applies with $S_i = S_{max}$. Heuristic search algorithms [24] can be adapted to solve the problem with energy constraints.

A different model is the imprecise computation model [21] in which the value of a task, τ_i, depends on the number of cycles, X_i, that the task actually executes. This dependence is usually expressed in the form of a non-decreasing continuous value function $V_i(X_i)$, in which more reward is given when the task executes more cycles.

Formulated in terms of reward functions, the problem is now to find the allotment X_i, $i = 1, \ldots, N$, and the execution speeds S_i such that to

$$maximize \quad \sum_{i=1}^{N} V_i(X_i) \tag{7.11}$$

subject to

$$\sum_{i=1}^{N} \frac{X_i}{S_i T_i} \leq 1 \tag{7.12}$$

$$\sum_{i=1}^{N} g_i(S_i) \frac{X_i}{S_i T_i} T_{lcm} \leq E_{lcm} \tag{7.13}$$

$$X_i \leq C_i \tag{7.14}$$

$$S_{min} \leq S_i \leq S_{max} \tag{7.15}$$

This is still an open problem and one of our next research targets.

8. Concluding remarks

In this chapter, we have introduced and described the concept of power management points (PMPs) in real-time systems. The main purpose of a PMP is to manage energy consumption by adjusting the processor speed based on task profiling information (static power management) and/or execution progress information (dynamic power management). Both static and dynamic power management can be done at the task-level, at the system-level, or a combination thereof. Although the static scheme is based on the worst-case workload offered to the system, the dynamic schemes is based on the fact that computations often finish before their predicted worst-case workload. Therefore, the dynamic schemes can take advantage of reclaiming unused computation (based on past actual workload) and speculatively predicting future early completions (based on statistical information of future workload).

The basic concept of reclaiming unused computation time to manage processor speed was described for single processor systems. The extension of these principles to multiprocessor systems depends on the mechanism used for mapping tasks to processors. If such a mapping is done statically, then the extension is rather simple, while if the mapping is done dynamically, then the extension should consider the interaction between the scheduling and the mapping mechanisms. For more details about power management in multiprocessors, see [39, 41].

The overhead of power management is a very important factor that determines the effectiveness of using PMPs. Given that a PMP has to consume some energy to save energy, the issue of the optimal number of PMPs and the placement of these PMPs becomes a crucial factor for the overall energy efficiency of the system. In [1], the tradeoff between the cost and the benefits of the PMP is discussed in a simple computational model. The assessment of this tradeoff in more realistic environments is still open for research.

A basic assumption made in this chapter is that the speed of the processor can be changed continuously within a given range. Current variable speed processors, however, allow the processor speed to be set only to a set of discrete values. For instance, the Transmeta processor allows the speed to be set to increments of 33 MHz within the range from 200 to 700 MHz. The power management technique that we described for the continuous speed range can be safely applied to the discrete speed processors if the speed is set to the closest available speed that is faster than the calculated one. Such a speed setting may not be optimum but guarantees that the deadlines are met.

References

[1] N. AbouGhazaleh, D. Mosse, B. Childers and R. Melhem. Toward The Placement of Power Management Points in Real Time Applications. *Work-*

shop on Compilers and Operating Systems for Low Power (COLP'01), September 2001.

[2] H. Aydin. *Enhancing Performance and Fault Tolerance in Reward-Based Scheduling.* Ph.D. Dissertation, University of Pittsburgh, August 2001.

[3] H. Aydin, R. Melhem, D. Mossé and P.M. Alvarez. Optimal Reward-Based Scheduling for Periodic Real-Time Tasks. *IEEE Transactions on Computers* 50(2): 111-130, February 2001.

[4] H. Aydin, R. Melhem, D. Mossé and P.M. Alvarez. Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics. In *Proceedings of the 13th EuroMicro Conference on Real-Time Systems (ECRTS'01)*, Delft, Netherlands, June 2001.

[5] H. Aydin, R. Melhem, D. Mossé and P.M. Alvarez. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *Proceedings of* 22^{nd} *Real-Time Systems Symposium*, December 2001.

[6] D. Burger, and T. Austin. The SimpleScalar Tool Set, Ver 2.0 . *University of Winsconsin-Madison CS Technical report no. 1342*, 1997.

[7] G. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997.

[8] E. Chan, K. Govil, and H. Wasserman. Comparing Algorithms for Dynamic Speed-setting of a Low-Power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM 95)*, November 1995.

[9] R. Ernst and W. Ye. Embedded Program Timing Analysis based on Path Clustering and Architecture Classification. In *Computer-Aided Design (ICCAD)'97*. pp. 598-604.

[10] V. Gutnik and A. Chandrakasan. An Efficient Controller for Variable Supply Voltage Low Power Processing. *Symposium on VLSI Circuits*, pp.158-159, 1996.

[11] F. Gruian. Hard Real-Time Scheduling for Low-Energy Using Stochastic Data and DVS processors. *International Symposium on Low Power Electronics and Design*, 2001.

[12] P. J. M. Havinga and G. J. M. Smith. Design Techniques for Low-power Systems. *Journal of Systems Architecture*. Vol. 46:1, 2000

[13] I. Hong, D. Kirovski, G. Qu, M. Potkonjak and M. Srivastava. Power optimization of variable voltage core-based systems. In *Proceedings of the 35th Design Automation Conference, DAC'98*

[14] I. Hong, M. Potkonjak and M. B. Srivastava. On-line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *Computer-Aided Design (ICCAD)'98*. pp. 653-656.

[15] I. Hong, G. Qu, M. Potkonjak and M. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors.

In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, December 1998.

[16] Intel, Microsoft, and Toshiba. Advanced Configuration and Power Management Interface (ACPI) Specification, 1999. www.intel.com/ial/WfM/design/pmdt/acpidesc.htm.

[17] C. M. Krishna and Y. H. Lee. Voltage Clock Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, Washington D.C., May 2000.

[18] P. Kumar and M. Srivastava. Predictive Strategies for Low-Power RTOS Scheduling. In *Proceedings of International Conference on Computer Design*, 2000.

[19] P. Kumar and M. Srivastava. Power-aware Multimedia Systems Using Run-Time Prediction In *Proceedings of International Conference on VLSI Design*, 2001.

[20] C.L. Liu and J.W.Layland. Scheduling Algorithms for Multiprogramming in Hard Real-time Environment. *J. of ACM* 20(1): pp.46-61, 1973.

[21] J. W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, C. Chung, J. Yao and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer,* 24(5): 58-68, May 1991.

[22] J. W. S. Liu. *Real-Time Systems*. Prebtice Hall, 2000.

[23] J. R. Lorch and A. J. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, Cambridge, MA, June 2001.

[24] S. Martello and P. Toth. "Knapsack Problems", *Wiley, InterScience Series in Discrete Mathematics and Optimization*, 1990.

[25] R. Min, T. Furrer, and A. Chandrakasan. Dynamic Voltage Scaling Techniques for Distributed Microsensor Networ ks. *IEEE VLSI Workshop*, 2000.

[26] D. Mossé, H. Aydın, B. Childers and R. Melhem. Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications. *Workshop on Compilers and Operating Systems for Low-Power (COLP'00),* Philadelphia, PA, October 2000.

[27] W. Namgoang, M. Yu and T. Meg. A High Efficiency Variable-Voltage CMOS Dynamic DC-DC Switching regulator. *IEEE International Solid-State Circuits Conference*, pp.380-391

[28] M. Pedram. Power Minimization in IC Design: Principles and Applications. *ACM Transactions on Design Automation of Electronics Systems*. 1:1 - pp. 3-56, January 1996.

[29] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the lpARM Microprocessor System. *International Symposium on Low Power Electronics and Design 2000,* pp.96-101, 2000.

[30] J. Pouwelse, K. Langendoen and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. *7th International Conference on Mobile Computing and Networking (MOBICOM)*, Rome, Italy, July 2001.

[31] V. Raghunathan, P. Spanos and M. Srivastava. Adaptive Power-Fidelity in Energy-Aware Wireless Embedded Systems. In *Proceedings of 22nd Real-Time Systems Symposium (RTSS'2001)*, December 2001.

[32] D. Shin, J. Kim and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18:(2), March-April 2001

[33] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proceedings of the 36th Design Automation Conference, DAC'99*, pp. 134-139.

[34] M. 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*, 4(1): 42-55, 1996.

[35] http://www.specbench.org

[36] http://www.transmeta.com

[37] A. Vahdat, A. R. Lebeck and C. S. Ellis. Every Joule is Precious: A Case for Revisiting Operating System Design for Energy Efficiency. In *the 9th ACM SIGOPS European Workshop*, September 2000.

[38] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[39] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest and R. Lauwereins. Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs. *IEEE Design & Test of Computers*, Sep/Oct 2001

[40] F. Yao, A. Demers and S. Shenker. A Scheduling Model for Reduced CPU Energy. *IEEE Annual Foundations of Computer Science*, pp. 374 - 382, 1995.

[41] D. Zhu, R. Melhem, and B. Childers. "Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems", RTSS'01 (Real-Time Systems Symposium), London, England, Dec 2001