Feedback EDF Scheduling of Real-Time Tasks Exploiting Dynamic Voltage Scaling [*]

Yifan Zhu and Frank Mueller (mueller@cs.ncsu.edu)
Department of Computer Science/ Center for Embedded Systems Research, North Carolina State University, Raleigh, NC 27695-7534, phone: +1.919.515.7889, fax: +1.919.515.7925

Abstract.

Many embedded systems are constrained by limits on power consumption, which are reflected in the design and implementation for conserving their energy utilization. Dynamic voltage scaling (DVS) has become a promising method for embedded systems to exploit multiple voltage and frequency levels and to prolong their battery life. However, pure DVS techniques do not perform well for systems with dynamic workloads where the job execution times vary significantly. In this paper, we present a novel approach combining feedback control with DVS schemes targeting hard real-time systems with dynamic workloads. Our method relies strictly on operating system support by integrating a DVS scheduler and a feedback controller within the earliest-deadline-first (EDF) scheduling algorithm. Each task is divided into two portions. The objective within the first portion is to exploit frequency scaling for the average execution time. Static and dynamic slack is accumulated for each task with slack-passing and preemption handling schemes. The objective within the second portion is to meet the hard real-time deadline requirements up to the worst-case execution time following a last-chance approach. Feedback control techniques make the system capable of selecting the right frequency and voltage settings for the first portion, as well as guaranteeing hard real-time requirements for the overall task. A feedback control model is given to describe our feedback DVS scheduler, which is used to analyze the system's stability. Simulation experiments demonstrate the ability of our algorithm to save up to 29% more energy than previous work for task sets with different dynamic workload characteristics.

Keywords: Real-Time Systems, Scheduling, Dynamic Voltage Scaling, Feedback Control

[*] This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.

Preliminary versions of parts of this work appeared in the ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02) (Dudani et al., 2002), in the Workshop on Compilers and Operating Systems for Low Power 2002 (Zhu and Mueller, 2002) and in the IEEE Real-Time Embedded Technology and Applications Symposium 2004 (Zhu and Mueller, 2004a).

1. Introduction

Energy consumption is a major concern for today's embedded systems due to their limited battery capacity. The availability of services provided by mobile devices powered by batteries is limited by the amount of power drawn from the batteries over time. The energy consumption is also a cost factor for non-battery powered systems since the operational costs of embedded systems running non-stop may be significant. Contemporary embedded processors support multiple voltage and clock frequency settings. The energy consumption of a processor can be reduced by modulating voltage and frequency dynamically because the power dissipation of a CMOS circuit is proportional to its clock frequency and its voltage square(Chandrakasan et al., 1992). We refer to dynamic voltage scaling (DVS) in the following whenever frequency or voltage are changed during execution.

DVS is particularly attractive to real-time systems for two reasons. First, real-time systems commonly execute periodic tasks, which implies that entering a low-power mode such as a standby or hibernation state is usually infeasible. The overhead associated with entering and leaving those low-power modes often causes deadline misses. Second, real-time requirements force system designers to choose embedded processors that are powerful enough to meet the worst-case execution demands although these demands may rarely occur. As a result, the system utilization is often kept at a low level with a high energy consumption to ensure operational safety. DVS techniques have shown their potential for real-time systems by pursuing lower energy consumption while maintaining system real-time requirements at the same time. This opens new opportunities for reduced operational costs for both embedded and non-embedded applications whose operation is constrained by battery capacity.

The potential to save energy by combining DVS techniques with operating system scheduling has been demonstrated by prior work, and significant savings have been reported for general-purpose computing systems (Govil et al., 1995; Grunwald et al., 2000; Krishna and Lee, 2000; Lorch and Smith, 2001; Pering et al., 1995; Weiser et al., 1994; Pouwelse et al., 2000; Gruian and Kuchcinski, 2001) as well as real-time systems (Hong et al., 1998a; Hong et al., 1998b; Lee and Krishna, 1999; Shin et al., 2000; Pillai and Shin, 2001; D. Shin and Lee, 2001; Mosse et al., 2000; Gruian, 2001; Aydin et al., 2001; Kang et al., 2002). However, pure DVS techniques do not perform well for dynamic systems where the system workloads vary significantly. Traditionally, hard real-time scheduling relies on *a priori* knowledge of the worst-case execution time (WCET) of a task to guarantee the schedulability of the system. A safe upper bound on the WCET of a task can be provided through static analysis, dynamic analysis or a combination of both techniques (Puschner and Koza, 1989; Park, 1993; Harmon et al., 1992; Zhang et al., 1993; Lim et al., 1994; Healy et al., 1995; Arnold et al., 1994; Li et al., 1995; Li et al., 1996; Ferdinand et al., 1997; Mueller, 2000; Wegener and Mueller, 2001). However, experiments have shown a wide variation between longest and shortest execution times for many actual applications. In (Wegener and Mueller, 2001), actual execution times of real-world embedded tasks are observed to vary by as much as 87% relative to their measured WCET. Budgeting for the WCET may result in excessive energy consumption even though actual utilizations are low compared to the worst case. Many of the existing hard real-time DVS schemes are not able to adapt well to dynamically changing workloads. For example, we compared the energy consumption of Look-ahead RT-DVS (Pillai and Shin, 2001) between constant workloads and fluctuating workloads, as depicted in Figure 1. Both workloads contain 3 tasks defined as T_1={3,8}, T_2={3,10} and T_3={1,14}, where T_i={WCET,Period} for $i = 1...3$, as described in (Pillai and Shin, 2001). The constant workloads consist of tasks whose actual execution times always equal 50% of their WCET. The fluctuating workloads consist of tasks with an average execution time of 50% WCET, but actual execution times fluctuate between 20% and 80% of their WCET (following variation patterns similar to Figure 11, discussed later). Figure 1 shows that, in the worst case, Look-ahead RT-DVS degrades up to 61% for fluctuating workloads.

The objective of our work is to develop a novel DVS technique targeting such dynamic changing workloads. We combine feedback control theory with DVS for hard real-time systems. Feedback control techniques have been shown to be a promising approach for real-time scheduling in prior work (Lu et al., 1999; Lu et al., 2002b; Minerick et al., 2002). But all of them are for soft real-time systems, where occasional deadline misses are acceptable. Our work extends beyond previous work and is, to the best of our knowledge, the first study of using feedback control techniques on DVS for hard real-time systems. On one hand, feedback techniques enable the system to select the right frequency/voltage settings so that energy consumption is significantly reduced. On the other hand, feedback control helps to guarantee the timing constraints of hard real-time tasks so that no tasks ever miss their deadlines.

This paper is structured as follows. In Section 2, we give a framework overview of the feedback-DVS scheme. We then describe the different elements of our feedback-DVS framework in detail, *i.e.*, the voltage-frequency selector in Section 3 and the feedback controller in Section 4. Section 5 is an example showing how our scheme works on practical task sets. Section 6 presents the experimental results to demonstrate the performance of our feedback-DVS scheme under different workload conditions. Section 7 discusses related work, and Section 8 summarizes our efforts.

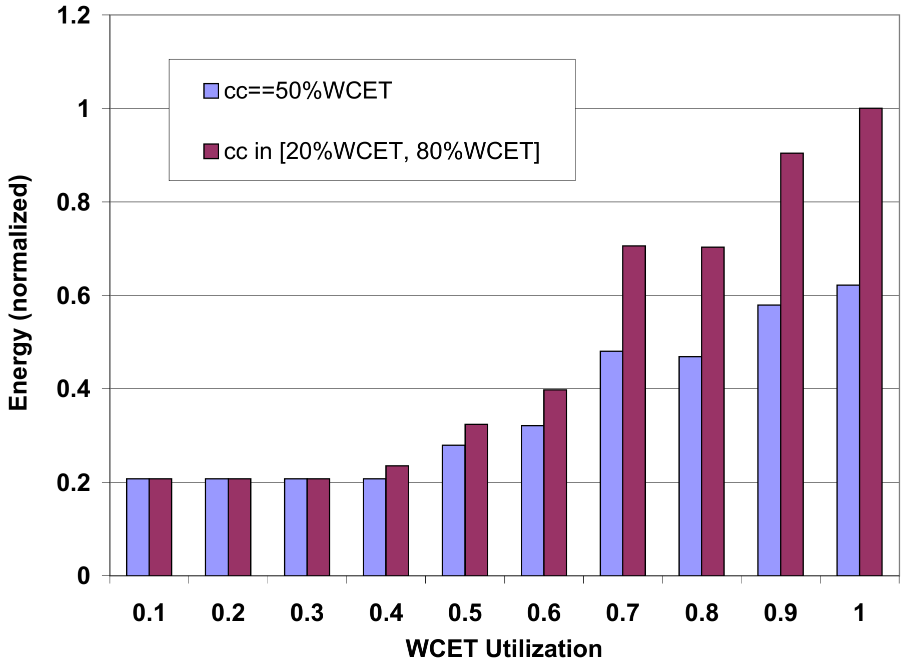


Figure 1. Look-ahead RT-DVS Energy for Constant/Fluctuating Workload

2. Feedback-DVS Framework Overview

Prior research on DVS for hard real-time system was primarily concerned with guaranteeing the schedulability of the task sets while energy consumption is minimized. But in a dynamic real-time environment where the workloads vary significantly from time to time, the DVS scheduler should not only produce a valid processor speed for each scheduling unit, it should also be able to adapt to the ever-changing workloads as fast as possible. One important performance metric of such a system is how fast the DVS scheme can adjust the processor according to different workloads so that energy consumption is significantly reduced. To address this issue, we propose a framework called feedback dynamic voltage scaling (feedback-DVS). In this framework, we consider the scheduling problem in hard real-time systems with the earliest deadline first (EDF) policy. The framework is based on feedback control that incrementally corrects system behavior to achieve its targets, while the hard real-time timing requirements are still preserved. We assume that the processor can operate at several discrete voltage/frequency levels, which represents contemporary processor technology on support of DVS. When there is no task running on the processor, the processor enters an idle state at a particular voltage/frequency level, usually the lowest voltage/frequency level on that processor.

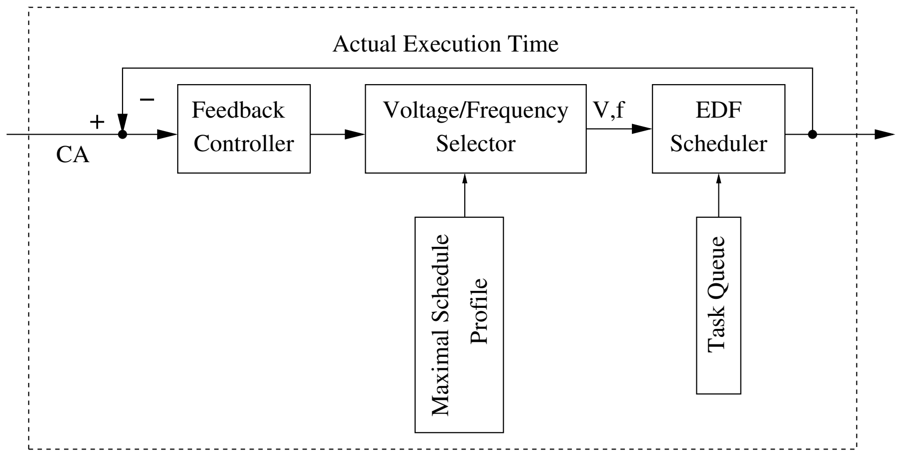


Figure 2. Feedback-DVS Framework

We use a periodic, fully preemptive and independent task model in our feedback DVS framework. We assume there are n tasks in total, T_1, T_2, ..., and T_n. Each task T_i is defined by a tuple (P_i, C_i), where P_i is the period of T_i, and C_i is the measured worst-case execution time of T_i. Each task's relative deadline d_i is equal to its period, and all tasks start at time 0. The periodically released instances of a task are called jobs. T_{ij} is used to denote the j^{th} job of task T_i. Its release time is $P_i * (j - 1)$ and its deadline is $P_i * j$. The hyperperiod H of the task set is defined as the least common multiple (LCM) of the tasks' periods. The schedule repeats at the end of each hyperperiod.

Feedback control is one of the fundamental mechanism for dynamic systems to achieve equilibrium. In a feedback system, some variables, *i.e.*, controlled variables, are monitored and measured by the feedback controller and compared to their desired values, so-called set points. The differences (errors) between the controlled variables and the set points are fed back to the controller for further actions. Corresponding system states are usually adjusted according to the differences to let the system variables approximate the set points as closely as possible.

Figure 2 depicts the framework of our feedback-DVS scheme. It consists of a feedback controller, a voltage-frequency selector, and an EDF scheduler. The feedback controller calculates the error from the difference between the actual execution time of a task and C_A, the execution time of the first portion of each task (detailed in the task-splitting scheme in the next section). The voltage-frequency selector chooses a voltage/frequency level according to the error and the maximal schedule profile. The error is used to adjust the estimation of the execution time for the next task. The maximal schedule profile includes a running scenario of the task set from the start time 0 to the end of a hyperperiod. It is generated offline assuming each task's actual execution time always equals its worst-case execution time. The voltage-frequency selector uses the information in the maximal schedule profile to choose the right voltage-frequency level without causing any deadline misses. As long as a voltage/frequency level is determined, the EDF scheduler schedules the next ready task at the specific processor speed. Tasks are scheduled according to EDF policy, *i.e.*, the task with the earliest deadline is given the highest priority. The actual execution time of each task is further fed back to the feedback controller for later decision making. The next two sections detail the mechanism of the voltage-frequency selector and the feedback controller in our feedback-DVS frame.

3. Voltage-Frequency Selector

The voltage-frequency selector is responsible for selecting a voltage-frequency pair each time a task is scheduled. Since power consumption increases proportionally to the processor frequency and to the square of the voltage in CMOS circuits (Ishihara and Yasuura, 1998), the minimal energy consumption is obtained by running every task at a uniform processor speed. But this is only a statically optimal solution. In a dynamic environment where a task's actual execution time is unknown until the task completes, it is not possible to derive the optimal uniform speed in advance. Our objective is to approximate a close-to-optimal solution by monitoring the actual execution time of each task. The start point of our scheme is the following inequation, which is a modification of the standard EDF (Liu and Layland, 1973) schedulability test:

$$\alpha^{-1}\frac{C_k}{P_k} + \sum_{i\in\{1,\ldots,n\}\setminus\{k\}} \frac{C_i}{P_i} \leq 1 \tag{1}$$

where α is a scaling factor defined as the ratio of the current processor frequency to the maximal available frequency, *i.e.*, $\alpha = f_k/f_m$. Instead of scaling at a single speed for all tasks, only the highest priority task (the task with the earliest deadline under EDF) is scaled. All remaining tasks are modeled to execute at the maximum frequency f_m in the future with a scaling factor of 1. The motivation of scaling only the current task comes from the observation that a greedy scheme usually gives a near-optimal result when optimal solutions are unavailable. In the following, we explain in detail the schemes used in our voltage-frequency selector to set the speed of a task.

3.1. Task Splitting

For each task, its α value depends on the total available slack when the task is scheduled. For example, at time 0, the available slack for the first task T_1 is derived from Inequation 1 as $P_1(1-\sum_{i=2}^{n}\frac{C_i}{P_i})$. Its α value is calculated as: $\alpha = \frac{C_1}{P_1(1-\sum_{i=2}^{n}\frac{C_i}{P_i})}$. In order to obtain an even lower speed for each task T_k and to make feedback control available for hard real-time systems, our scheme goes beyond that by splitting each task into two subtasks T_A and T_B. These two subtasks are allowed to execute at different frequency and voltage levels. As shown in Figure 3, T_B always executes at the maximum frequency

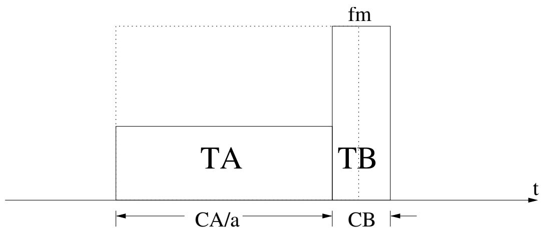


Figure 3. Task Splitting

level f_m, while T_A is able to execute at a lower frequency level than it could without task splitting. We expect that a task can finish its actual execution within T_A while reserving enough time in T_B to meet the deadline if it requires its full WCET. With this scheme, we can safely scale the frequency of T_A using available slack while T_B executes at maximum frequency following a last-chance approach (Chetto and Chetto, 1989). In the next section, we can also see that such a task splitting scheme is necessary for applying feedback control on hard real-time systems. By splitting each task into at most two subtasks, we add at most one additional speed change to each task and keep the impact of voltage and frequency switching overhead of our scheme to a minimum. Task splitting is supported by the operating system in a manner transparent to users. It can be implemented as a timer handler, triggered at the end of T_A, that changes frequency and voltage. The timer is set up upon dispatching T_A. If execution completes within T_A or a preemption occurs, the timer can be canceled and no additional overhead will the incurred. Only if execution cannot complete in T_A will the timer go off and trigger the DVS switch prior to executing the remainder of the task in T_B.

Let C_k, C_A and C_B be the worst-case execution cycles of task T_k and its two subtasks T_A and T_B, and s_k be the slack available to T_k when T_k is scheduled. From

$$C_k = C_A + C_B, \frac{C_A}{\alpha} + C_B = C_k + s_k \tag{2}$$

we derive

$$\alpha = \frac{C_A}{C_A + s_k} \tag{3}$$

Equation 3 shows that when task splitting is used, the scaling factor α depends not only on the amount of available slack (s_k), but also on the number of execution cycles assigned to T_A.

3.2. Static Slack Utilization

The type of slack available during the scheduling of a real-time system falls into two categories. One is static slack due to under-utilized system workloads. The other one is dynamic slack due to early completion of tasks. In order to exploit these two types of slack, we consider an actual schedule and a maximal schedule. The maximal or *worst-case* schedule is the schedule produced by a standard EDF algorithm when the execution time of every tasks' job has its maximum value given by the WCET. The actual schedule is the actual execution scenario produced by our feedback-DVS algorithm where the execution time of every tasks' jobs may be scaled. The maximal schedule is constructed offline in O(N) complexity, where N is the total number of jobs executed in a hyperperiod H. The static slack is exploited by adding an idle task, T_{n+1}, into the original task set to fill the gap between the actual utilization and 100% utilization. The idle task distributes the static slack throughout the entire hyper-period. Hence, static slack is not monopolized by a single task but evenly distributed. This also facilitates the online computation of the static slack. The idle task has a non-zero WCET but its actual execution time is always zero. The WCET and the period of the idle task are chosen in such a way that the total utilization of the new task set becomes 100%. In other words,

$$P_{n+1} = P_1, C_{n+1} = P_{n+1}(1-U), c_{n+1} = 0. \tag{4}$$

Notice that any other choice of idle task periods is legal. Most notably, the shortest period of any task P_1 and the longest one P_n are interesting choices. We consider these options since they affect the amount of static slack available for other tasks. We choose the shortest task period in the task set as the idle task's period to ensure that there is at least one idle task being released between any actual task's invocation to provide static slack for that task. The total static slack generated by idle task T_{n+1} in the interval $[t1..t2]$ is denoted by:

$$idle(t1...t2) = \sum_{t1..t2} idle\ slots \tag{5}$$

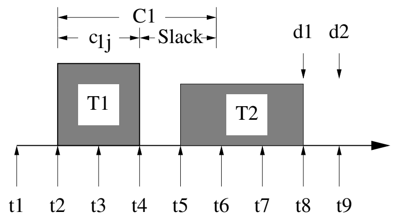


Figure 4. Dynamic Slack Passing

3.3. Dynamic Slack Passing

Dynamic slack passing is a technique to reduce the online complexity of calculating the amount of dynamic slack. It is based on the observation that slack generated by a task is usually not exhausted during the task's execution even if the processor speed has been reduced. Instead of computing each task's slack from the scratch, we let a previous task pass its unused amount of slack to the succeeding task. That slack is further augmented by any static idle task slots between the deadline of the previous task and the succeeding task.

When preemption does not exist, we express dynamic slack passing in terms of the release time r_{ij} of a task T_{ij} in the actual schedule, and the initiation time I_{pk} and the worst case completion time F_{pk} of the immediately previous task T_{pk} in the maximal schedule. The slack s_{ij} available to T_{ij} is defined as:

$$s_{ij} = \begin{cases} C_p - c_{pk} & if\ r_{ij} \leq I_{pk} + c_{pk} \\ F_{pk} - r_{ij} & if\ I_{pk} + c_{pk} < r_{ij} < F_{pk} \\ 0 & if\ r_{ij} \geq F_{pk} \end{cases} \tag{6}$$

An example is depicted in Figure 4. Let task T_1 with WCET C_1 and deadline t8 execute its j^{th} invocation with an actual execution time of c_{1j}. Assume that when T_1 was invoked at time t2, it inherited a total slack of S from its previous tasks. T_1 was then scaled to a suitable level with that slack and completed at time t4. The difference between C_1 and c_{1j} is the new slack dynamically generated by T_1. So the total slack available at t5 is $S = S + C_1 - c_{1j}$. Note the fact that the actual execution time c_{1j} may be less than, equal to, or greater than the worst-case execution time C_1 because of task scaling. If $C_1 > c_{1j}$, Equation 6 just adds the slack produced by the early completion of T_1 into the total slack. If $C_1 < c_{1j}$, Equation 6, in fact, reduces the total slack because the task exceeded its WCET in the maximal schedule (it is feasible under DVS as long as the available slack is not exceeded as well). The adjusted total slack is passed in full or in part to the next task T_2 depending on T_2's release time and deadline. Slack beyond T_2's release time and deadline cannot be used by T_2 and therefore will not be passed on to it.

When task preemption exists in the schedule, slack passing needs to be handled specially. In the next section, we derive formulas to compute the slack available for a preempting task.

3.4. Preemption Handling

Preemption handling follows a greedy scheme in that we try to pass as much slack as possible to scale the running task. We speculate on its early completion to aggregate more slack for following tasks. When preemption occurs, the preempted task will relinquish its remaining slack and pass it on to the next task, just as it does when a task completes. But there are two differences here. First, the preempted task itself cannot generate any slack based on its own execution at the preemption point since the task's completion time is unknown. Hence, no additional slack is added to its inherited total slack. Second, the preempted task still needs some time to complete its execution in the future. The remaining execution time must be reserved in advance to avoid future deadline misses caused by over-exploiting slack from other tasks. At the preemption point, the worst case remaining execution time $left_{ij}$ of the preempted task is:

$$left_{ij} = C_i - c_{ij} \times \alpha^{-1} \tag{7}$$

where c_{ij} is the actual execution time up to the preemption point. Our slack passing scheme promises that the preempted task will not miss its deadline by reserving the worst case remaining execution time from its slack:

$$s_{k,r} = s_k - left_{ij} \qquad (\textit{future slots}) \tag{8}$$

where s_k is derived from Equation 6 and the resulting slack $s_{k,r}$ can be passed to the next task.

Future slot allocation in this manner is essential to ensure the feasibility of the schedule under DVS. Future slots will be allocated only if the maximal schedule does not include sufficient slots for the preempted task's job between the preemption point and its deadline. We devised multiple schemes for reserving these slots.

– Forward sweep: When a task T_1 is preempted and requires $left_{1j}$ future slots, the preempting task T_2 deducts this amount from its available slack s. If $left_{1j} > s$, then T_2 remains without slack. If another task T_3 is initiated, the calculation repeats itself.
– Backward sweep: Future slots of T_1 are allocated in idle slots within the maximal schedule from its deadline d_1 backwards. Any of these idle slots become unavailable for slack generation, *i.e.*, these slots are excluded in Equation 6.

An example is depicted in Figure 5. The upper time line of idle slots presents a excerpt of the maximal schedule that depicts idle task allocations, only. The lower time line shows the dynamic schedule of tasks. Upon release of T_2 at t2, T_1 is preempted. Let us assume that T_1 does not have sufficient static slots (three slots) beyond t2 to finish its execution. Hence, it has to rely on future idle slots. During T_2's execution, T_3 is released. Both T_2 and T_3 have smaller deadlines than T_1 ($d_2 < d_3 < d_1$). Subsequently, T_1 only resumes some time after T_3 completes.

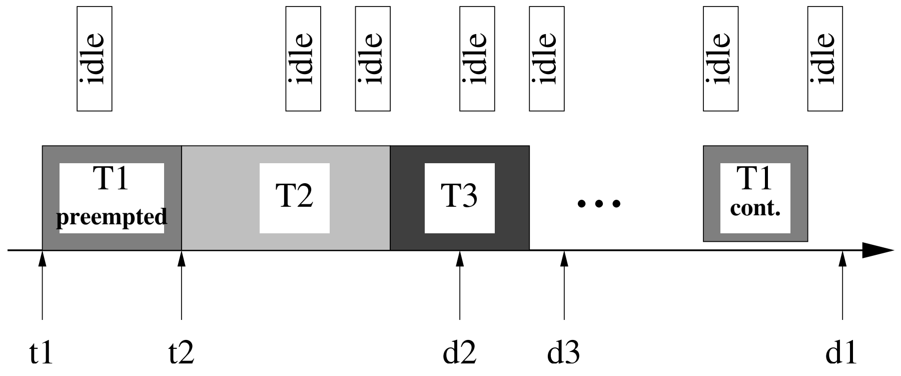


Figure 5. Future Slot Reservation

Future slot allocation of T_1 then depends on the chosen scheme. The forward sweep results in zero idle slack for T_2 and T_3 since idle slots during the tasks' periods are not sufficient to cover T_1's future needs of three slots at the respective invocation times. The backward sweep, on the other hand, reserves the last 3 idle slots (from d_1 backwards), such that T_2 and T_3 may consume at least two and one idle slots for scaling, respectively, even if they use up their time quantum in full.

Overall, the forward sweep is not as greedy as the backward sweep in the sense that earlier tasks may not be scaled due to T_1's future slots. A forward sweep is likely to result in zero slack for the preempting task T_2 if $P_2 << P_1$, *i.e.*, if its period is much shorter. There are simply fewer idle slots available, which may not suffice to cover T_1's future requirements. More idle slots past d_2 will be required in this case. The backward sweep always results in the most greedy approach in delaying the needs of T_1 as long as possible. This is consistent with the observation that early completion is likely to generate slack for each task, a property inherent to our algorithm.

4. Feedback Controller

Equation 3 shows that the scaling factor α depends not only on the amount of available slack but also on C_A, the number of execution cycles assigned to the first subtask T_A. Static slack utilization and dynamic slack passing, as described in the previous section, helps to determine the amount of slack available for each task. In this section, we focus on another key issue, *i.e.*, how to determine the value of C_A. Since C_A is based on the estimated worst-case execution time of the first subtask T_A, our objective is to let C_A approximate T_{ij}'s actual execution time c_{ij} so that T_{ij} can be completed before it gets into the second subtask T_B. If C_A was not exceeded by T_{ij}'s actual execution time c_{ij}, the entire part of T_{ij} could execute at the low frequency level corresponding to α. It would not be necessary for T_{ij} to switch to the maximum processor frequency. Hence, a near-optimal energy-aware schedule can be obtained.

In real-time applications, the actual execution time c_{ij} of each task T_i often experiences fluctuations over different intervals. Different jobs of a certain task usually present different actual execution times requirements. The fluctuations may result in tendencies leading to higher processing demands up to some point and receding demands after that peak point. Past work in dynamic real-time scheduling has demonstrated that adaptive techniques derived from control theory can enhance a schedule by reacting to tendencies in execution time fluctuations (Lu et al., 1999). In order to devise a DVS algorithm adaptive to such a dynamic environment, we integrate a closed-loop feedback controller into our DVS systems.

Feedback control is one of the fundamental mechanisms for dynamic systems to achieve equilibrium. PID-feedback control is a continuous feedback controller capable of providing sophisticated control response. The controlled variable can usually reach its set point and stabilize within a short period. A PID controller consists of three different elements, namely, proportional control, integral control, and derivative control. Proportional control influences the speed of the system adapting to errors, which is defined as the difference between the controlled variable and the set point, by a pure proportional gain item. Integral control is used to adjust the accuracy of the system through the introduction of an integrator on past error histories. Derivative control usually increases the stability of the system through the introduction of a derivative of the errors.

The PID feedback controller can be described in three major forms: the ideal form, the discrete form and the parallel form. Although the discrete form is often used in digital algorithms to keep tuning similar to electronic controllers, the parallel form is the simplest one. The integral and derivative actions are also independent of the proportional gain in the parallel form. We choose the following parallel form as the base of our PID feedback implementation:

$$output = K_P * \epsilon_i + \frac{1}{K_I} \int \epsilon_i \, dt + K_D \frac{d\epsilon_i}{dt} \tag{9}$$

where K_P, K_I and K_D are the proportional, integral and derivative parameters, respectively, and ϵ_i is the system error. The transfer function of the PID controller in the Laplace-domain (s-domain) is given by:

$$G_P(s) = K_P + \frac{K_I}{s} + sK_D \tag{10}$$

We integrated the above PID controller into our DVS scheme to control the number of execution cycles assigned to C_A. According to the objective described above, our system is presented as a multiple-input multiple-output (MIMO) control system. For every task T_i in the system, its C_{Ai} value is chosen as the controlled variable while its actual execution time c_{ij} is chosen as the set point. The system error is defined as the difference between the controlled variable and the set point, *i.e.*,

$$\epsilon_i = c_{ij} - C_{Ai}. \tag{11}$$

The error is measured periodically by the PID controller. Its output is fed back to the feedback-DVS scheduler to adjust the value for C_{Ai}. For n tasks in the task set, there are altogether n feedback inputs (ϵ_i, i=1...n) and n system outputs (C_{Ai}, i=1...n). Let C_{Aij} be the estimated C_A value for the j^{th} job of a task T_i. For each task T_i, the following discrete PID control formula is used in our feedback-DVS scheduler:

$$\begin{array}{l} \Delta C_{Aij} = K_P * \epsilon_i + \frac{1}{K_I} \sum_{IW} \epsilon_i + K_D \frac{\epsilon_i - \epsilon_i(t-DW)}{DW} \\ C_{Ai(j+1)} = C_{Aij} + \Delta C_{Aij} \end{array} \tag{12}$$

where K_P, K_I and D_d are proportional, integral, and derivative parameters, respectively. ϵ_i is the monitored error. The output ΔC_{Aij} is fed back to the scheduler and is used to regulate the next anticipated value for C_{Ai}. IW and DW are tunable window sizes such that only the errors from the last IW (DW) task jobs will be considered in the integral (derivative) term. We use $DW = 1$ to limit the history, which ensures that multiple feedback corrections do not affect one another. The three control parameters K_P, K_I and K_D adjust the control response amplitude and its dynamic behavior with great versatility. It is therefore important to choose and tune these parameters for the controller. The process of adjusting the control parameters is a compromise among different system performance metrics. For example, the system may be tuned to have either a stable but slow control response, or an instable but dynamic control response. What is preferred in our system is a sufficiently rapid and stable control output during the entire scheduling process. Due to the MIMO property of the feedback system proposed above, multiple inputs need to be manipulated and multiple outputs need to be controlled simultaneously. The system behavior depends on interactions upon those multiple variables. It adds substantial complexity into the theoretical analysis and implementation of the feedback scheduling system. Given the difficulty of precisely characterizing the dynamic behavior of such MIMO control systems, we chose to tune those PID parameters by trial and error in our simulation experiments. As seen in Section 6, it gave us a satisfactory system performance as long as the parameters were accurately tuned. But such an approach usually requires a large amount of experiments before the final parameters can be determined.

In order to get around the complexity brought by the MIMO control system, we transform the above MIMO control problem into a single-input single-output (SISO) control model in the following and give an analysis to our system's stability.

4.1. A Simplified Control Design

We now present a simplified design for the system model. Instead of choosing $C_{Ai}(i = 1...n)$ as the controlled variable for each task T_i, we define a single variable r as the controlled variable for the entire system:

$$r = \frac{1}{n} \sum_{i=1}^{n} \frac{C_{Aij} - c_{ij}}{c_{ij}} \tag{13}$$

where j is the index of the latest job of task T_i before the sampling point. Our objective is to make r approximate 0 (*i.e.,* the set point). The system error becomes

$$\epsilon_i = r - 0. \tag{14}$$

ϵ_i is fed back to the PID scheduler to regulate the controlled variable r. The PID feedback controller is now defined as:

$$\begin{aligned} &\Delta r_j = K_P \epsilon_i + \tfrac{1}{K_I} \textstyle\sum_{IW} \epsilon_i + K_D \tfrac{\epsilon_i - \epsilon_i(t-DW)}{DW} \\ &r_{j+1} = r_j + \Delta r_j \end{aligned} \tag{15}$$

For each r_j, we adjust the C_A value for task T_i by $C_{Ai(j+1)} = r_j c_{ij} + c_{ij}$. The transfer function G_r between r and C_A can be derived by taking derivative of both sides of equation 13:

$$G_r(s) = Ms \tag{16}$$

where $M = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{c_i}$. The block diagram of the whole model is shown in Fig. 6. Its transfer function is

$$\frac{G_P(s)G_r(s)}{1 + G_P(s)G_r(s)} = \frac{MK_P s + MK_I + MK_D s^2}{1 + MK_P s + MK_I + MK_D s^2} \tag{17}$$

Such a transformation simplifies the control system so that there is only one system input ϵ_i and one system output r. It eases the analysis and implementation of the feedback controller in our scheduler. But a drawback of

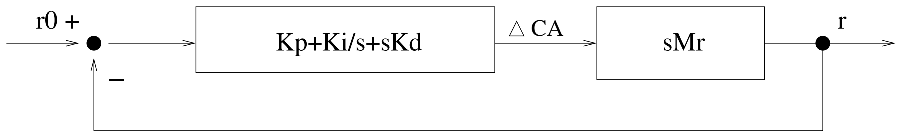


Figure 6. Control Loop Model

the model is that it does not provide direct feedback of the C_A value for each individual task. A zero value of r may not necessarily imply that every task's C_A has approximated its actual execution time. It is only an imprecise description of the original scheduling objective and may take longer to get the system into a stable status. But we expect that this model still captures the characteristics of the overall system behavior and leads to acceptable performance, which has been confirmed in our experiments. In the following, we analyze the system to assess the stability of our control model.

4.2. Stability Analysis

According to control theory, a system is stable if and only if all the poles of its transfer function are in the negative half-plane of the s-domain. From equation 17 we get the poles of our system as

$$\frac{-MK_P \pm \sqrt{MK_P^2 - 4MK_D(MK_I + 1)}}{2MK_D} \tag{18}$$

Note that $-MK_P + \sqrt{MK_P^2 - 4MK_D(MK_I + 1)}$ is still less than 0 when $MK_P^2 - 4MK_D(MK_I + 1) > 0$. All the poles are hence in the negative half-plan of the s-domain. Therefore, the stability of our system is ensured.

Due to the task splitting scheme, all tasks can still meet their deadline, even if the PID feedback controller does not adjust the C_{Aij} value close enough to c_{ij}. For example, if $C_{Aij} \le c_{ij}$, the task will enter its second portion and start T_B at the maximal frequency level. The feedback scheme, together with the task splitting scheme, guarantees the deadline requirements of real-time tasks.

5. Example

Combining all the techniques illustrated above, we now turn to a description of the entire algorithm. Our algorithm starts with an offline construction of the static maximal EDF schedule within the interval of the hyper-period. Figure 7(i) shows an example of such a maximal EDF schedule based on a task set from Pillai *et al.* (Pillai and Shin, 2001). The set consists of three tasks T_1={3,8}, T_2={3,10} and T_3={1,14}, where $T_i = \{C_i, P_i\}$ for $i = 1...3$, with worst-case execution times and periods C_i and P_i for task T_i, respectively. An idle task I={1,4} is also presented in the maximal schedule to fill underutilized processor time niches. Every task's actual execution time is 1 except the first job of T_1, which has an actual execution time of 2. All scheduling events (task release, preemption, resumption, and completion) of the maximal EDF schedule are stored in a look-up table to reduce time complexity. We assume that four relative frequency levels are available, namely, 25%, 50%, 75% and 100% of the maximal frequency.

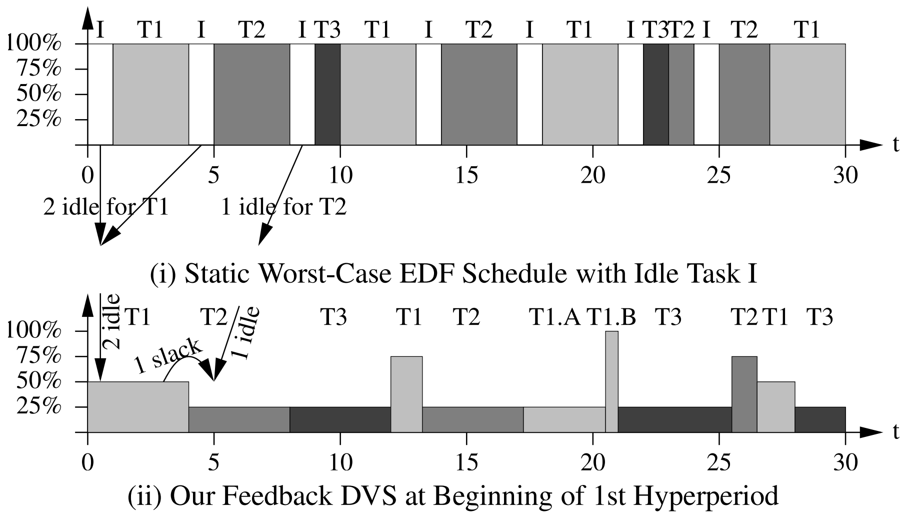


Figure 7. Discrete Scaling Levels for 3 Tasks

Next, the task set is scheduled according to our algorithm (without the idle task). As shown in Figure 7(ii), when $T_{1,0}$ (with a deadline of 8) is activated at time 0, no previous slack is passed on to $T_{1,0}$ because T_1 is the first task being scheduled. The value of $idle(0..d_1)$ is 2, which is obtained from the pre-calculated maximal EDF schedule. Hence the total slack $s_{1,0}$ available for $T_{1,0}$ is 2. $T_{1,0}$'s initial C_A is set to 1.5, which is 50% of its WCET. In fact, the initial C_A can be any values in (0,WCET). It does not need to be accurate enough because later one the feedback scheme will adjust C_A to make it approximate the task's actual execution time. Finally, a frequency scaling factor α is set according to Equation 3: $\alpha = C_A/(C_A + s_{1,0}) = 1.5/(1.5+2) = 0.428$. The CPU frequency is then set to 50% of the maximal frequency, which is the closest available level to 0.428. Since $T_{1,0}$'s actual execution time is only 2, it completes at time 4 and passed on one unit of unused slack to the next task $T_{2,0}$ according to Equations 7 and 8. Its C_A is also updated at that time according to the feedback scheme. $T_{2,0}$'s slack is again determined by Equation 6. Besides of the slack from the idle task, $T_{2,0}$ also gets one unit of slack from $T_{1,0}$. Its frequency level is determined in

```
Procedure Initialization
    for each $T_k \in \{T_1, T_2, \ldots, T_n\}$ do
        $C_{Ak} \leftarrow C_k/2$
        $left_{k0} \leftarrow C_k$
        $t_i \leftarrow 0$
    $U \leftarrow \frac{C_1}{P_1} + \frac{C_2}{P_2} + \ldots + \frac{C_n}{P_n}$
    $P_{n+1} \leftarrow P_1$
    $C_{n+1} \leftarrow P_1 \times (1 - U)$
    $c_{n+1} \leftarrow 0$
    $slack \leftarrow 0$

Procedure TaskCompletion($\mathbf{T_{ij}}$)
    $slack \leftarrow slack - c_{ij} + C_i$
    $\epsilon \leftarrow c_{ij} - C_{Aij}$
    $\Delta C_{Aij} \leftarrow K_P * \epsilon(t_i)+$
        $\frac{1}{K_I} \sum_{IW} \epsilon(t_i)+$
        $K_D \frac{\epsilon(t_i) - \epsilon(t_i - DW)}{DW}$
    $C_{Ai(j+1)} = C_{Aij} + \Delta C_{Aij}$
    $t_i \leftarrow t_i + 1$
    $left_{i(j+1)} = C_i$
    if $reserve_{ij} > 0$ then
        release $idle(now..d_{ij})+$
            $completed(now..d_{ij})$
                up to $|reserve_{ij}|$
Procedure SetInterrupt($\mathbf{T_{ij}}, \mathbf{C_A}$)
        Set timer interrupt for $T_{ij}$,
        triggered $C_A$ time units later
Procedure SetFrequency($\alpha$)
        $f \leftarrow \alpha \times f_m$

Procedure TaskActivation($\mathbf{T_{ij}}$)
    if processor was idle for $t$ then
        $slack \leftarrow slack - t$
    if $T_{pk}$ preempted/interrupted then
        $left_{pk} = C_p - c_{pk} \times \alpha$
        $slack \leftarrow slack - idle(d_{ij}..d_{pk})$
        if $left_{pk} > slots(T_{pk}, now..d_{pk})$
        then
            $reserve_{pk} \leftarrow left_{pk} -$
                $slots(T_{pk}, now..d_{pk})$
            allocate $reserve_{pk}$ in
                $idle(now..d_{pk})+$
                    $completed(now..d_{pk})$
            $slack \leftarrow slack - reserve_{pk}$
    else ($T_{pk}$ completed execution)
        if $now > d_{pk}$ then
            $slack \leftarrow slack-$
                $idle(d_{pk}, now)$
        $slack \leftarrow slack + idle(d_{pk}..dij)$
    $\alpha\prime \leftarrow \min\{\frac{f_1}{f_m}, \ldots, \frac{f_m}{f_m} |$
            $\frac{f_i}{f_m} \geq \frac{C_{Aij}}{C_{Aij} + slack}\}$
    if $(\alpha = 1)$ then
        $C_A \leftarrow 0$
    else
        $C_A \leftarrow slack \times \alpha/(1 - \alpha)$
    SetInterrupt($T_{ij}, C_A/\alpha$)
    SetFrequency($\alpha$)
```

Figure 8. Pseudocode of Feedback DVS Scheme

a similar way as the first task. For later task instances, the feedback scheme chooses C_A to approximate the task's actual execution time. Hence, the entire task is scaled at a low frequency level. Preemption handling, as described in Section 3.4, is also applied but not shown here to simplify the example.

An algorithmic description of our feedback-DVS scheme integrated with the PID feedback control is given in Figure 8. This algorithm is a refinement of our previous work (Dudani et al., 2002) and integrates the PID feedback scheme and preemption handling with future slot reservation. Only the MIMO control model is presented in the pseudocode, because the SISO model can be implemented in a similar way.

The following notations are used in our algorithmic description:

– T_{ij}: the j-th job of task T_i

- ij, pk : indices for the current and previous tasks relative to T_{ij}
- now: the current time
- P_i: the Period of T_i
- d_{ij}: the deadline of T_{ij}
- C_i: the WCET of T_i (without scaling)
- c_{ij}: the actual execution time of T_{ij} up to now (with scaling)
- K_P, K_I, K_D: the PID parameters
- IW, DW: the integral and derivative window size
- $left_{ij}$: the worst case remaining execution time of T_{ij} (without scaling)
- $slack$: system current slack
- $idle(t1..t2)$: the amount of idle slots between times [t1,t2]
- $completed(t1..t2)$: slots of already completed tasks between times [t1,t2]
- $slots(T_{ij}, t1..t2)$: the amount of time slots reserved for T_{ij} in the worst case between times [t1,t2]
- f: the processor frequency
- f_m: the maximal processor frequency
- α: the frequency scaling factor
- C_A: the execution time of the T_A subtask

The effect of the PID feedback scheme is shown in the following example. Consider a task set of three tasks T_1=$\{$12,32$\}$, T_2=$\{$12,40$\}$ and T_3=$\{$4,65$\}$. Let the actual execution times of different jobs of a task fluctuate according to the execution time pattern 1, as depicted in Figure 11. Figure 9(a) is a snapshot of the feedback-DVS schedule for this task set without PID-feedback. Figure 9(b) depicts the feedback-DVS schedule for the same task set using feedback with PID parameters K_P=0.9, K_I=0.08 and K_D=0.1.

We can see from the figures that the first job of Tx_3 and the second job of T_2 are scheduled to run at a much lower frequency in the PID feedback schedule than the one without PID-feedback. The first job of T_3 with an actual execution time of 2.57 starts at time 524 in the schedule without PID-feedback, and starts at time 520 in the PID feedback schedule. The PID feedback scheme gets an execution time of 3.06 for its C_A according to Equation 4. With the closer approximation of c_{ij}, the PID scheduler is able to scale the task more aggressively than the one without PID-feedback. Similarly, the non-feedback schedule only gets an average execution time of 5.26 for the second job of T_2, which has an actual execution time of 7.07. But the PID feedback scheme obtains a $C_A = 6.76$, which is again closer to T_2's actual execution time. This demonstrates the superiority of our feedback-DVS scheme in adapting to dynamic workloads resulting in additional energy savings.

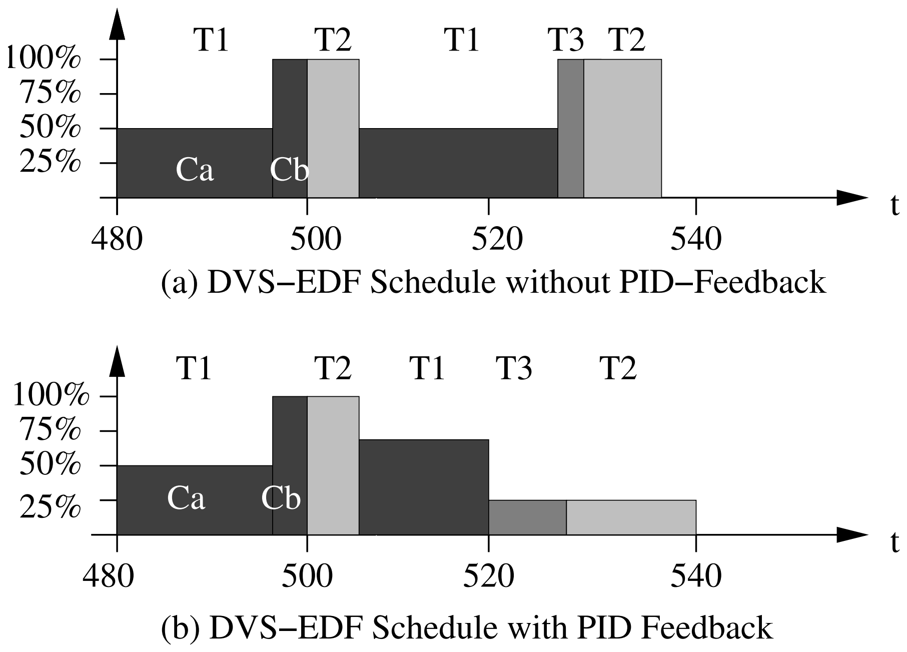


Figure 9. Schedules: Simple and PID Feedback

In traditional EDF scheduling, any job's actual start time s_i is less than or equal to its worst-case start time in the maximal schedule. But this is no longer the case in our feedback-DVS schedule. Because feedback-DVS may scale a job's execution time to be larger than its WCET value, a job's actual start time may be later than its start time in the maximal schedule. The next example shows a case where a job's actual start time exceeds its worst-case start time.

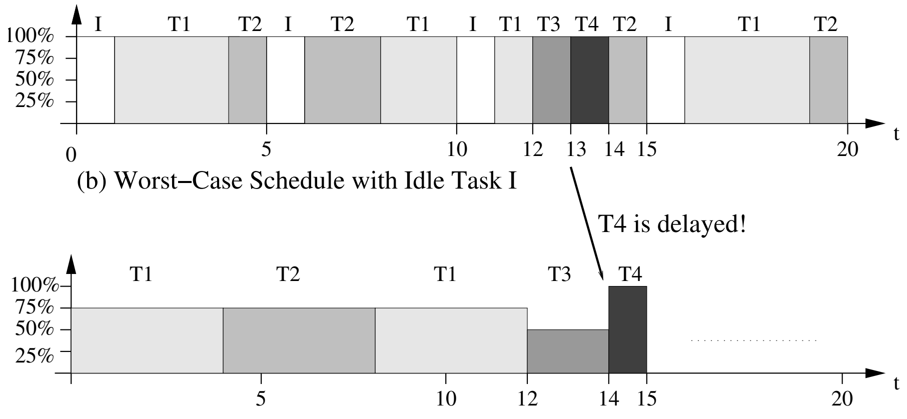
Consider the task set in Figure 10(a). Its worst-case schedule with an idle task and its actual schedule under feedback-DVS are shown in Figure 10(b) and Figure 10(c), respectively. When task T_3's second job starts at time 12 in the actual schedule, its absolute deadline is at time 18. There is only one idle slot between time 12 and time 18, which can be used to scale T_3 at a 50% frequency level. Since T_3's actual execution time equals its worst-case execution time, it runs for 2 time units and ends at time 14 with an actual execution time of 2. When T_4 starts execution at time 14, it has been delayed by one time unit relative to its start time in the worst-case schedule.

We show the correctness of our feedback-DVS algorithm, by the following theorem.

THEOREM 1 (Correctness). *The feedback-DVS algorithm results in a feasible schedule for a set T of tasks with periods equal to their relative deadlines if a feasible schedule exists for T under preemptive EDF.*

Task T_i	WCET C_i	Period P_i	c_i	r_i
1	3 ms	8 ms	3 ms	0 ms
2	3 ms	10 ms	3 ms	4 ms
3	1 ms	14 ms	1 ms	4 ms
4	1 ms	20 ms	1 ms	0 ms
idle	1 ms	5 ms	1 ms	0 ms

(a) Task Set



(c)Actual DVS−EDF Schedule

Figure 10. Delayed Start of Tasks due to Scaling

A detailed proof is presented in the Appendix.

6. Experiments

We evaluated the performance of our schemes in a simulation environment that supports feedback-DVS scheduling. In order to make a comparison with our algorithm, Pillai and Shin's Look-ahead RT-DVS algorithm was also implemented (Pillai and Shin, 2001). We assume a processor model capable of operating at four different voltage and frequency levels, as depicted in Table I. Comparable frequency and voltage setting were also used in the Look-ahead RT-DVS work (Pillai and Shin, 2001) and the experimental work with StrongARM processors (Pouwelse et al., 2000). The results discussed hereafter are also consistent in their trends for savings with a concrete DVS-capable architecture, albeit the details of these experiments are beyond the scope of the paper (Zhu and Mueller, 2004b). In our simulations, the processor enters an idle state and operates at the lowest frequency and voltage level when no tasks are ready. We use a simplified energy model in our experiment as

$E = \int_0^t fV^2$. Energy values reported in the following experiments were normalized for ease of comparison.

Table I. Processor Model for Scaling

frequency	voltage
25%	2 V
50%	3 V
75%	4 V
100%	5 V

Altogether, 50 task sets were generated, each consisting of either 3 or 10 tasks. In our experiments, we first investigated the performance of our scheme over fluctuating workload patterns. The objective in studying different patterns is to assess the sensitivity of feedback DVS to different types of execution time fluctuations, which have been observed in interrupt-driven systems (Mächtel and Rzehak, 1996). Since it is not practical to examine every possible type of fluctuation, we constructed three synthesized execution time patterns based on our observation of some typical real-time applications, as shown in Figure 11.

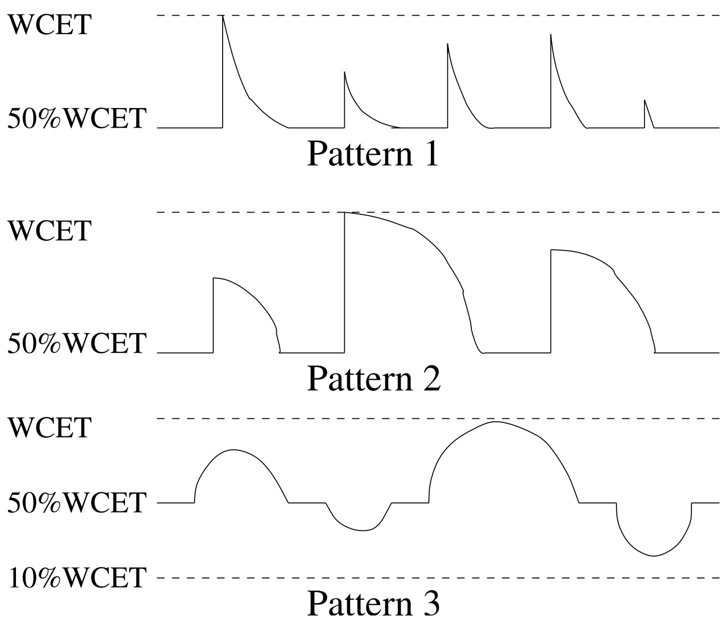


Figure 11. Task Actual Execution Time Pattern

In the first pattern, the actual execution time of a job starts at 50% of the task's WCET before spiking to a peak value c_m every 10th job. The peak value c_m is randomly generated for each spike from a uniform distribution between 50% of WCET and 100% of WCET. After the peak value is reached, the actual execution time of the following jobs drops exponentially (modeled as $c_i = 1/2^{(t-c_m)}$) until it reaches 50% of the WCET again. This pattern simulates event-triggered activities that result in sudden, yet short-term computational demands due to complex inputs often observed in interrupt-driven systems. In the second execution time pattern, the peak execution time c_m still follows a random uniform distribution between 50% of WCET and 100% of WCET. But the actual execution time of the following jobs initially drops more gradually, modeled as $c_i = c_m sin(t + \pi/2)$. This pattern simulates events resulting in computational demands in a phase of subsequent complex inputs (with a decaying tendency). In the third execution pattern, the actual execution time of the jobs alternates between positive and negative peaks every 10 jobs. Both the peak values in either direction are randomly generated from a uniform distribution between 50% of WCET and 100% of WCET. The actual execution time of the jobs following the peak value is modeled as $c_i = c_m sin(t)$ and $c_i = -c_m sin(t)$. This pattern represents periodically fluctuating activities with gradually increasing and decreasing computational needs around peaks. For each execution time pattern, the task sets' WCETs were uniformly distributed in the range [10,1000]. When tasks' WCETs were generated, each task's period was chosen so that the worst case utilization of the task set (*i.e.*, $\sum \frac{WCET_i}{P_i}$) varies from 0.1 to 1.0 in increments of 0.1.

Both of the original MIMO feedback control model and the simplified SISO feedback control model were evaluated in our experiment. The corresponding feedback-DVS schedulers are referred to as MF-DVS and SF-DVS, respectively. Different combinations of PID coefficients were investigated in our experiments. It was observed that both increasing or decreasing the proportional coefficient resulted in less accurate system estimations for C_A. The derivative item is less significant compared to the other two parameters. Increasing the integral window size improves the energy saving effect in the very beginning, but when IW becomes larger than 10, no dramatic system performance improvements were observed. We restrict ourselves here to report results based on the PID coefficients of $K_P = 0.9$, $K_I = 0.08$, $K_D = 0.1$. The derivative and integral window size were 1 and 10, respectively.

Figure 12 compares the energy consumption between our feedback-DVS scheme and the Look-ahead RT-DVS scheme under the execution time pattern 1. When the task set utilization is less than 0.3, it is observed that all schemes consume the same amount of energy. This is because task sets with low utilization usually have enough slack and idle slots, so that all jobs are able to be scaled to the lowest speed level. In this case the processor always operates at the 25% frequency level and consumes the same amount of energy

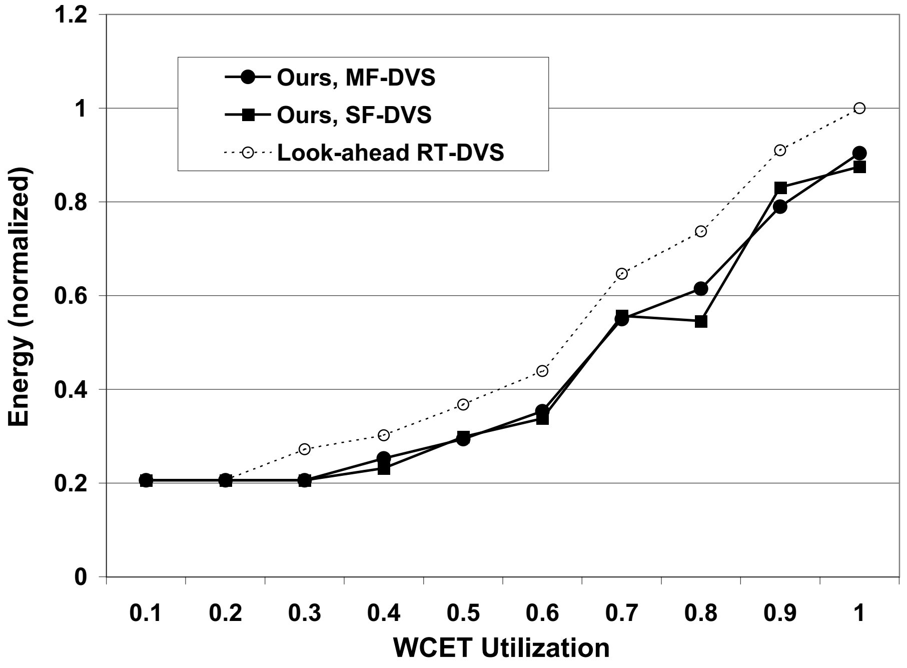


Figure 12. Execution Time Pattern 1

for all schemes. With the increase of the worst-case utilization, our feedback-DVS scheme started saving more energy than Look-ahead RT-DVS. MF-DVS adapts to the changing workload better than Look-ahead RT-DVS and costs 8% to 24% less energy than it. The maximal energy savings (24%) can be observed at 80% utilization. SF-DVS works almost as well as MF-DVS, which shows that the simplified model still captures the dynamic system behavior and adapts to the changing workload efficiently. Similar results can be observed for execution time pattern 2 and 3, as depicted in Figures 13 and 14. The maximal energy savings of MF-DFS, 22% and 16%, are at 0.5 and 0.9 utilizations, respectively. Its average energy saving over Look-ahead RT-DVS is around 15%. SF-DVS costs a little more energy than MF-DVS in some cases because SF-DVS usually takes longer to respond to tasks' execution time variations than MF-DVS does. But overall, SF-DVS still saves up to 20% and 19% energy over Look-ahead under pattern 2 and pattern 3, respectively. These experiments show that our feedback-DVS scheme is not sensitive to these three execution time patterns.

In order to further observe the scalability of our algorithm, we generated three task sets following execution time pattern 1, but with different baseline values. While the pattern depicted in Figure 11 has a 50% WCET baseline, the other two task sets have baselines of 75% and 25% WCET, respectively.

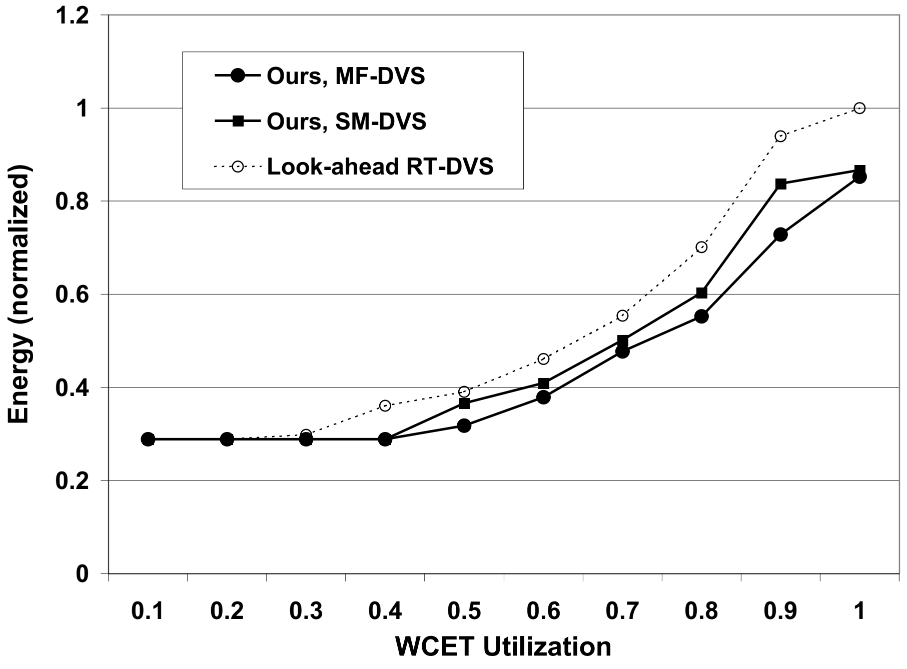


Figure 13. Execution Time Pattern 2

Shifting the baseline among different task sets also results in a change of their actual utilization. Figure 15 and Figure 16 compare the energy consumption between our feedback-DVS and Look-ahead RT-DVS for these three task sets. The energy values are normalized to the maximal point of the 75% WCET baseline task set. The result shows that our scheme is able to scale to task sets with different baselines very well. MF-DVS saved up to 20% more energy than Look-ahead RT-DVS for the baseline of 75% WCET case. When the baseline is 25% of the WCET, up to 29% more energy savings are observed. The maximal energy saving appears in the task set with 25% WCET baseline since it provides the largest range for execution time fluctuation. Similar results can also be observed for SF-DVS. A maximum energy saving of 26% over Look-ahead are observed at the 50% WCET baseline case. Both our schemes, MF-DVS and SF-DVS, are able to adapt to workloads with baseline variations.

Figure 17 illustrates the performance of our feedback-DVS scheme by varying the number of tasks in the task sets. We compared the energy consumption between our algorithm and Look-ahead RT-DVS for task sets with 10 and 3 tasks. All energy values are normalized to the maximal point of Look-ahead RT-DVS in the 10-task set case. We notice that there is little effect of varying the number of tasks on our scheme. Both MF-DVS and SF-

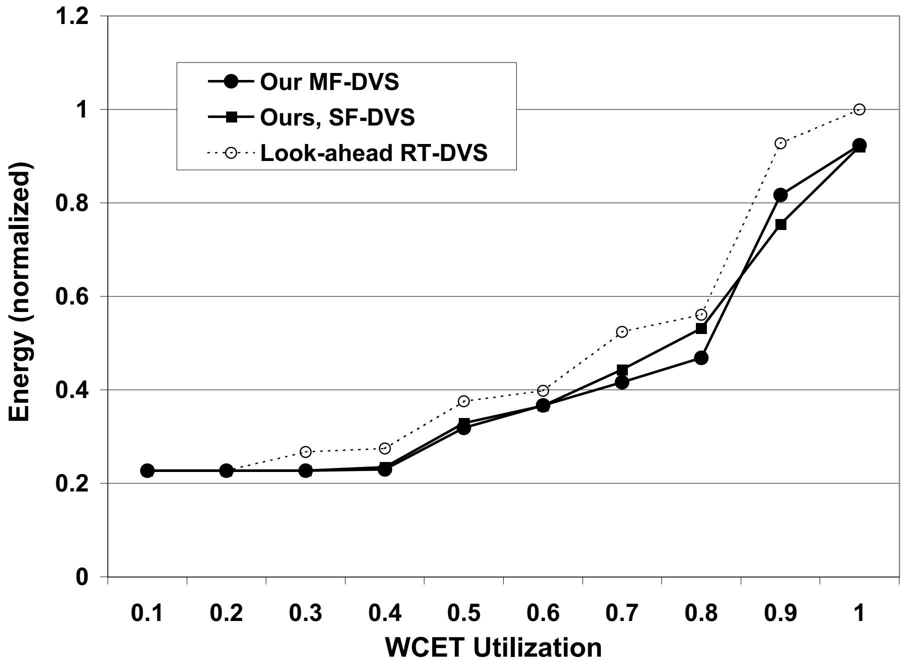


Figure 14. Execution Time Pattern 3

DVS are able to save roughly the same percentage of energy over Look-ahead RT-DVS between 10-task sets and 3-task sets. However, a larger number of tasks tends to result in lower overall energy consumption.

Besides the execution time patterns listed in Figure 11, we also investigated task sets with random execution characteristics, *i.e.*, tasks' actual execution times are derived from a random uniform distribution. We performed this experiment in order to assess the worst-case behavior of our algorithm for task sets with highly fluctuating execution time patterns. Our feedback-DVS scheme resulted in similar energy savings as Look-ahead DVS. Random execution times do not give additional benefits to our algorithm because the algorithm cannot supply any useful history information to the feedback controller. This is a limitation of feedback schemes in general. Nonetheless, even in this worst case, our feedback-DVS algorithm behaves no worse than Look-ahead DVS.

Overall, our Feedback feedback-DVS algorithm is able to exhibit considerable energy savings for different task sets. The simplified feedback control model, SF-DVS, captures the characteristics of the overall system behavior and leads to acceptable performance comparable with MF-DVS. Feedback control in conjunction with DVS scheduling makes the system more adaptive

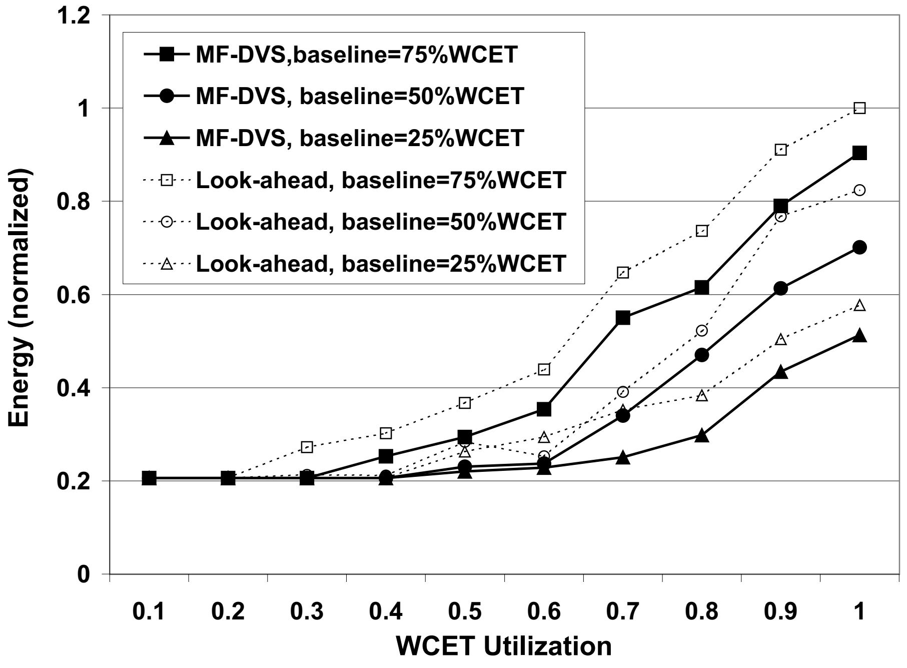


Figure 15. MF-DVS, Varying Baseline

to dynamically changing workloads, and achieves lower energy consumption levels than other less adaptive schemes.

7. Related Work

There have been a number of efforts of applying feedback techniques on general-purpose control systems. But only recently did researchers begin to incorporate feedback control to real-time scheduler with timing constraints (Lu et al., 1999; Lu et al., 2002a). Lu *at al.* proposed a feedback control real-time scheduling framework for unpredictable dynamic real-time systems where task execution times diverge from their worst case (Lu et al., 2002a). Real-time system performance specifications are analyzed and satisfied systematically through a control theory-based methodology. Dynamic models of real-time systems are developed to identify different categories of real-time applications with different feedback control algorithms. While their feedback control framework is mainly used to satisfy general purpose real-time system requirements, our scheme focuses on exploiting feedback control schemes to reduce energy consumption.

Our work is more closely related to the ones in (Lu et al., 2002b) and (Minerick et al., 2002). Lu *et al.* describe a formal feedback control algo-

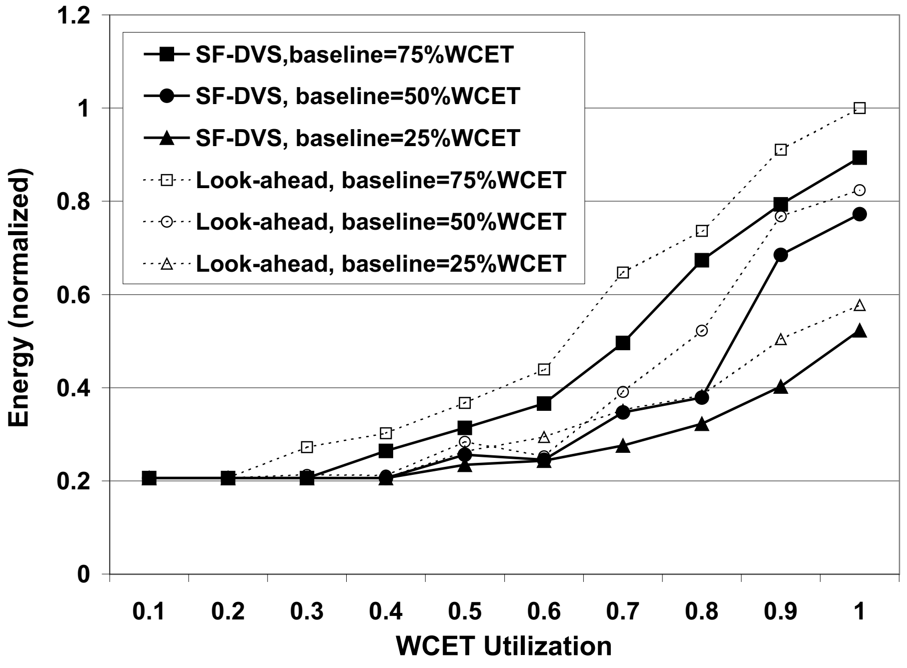


Figure 16. SF-DVS, Varying Baseline

rithm combined with dynamic voltage/frequency scaling technologies for multimedia systems (Lu et al., 2002b). Both continuous and discrete DVS settings are exploited in a scheme to reduce energy consumption while still guaranteeing real-time requirements. An adaptive set-point is used to achieve fast responses with a stable multimedia throughput. Both their work and our approach exploit feedback control to DVS/DFS technologies. They target soft real-time/multimedia systems, while we focus on hard real-time systems where timing constraints must not be violated.

A general energy management scheme with feedback control was proposed by Minerick *at al.* (Minerick et al., 2002). An average energy usage is achieved by continuously adjusting the voltage/frequency of a processor to meet the energy consumption goal. A PI (proportional and integral) feedback controller is used to adapt the proper power setting based on previous energy consumptions without the prediction of future system workloads. While their objective is to obtain low energy consumption for general purpose systems, we target hard real-time systems with deadline requirements.

Dynamic voltage scaling has been studied by many previous researchers. Saewong *et al.* (Saewong and Rajkumar, 2003) proposed a series of voltage scaling schemes targeting different hardware configurations and task set characteristics. Their results showed that some non-optimal schemes may be more

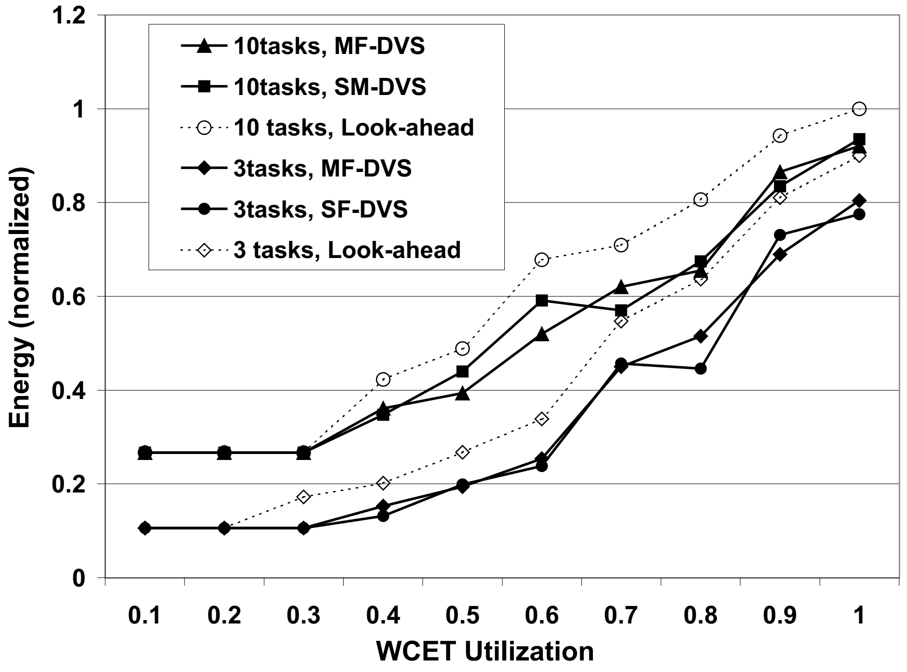


Figure 17. 10-task vs. 3-task under Pattern 1

suitable than optimal schemes when the system has a high voltage scaling overhead. Lee *et al.* (Lee and Krishna, 2003) presented a branch-and-bound algorithm to statically determine the operating frequency of real-time task sets. But due to the complexity of the algorithm, only two frequency levels are assumed in their model. Ishihara *et al.* (Ishihara and Yasuura, 1998) proved that on a processor with a small number of discretely variable voltages the energy consumption is minimized when the voltage scheduling contains at most two voltage levels. But the two-speed schedule is only optimal when a task's actual execution time can be determined statically, which is generally not possible. When actual execution time varies from instance to instance, a multiple-speed schedule can result in lower energy than a dual-speed schedule. The algorithm proposed in (Liu and Mok, 2003) derives optimal speed functions between an upper bound and a lower bound of processor cycles. Their online algorithm reclaims unused execution cycles to further reduce energy consumption. The algorithms in (Pillai and Shin, 2001; Aydin et al., 2001; Gruian, 2001) are more closely related to ours. Pillai and Shin (Pillai and Shin, 2001) proposed a set of dynamic DVS algorithms based on traditional hard real-time mechanisms, namely rate-monotone (RM) scheduling and EDF scheduling. They extended the schedulability test of RM and EDF algorithms to incorporate CPU frequency scaling. Unlike our algorithm that applies frequency scaling to only the current task, they assumed a unified frequency scaling factor upon all tasks. In their most aggressive variant, the look-ahead technique is used to achieve extensive energy savings by deferring as much work as possible. However, the frequency value obtained in their algorithm is not always the lowest possible frequency for a single task, as shown in (Dudani et al., 2002).

Some of the other aggressive real-time DVS schemes exploit early completion of task executions based on statistical information of the workload under dynamic scheduling (Aydin et al., 2001) or static priority scheduling (Gruian, 2001). The algorithm proposed in (Aydin et al., 2001) was based on early completion of tasks and idle time up to the next task's activation. The feedback scheme in our algorithm adapts even to dynamically changing execution demands, not just statistical information. We exploit both the idle time prior to the next task's activation and any idle slots up to the deadline of the task in the maximal schedule. We show in other work that Aydin's algorithm only outperforms Pillai's approach for large task sets with extremely low or high utilizations (Zhu and Mueller, 2004b).

The idea of deriving a feasible dual-level DVS schedule from an ideal case was first proposed by Gruian (Gruian, 2001; Gruian and Kuchcinski, 2001). It combines off-line and on-line scheduling at both task level and task-set level. Stochastic data was used to derive energy-efficient schedules. Multiple frequency levels may be assigned to a single task. In our approach, we assign at most two different frequencies for each task, and the highest frequency is always assigned to the second subtask. A single frequency, as dominant in our algorithm, results in lower energy consumption than multiple frequencies as advocated by Gruian. Our algorithm also targets dynamic scheduling (EDF) while Gruian restricts his approach to fixed-priority static scheduling. Dual speed scheduling was also proposed in two other approaches. First, Zhang *et al.* switch the processor speed between high and low whenever non-preemption blocking occurs among tasks that share resources (Zhang and Chanson, 2002). Second, Lee *et al.* assume an architecture model where only two physical speed levels exist (Lee and Krishna, 2003). Our approach considers a more general case where multiple frequency and voltages levels are chosen by subsequent jobs of the same task or even different tasks, although for a single job, only two speeds are used. Last-chance scheduling without energy considerations goes back at least to Chetto and Chetto (Chetto and Chetto, 1989). We apply this philosophy in a DVS context. We develop a novel variant based on task splitting with exactly two parts. Such a dual-subtask approach aggressively reduces power consumption if the first subtask is fully utilized while the second subtask never executes. Our feedback approach triggers this behavior, which is superior to Gruian's step-wise increase of frequencies using stochastic approach.

8. Conclusion

This paper presents a novel scheduling approach combining DVS with feedback control schemes, which extends EDF in a most aggressive manner. The technique relies strictly on operating system support to implement both the real-time scheduler and the feedback controller. Our contributions include techniques for slack exploiting, preemption handling and feedback control schemes for hard real-time systems with dynamically fluctuating workload characteristics, where execution times of a periodic task vary significantly. Early scaling at a low frequency capitalizes on the probability of a task behavior when it completes execution without exhausting its worst-case execution budget. A last-chance approach is used if a task does not complete at a certain point in time with a low frequency. The remainder of the task continues at a higher frequency to ensure its deadline requirement. A feedback scheme is applied on the system to make it capable of selecting the right frequency and voltage settings for the first potion, as well as guaranteeing hard real-time requirements for the overall task. For predictable fluctuating execution time patterns, our feedback DVS scheme is able to adapt to dynamically fluctuating workloads better than previous work and saves up to 29% additional energy. The scheme is not sensitive to particular workload characteristics, *i.e.*, the execution time patterns, and is capable of scaling for task sets with different number of tasks.

Acknowledgments

We want to especially thank Ajay Dudani for his contribution on our early work of the Feedback-DVS scheme (Dudani et al., 2002).

Appendix

A. Correctness

Theorem *The feedback-DVS algorithm results in a feasible schedule for a set T of tasks with periods equal to their relative deadlines if a feasible schedule exists for T under EDF.*

We call the schedule produced by our feedback-DVS algorithm the *actual* schedule, where the execution time of a task is variable for different task instances (jobs). We call the schedule under EDF where each task's actual execution time always equals its WCET the *maximal* schedule. Let s_i and s_i^+ be T_i's absolute start times in the actual and the maximal schedule, respectively. (We use the simplified shortcut T_i to denote a certain j^{th} job T_{ij} of task T_i). Similarly, let f_i and f_i^+ be the absolute completion times of T_i in the actual and the maximal schedule, respectively. In order to prove the theorem, we first prove the following lemma:

LEMMA 1. *The difference between a task's start time in the actual schedule and the maximal schedule is bounded in feedback-DVS by the following inequation:*

$$s_i - s_i^+ \leq idle(f_{i-1}^+, d_{i-1}) + \sum_{T_l \in [f_{i-1}^+, d_{i-1}]; d_l > d_i} C_l \tag{19}$$

where $idle(f_{i-1}^+, d_{i-1})$ is the length of all idle slots existing between $[f_{i-1}^+, d_{i-1}]$ in the maximal schedule. C_l is the WCET of any task T_l in the maximal schedule with a priority lower than T_i. f_{i-1}^+ and d_{i-1} are the completion time and absolute deadline of task T_{i-1}, which is the most recently executed task before T_i.

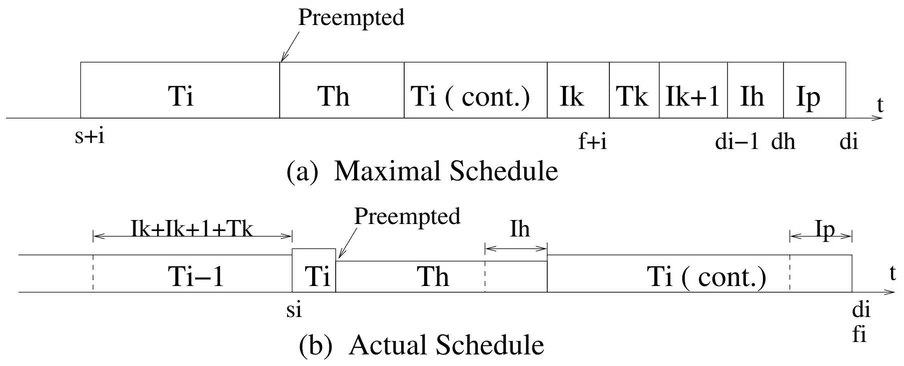


Figure 18. Maximal vs. Actual Schedule

Proof:

We will use induction to prove the lemma. First, consider the highest priority task T_1 as the base case. Since T_1 always starts execution immediately at its release time under both the actual schedule and the maximal schedule, we have,

$$s_1 - s_1^+ = 0. \tag{20}$$

Hence, the lemma holds for T_1.

Now assume that a certain task T_i satisfies the lemma. We need to show that T_{i+1}, the task with the next lower priority than T_i, also satisfies the lemma. We only need to consider the case where $s_{i+1} > s_{i+1}^+$, since this is where feedback-DVS diverges from conventional EDF. The only reason for T_{i+1} to be delayed is that some higher priority tasks are still running at time s_{i+1}^+. Without loss of generality, we assume that in the maximal schedule there are m ($m \geq 0$) idle slots and q ($q \geq 0$) lower priority tasks in $[f_{i-1}^+, d_{i-1}]$, namely, I_k, I_{k+1},...,I_{k+m-1} and T_k, T_{k+1},...,T_{k+q-1}. Their

WCETs are denoted by I_k, I_{k+1},...,I_{k+m-1} and C_k, C_{k+1},...,C_{k+q-1}, respectively. We have $\sum_{l=k}^{k+m-1} I_l = idle(f^+_{i-1}, d_{i-1})$. Let $I_h = idle(d_{i-1}, d_h)$ and $I_p = idle(d_h, d_i)$. It is also possible that T_i be preempted by a certain higher priority task T_h during its execution. Figure 18 shows a simplified case where only I_k,I_{k+1} and T_k are shown before d_{i-1}. Since both T_{i-1} and T_h have priorities higher than T_i, we have $d_i \geq d_{i-1}$ and $d_i \geq d_h$. We note that at the time s^+_i in the maximal schedule, all other tasks with priorities higher than T_i must have completed, and all other lower priority tasks will not be scheduled before f^+_i. Only newly released high priority tasks can execute in [s^+_i, f^+_i] and may preempt T_i. Since the lemma holds for T_i, we have :

$$s_i - s^+_i \leq \sum_{l=k}^{k+m-1} I_l + \sum_{l=k}^{k+q-1} C_l = idle(f^+_{i-1}, d_{i-1}) + \sum_{T_l \in [f^+_{i-1}, d_{i-1}]; d_l > d_i} C_l \quad (21)$$

Our feedback-DVS scheme moves I_k, I_{k+1},...,I_{k+m-1} and T_k backward to s^+_i, and moves the corresponding portion of T_i forward. These transformations are legal since T_i still resides within $[r_i, d_i]$. The high priority task T_h is left untouched, because it can always preempt T_i at s^+_h in the actual case, *i.e.*, $s_h = s^+_h$. When T_i is preempted at time s_h, the forward slack reservation scheme in feedback-DVS reserves $C_i - (s_h - s_i)$, the worst-case remaining execution time left for T_i, from T_k, I_{k+m-1},...forward. The backward slack reservation scheme reserves the above amount of time from the I_p, I_h,...,backward. In either case, we denote the total execution time of reserved slots by C_R. At time s^+_h, the frequency scaling decision is made for T_h. The scheduler collects all available idle slots and early completion of low priority task slots in $[s^+_h, d_h]$ in the maximal schedule excluding any slots reserved for future resumption of preempted tasks. The final amount of slack available for T_h equals to $\sum_{i=k+1}^{k+m-1} I_i + I_h + \sum_{l=k}^{k+q-1} C_l - C_R$. T_h uses the slack to scale itself to a lower frequency and voltage level. It is equivalent to the transformations that move the non-reserved portion of I_{k+1},...,I_{k+m-1},I_h and T_l backward and move the corresponding portion of T_i forward. The result is shown in Figure 18(b). When T_i resumes execution, it can be scaled again exploiting slack from the idle slots and early-completed task slots before d_i. Similar transformations apply when moving I_p backward and T_i forward. T_i releases all its unused slack when it completes and passes it on to following tasks.

Except for the idle slots and early completion of lower priority tasks, there are no other cases where T_i will be moved forward and thus be delayed during the above transformations. Hence, the following inequation holds:

$$f_i - f^+_i \leq idle(f^+_i, d_i) - C_R + \sum_{T_l \in [f^+_i, d_i];\ d_l > d_{i+1}} C_l \quad (22)$$

Because $d_i \geq d_{i-1}$ and $d_i \geq d_h$, the aforementioned transformations never move T_i forward beyond d_i. Hence, T_i will not miss its deadline after these transformations. If the start time of T_{i+1} is delayed in the actual schedule by T_i, we have: $s_{i+1} = f_i$ and $s^+_{i+1} \geq f^+_i$. From the above equation we get:

$$s_{i+1} - s^+_{i+1} \leq f_i - f^+_i \leq idle(f^+_i, d_i) + \sum_{T_l \in [f^+_i, d_i];\ d_l > d_{i+1}} C_l \quad (23)$$

Hence, inequation 19 also holds for T_{i+1}, and we proved the lemma.

This lemma, in fact, describes a worst-case scenario. It shows that no matter how aggressively previous tasks T_1, T_2,...T_i are scaled, the start time of the next task T_{i+1} will not be delayed for more than the interval of $idle(f^+_i, d_i) + \sum_{T_l \in [f^+_i, d_i];\ d_l > d_{i+1}} C_l$. In such a worst case scenario, the feedback-DVS scheduler will always set T_{i+1}'s speed to maximal so that T_{i+1}'s actual execution time will not exceed C_{i+1}. Since in the maximal schedule we always have:

$$s^+_{i+1} + C_{i+1} + idle(f^+_i, d_i) + \sum_{T_l \in [f^+_i, d_i];\ d_l > d_{i+1}} C_l \leq d_{i+1} \quad (24)$$

using inequation 23 and 24, we get:

$$s_{i+1} + C_{i+1} \leq s^+_{i+1} + C_{i+1} + idle(f^+_i, d_i) + \sum_{T_l \in [f^+_i, d_i];\ d_l > d_{i+1}} C_l \leq d_{i+1} \quad (25)$$

which shows that T_{i+1} meets its deadline. Thus, our feedback-DVS always results in a feasible schedule. The theorem is proved.

References

Arnold, R., F. Mueller, D. B. Whalley, and M. Harmon: 1994, ‘Bounding Worst-Case Instruction Cache Performance’. In: *IEEE Real-Time Systems Symposium*. pp. 172–181.

Aydin, H., R. Melhem, D. Mosse, and P. Mejia-Alvarez: 2001, ‘Dynamic and Agressive Scheduling Techniques for Power-Aware Real-Time Systems’. In: *IEEE Real-Time Systems Symposium*.

Chandrakasan, A., S. Sheng, and R. W. Brodersen: April, 1992, ‘Low-power CMOS digital design’. In: *IEEE Journal of Solid-State Circuits, Vol. 27, pp. 473-484.*

Chetto, H. and M. Chetto: 1989, ‘Some Results of the Earliest Deadline Scheduling Algorithm’. *IEEE Transactions on Software Engineering* **15**(10), 1261–1269.

D. Shin, J. K. and S. Lee: 2001, ‘Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications’. In: *IEEE Design and Test of Computers*.

Dudani, A., F. Mueller, and Y. Zhu: 2002, ‘Energy-Conserving Feedback EDF Scheduling for Embedded Systems with Real-Time Constraints’. In: *ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES’02) and Software and Compilers for Embedded Systems (SCOPES’02)*. pp. 213–222.

Ferdinand, C., F. Martin, and R. Wilhelm: 1997, ‘Applying Compiler Techiniques to Cache Behavior Prediction’. In: *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*. pp. 37–46.

Govil, K., E. Chan, and H. Wasserman: 1995, ‘Comparing algorithms for dynamic speed-setting of a low-power CPU’. In: *1st Int’l Conference on Mobile Computing and Networking*.

Gruian, F.: 2001, 'Hard real-time scheduling for low energy using stochastic data and DVS processors'. In: *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*.

Gruian, F. and Kuchcinski: 2001, 'LEneS: task scheduling for low-energy systems using variable voltage processors'. In: *Proceedings of ASP-DAC*.

Grunwald, D., P. Levis, C. M. III, M. Neufeld, and K. Farkas: 2000, 'Policies for dynamic clock scheduling'. In: *Symp. on Operating Systems Design and Implementation*.

Harmon, M., T. P. Baker, and D. B. Whalley: 1992, 'A Retargetable Technique for Predicting Execution Time'. In: *IEEE Real-Time Systems Symposium*. pp. 68–77.

Healy, C. A., D. B. Whalley, and M. G. Harmon: 1995, 'Integrating the Timing Analysis of Pipelining and Instruction Caching'. In: *IEEE Real-Time Systems Symposium*. pp. 288–297.

Hong, I., M. Potkonjak, and M. Srivastava: 1998a, 'On-line scheduling of hard real-time tasks on variable voltage processor'. In: *Int'l Conference on Computer-Aided Design*.

Hong, I., G. Qu, M. Potkonjak, and M. Srivastava: 1998b, 'Synthesis techniques for low-power hard real-time systems on variable voltage processors'. In: *19th Real-Time Systems Symposium*.

Ishihara, T. and H. Yasuura: 1998, 'Voltage scheduling problem for dynamically variable voltage processors'. In: *Proceedings of the 1998 international symposium on Low power electronics and design*. pp. 197–202, ACM Press.

Kang, D., S. Crago, and J. Suh: 2002, 'A Fast Resource Synthesis Technique for Energy-Efficient Real-time Systems'. In: *IEEE Real-Time Systems Symposium*.

Krishna, C. and Y. Lee: 2000, 'Voltage clock scaling adaptive scheduling techniques for low power in hard real-time systems'. In: *6th Real-Time Technology and Applications Symposium*.

Lee, Y. and C. Krishna: 1999, 'Voltage clock scaling for low energy consumption in real-time embedded systems'. In: *6th Int'l Conf. on Real-Time Computing Systems and Applications*.

Lee, Y.-H. and C. M. Krishna: 2003, 'Voltage-Clock Scaling for Low Energy Consumption in Fixed-Priority Real-Time Systems'. *Real-Time Syst.* **24**(3), 303–317.

Li, Y.-T. S., S. Malik, and A. Wolfe: 1995, 'Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software'. In: *IEEE Real-Time Systems Symposium*. pp. 298–397.

Li, Y.-T. S., S. Malik, and A. Wolfe: 1996, 'Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches'. In: *IEEE Real-Time Systems Symposium*. pp. 254–263.

Lim, S.-S., Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim: 1994, 'An Accurate Worst Case Timing Analysis for RISC Processors'. In: *IEEE Real-Time Systems Symposium*. pp. 97–108.

Liu, C. and J. Layland: 1973, 'Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment'. *J. of the Association for Computing Machinery* **20**(1), 46–61.

Liu, Y. and A. K. Mok: 2003, 'An integrated approach for applying dynamic voltage scaling to hard real-time systems'. In: *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposium*.

Lorch, J. and A. J. Smith: 2001, 'Improving dynamic voltage scaling algorithms with PACE'. In: *Proceedings of the ACM SIGMETRICS 2001 Conference*.

Lu, C., J. Stankovic, G. Tao, and S. Son: 1999, 'Design and evaluation of a feedback control EDF scheduling algorithm'. In: *IEEE Real-Time Systems Symposium*.

Lu, C., J. A. Stankovic, G. Tao, and S. H. Son: 2002a, 'Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms'. *Real-Time Syst.* **23**, 85–126.

Lu, Z., J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron: 2002b, 'Control-Theoretic Dynamic Frequency and Voltage Scaling for Multimedia Workloads'. In: *Conference on Compilers, Architecture and Synthesis for Embedded Systems*. pp. 156–63.

Mächtel, M. and H. Rzehak: 1996, 'Measuring the Influence if Real-Time Operating Systems on Performance and Determinism'. *Control Eng. Practice* **4**(10), 1461–1469.

Minerick, R., V. W. Freeh, and P. M. Kogge: 2002, 'Dynamic Power Management using Feedback'. In: *Proceedings of Workshop on Compilers and Operating Systems for Low Power*.

Mosse, D., H. Aydin, B. Childers, and R. Melhem: 2000, 'Compiler-assisted dynamic power-aware scheduling for real-time applications'. In: *Workshop on Compilers and Operating Systems for Low Power*.

Mueller, F.: 2000, 'Timing Analysis for Instruction Caches'. *Real-Time Systems* **18**(2/3), 209–239.

Park, C. Y.: 1993, 'Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths'. *Real-Time Systems* **5**(1), 31–61.

Pering, T., T. Burd, and R. Brodersen: 1995, 'The simulation of dynamic voltage scaling algorithms'. In: *Symp. on Low Power Electronics*.

Pillai, P. and K. Shin: 2001, 'Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems'. In: *Symposium on Operating Systems Principles*.

Pouwelse, J., K. Langendoen, and H. Sips: 2000, 'Dynamic Voltage Scaling on a Low-Power Microprocessor'. *Technical report,Delft University of Technology*.

Puschner, P. and C. Koza: 1989, 'Calculating the Maximum Execution Time of Real-Time Programs'. *Real-Time Systems* **1**(2), 159–176.

Saewong, S. and R. Rajkumar: 2003, 'Practical Voltage-Scaling for Fixed-Priority RT-Systems'. In: *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposium*.

Shin, Y., K. Choi, and T. Sakurai: 2000, 'Power optimization of real-time embedded systems on variable speed processors'. In: *Int'l Conf. on Computer-Aided Design*.

Wegener, J. and F. Mueller: 2001, 'A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints'. *Real-Time Systems* **21**(3), 241–268.

Weiser, M., B. Welch, A. Demers, and S. Shenker: 1994, 'Scheduling for reduced cpu energy'. In: *1st Symp. on Operating Systems Design and Implementation*.

Zhang, F. and S. T. Chanson: 2002, 'Processor Voltage Scheduling for Real-Time Tasks with Non-Preemptable Sections'. In: *IEEE Real-Time Systems Symposium*.

Zhang, N., A. Burns, and M. Nicholson: 1993, 'Pipelined Processors and Worst Case Execution Times'. *Real-Time Systems* **5**(4), 319–343.

Zhu, Y. and F. Mueller: 2002, 'Preemption Handling and Scalability of Feedback DVS-EDF'. In: *Workshop on Compilers and Operating Systems for Low Power*.

Zhu, Y. and F. Mueller: 2004a, 'Feedback EDF Scheduling Exploiting Dynamic Voltage Scaling'. In: *IEEE Real-Time Embedded Technology and Applications Symposium*. pp. 84–93.

Zhu, Y. and F. Mueller: 2004b, 'Feedback EDF Scheduling Exploiting Hardware-Assisted Asynchronous Dynamic Voltage Scaling'. Technical Report TR 2004-35, Dept. of Computer Science, North Carolina State University.