# ABSTRACT

ZHU, YIFAN **Dynamic Voltage Scaling with Feedback Scheduling for Real-time Embedded Systems.**(Under the direction of Dr. Frank Mueller).

Dynamic voltage scaling (DVS) is a promising method to reduce the power consumption of CMOS-based embedded processors. However, pure DVS techniques do not perform well for dynamic systems where the execution times of different jobs vary significantly. A novel DVS scheme with feedback control mechanisms for hard real-time systems is proposed in this work. It produces energy-efficient schedules for both static and dynamic workloads. Task-splitting, slack-passing and preemption-handling schemes are proposed to aggressively reduce the speed of each task. Different feedback control structures are integrated into the DVS algorithm to make it adaptable to workload variations. This scheme relies strictly on operating system support. It is evaluated in simulation as well as on an embedded platform. For given task sets, simulation experiments demonstrate the benefits of this scheme with savings of up to 29% in energy over previous work. This scheme exhibits up to 24% additional energy savings over other DVS algorithms on the embedded platform. The feedback-based DVS scheme is further extended to be leakage aware, which considers not only dynamic but also static power consumption caused by leakage current in circuits. A combined DVS, delay and sleeping scheme is proposed for architectures where static power exceeds dynamic power in some cases. DVS is used when dynamic power dominates the total power consumption, while a sleep mode is entered when static power becomes dominant. The extended algorithm, DVSleak, shows 30% additional energy savings on average over a pure DVS algorithm in the simulation experiment.

# Dynamic Voltage Scaling with Feedback EDF Scheduling for Real-time Embedded Systems

by

**Yifan Zhu**

A dissertation proposal submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Doctor of Philosophy

**Department of Computer Science**

Raleigh

2005

**Approved By:**

| | |
|---|---|
| Dr. Robert Fornaro | Dr. Vincent W. Freeh |
| Dr. Frank Mueller<br>Chair of Advisory Committee | Dr. Douglas S. Reeves |

## Biography

Yifan Zhu was born in Hangzhou, a city on the east coast of China, and moved to the city of Wuhan when he was five. He pursued a Bachelor of Science degree in Computer Science from Huazhong University of Science and Technology at Wuhan, China in 1998. Upon completion of his undergraduate coursework, Yifan entered the graduate school at the same university and received a Master of Science degree in Computer Science in 2001. He came to North Carolina State University in the Fall of 2001. He is a research assistant at the Center for Embedded Systems Research (CESR). With the defense of this dissertation, he is receiving the Ph.D degree in Computer Science from North Carolina State University in August 2005.

## Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. First, I would like to thank my adviser, Dr. Frank Mueller, for being a great mentor to me both personally and professionally. His suggestions and encouragement stimulated me all the time during the research and writing of this thesis. I also want to thank all the members in the embedded group for their support in my research work. Especially, I am obliged to Ajay Dudani for his contribution on our early work of the DVS research. I also want to thank Aravindh v. Anantaraman, Ali El-Haj Mahmoud and Rvai K. Venkatesan for their assistantship in an initial design and implementation of the DVS system on the IBM 405LP board. I would like to thank Bishop Brock from IBM, Austin for his valuable help on some of the technical details of the IBM experimental board.

I am also indebted to Dr. Douglas S. Reeves, Dr. Vincent W. Freeh and Dr. Robert Fornaro for serving on my dissertation committee and their valuable suggestions on this thesis.

Finally, but not least, I want to thank my parents and my girlfriend for their constant emotional support and good wishes during the odyssey of my Ph.D study.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Energy consumption is a major concern for today's computer systems. For general-purpose systems, such as desktop servers or cluster computers, energy management is necessary due to operational cost and environment issues. For example, currently over 25% of the total operational cost comes from air conditioning, backup cooling and power delivery systems [48, 59]. The worldwide total power dissipation of desktop computer processors was 160 megawatts in 1992, which increased to 9000 megawatts in 2001 [66]. Sustained high power consumption produces excessive heat, which may cause failure of the CPU and other hardware components. Empirical data from two leading vendors indicates that the failure rate of a computer node doubles with every 10°C increase [9]. The total product cost also increases since more complex cooling and packaging designs are required to serve high energy systems. Intel estimates that more than $1/W per CPU chip will be incurred if the CPU's power dissipation exceeds 35-40W [65].

For battery-powered embedded systems, efficient energy management is especially important. Examples of such devices include Web pads, advanced personal digital assistants (PDAs), cell phones and pocket PCs. Their peak performance demands could exceed 500 MIPS but the power consumption of the processor core during that peak activity should best be kept at or below 500mW [50]. Increasing battery capacity is one solution to this problem, although no dramatic breakthrough is foreseeable right now. Projected improvements in the capacity of batteries (5-10% annual growth rate) are much slower than what is needed to support ever-increasing processor power [31]. Therefore, it becomes an urgent challenge to cut energy consumption with efficient energy management schemes. Reducing power consumption can result in the same order of magnitude of energy savings as the improvement of battery technology itself.

The energy consumption problem can be approached from either the hardware perspective or the software perspective. From the hardware perspective, multiple power states are integrated into the micro-architecture level, the circuit level, and the device level. Some industry standards were developed, such as the Advanced Power management (APM) specification and the Advanced Configuration and Power Interface (ACPI). Today's major microprocessor manufacturers developed their own power management schemes in processor products, such as AMD's PowerNow technology, Intel's SpeedStep technology and Transmeta's LongRun technology. Lower power consumption also allows more densely packed circuits, resulting in higher speed and more affordable microprocessor chips. From the software perspective, energy management has traditionally focused on coarse-grained power shutdown strategies, which put the computer into a sleep or suspend state whenever the system is idle. Since the CPU sleep state requires a high-overhead shutdown and wake-up operation, it is not an available option in many situations. The sleep state also restricts system functionality, resulting in slower response time for external requests. In the absence of long idle periods in a system schedule, such a coarse-grained power management strategy becomes infeasible. More sophisticated approaches are required, which is one of the contributions of this thesis.

To study the power consumption problem, an appropriate power model is required. The following power dissipation model for CMOS-based processors is widely used [48]:

$$P \approx AC_L V_{dd}^2 f_{clk} + I_{leak} V_{dd} + P_{short} \tag{1.1}$$

where $P$ is the power dissipation, $V_{dd}$ is the voltage supply, $f_{clk}$ is the clock frequency, $A$ is the activity of the gates in the system, and $C_L$ is the total capacitance seen by the gate outputs. The first component, $AC_L V_{dd}^2 f_{clk}$, is called dynamic power, which is the power consumption of charging and discharging the capacitive load on each gate's output. The second component, $I_{leak}V_{dd}$, is called static power or leakage power, which is caused by the leakage current in the circuit. The third component, $P_{short}$, is the short-circuit power, which reflects the power dissipation due to short-circuit current. Among the three components, dynamic power is the dominant factor, while the other two can be ignored in most situations. This is the assumption of our research, although in later chapters of this dissertation we consider systems where the leakage power is dominant due to a trend toward lower threshold voltages.

Different solutions have been proposed to reduce the energy consumption of a CMOS-based processor based on Equation 1.1. They include circuit redesign, clock gating and dynamic voltage scaling [65]. Circuit redesign reduces the capacitance $C_L$ by restructuring the logic. Clock gating partitions the circuit into different clock domains and turns different domains on or off according to their usage requirements. Dynamic voltage scaling lowers the supply voltage $V_{dd}$ and clock frequency $f_{clk}$ to reduce the dynamic power of the processor. Because of the quadratic relationship between the supply voltage and power consumption, dynamic voltage scaling can result in significant power reduction for processors.

## 1.1 Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) and dynamic frequency scaling (DFS) are mechanisms that dynamically change the voltage and frequency of a processor to reduce its energy consumption. DVS and DFS are usually combined together because reduced voltage also limits the maximum clock frequency, $f_{max}$, as shown in the following:

$$f_{max} \propto (V_{dd} - V_t)/V_{dd} \tag{1.2}$$

In this dissertation, we use DVS to refer to both dynamic voltage scaling and dynamic frequency scaling. Processors with DVS functionalities provide special control registers to determine the CPU clock frequency and supply voltage. Software updates these registers dynamically during program execution to change the CPU frequency and voltage. Although this feature is provided by hardware, it is the software which decides when and how to adjust the frequency and voltage.

In contrast to previous coarse-grained energy saving solutions such as turning off processors or I/O devices, DVS is a fine-grained energy saving mechanism. Frequency and voltage scaling incur much less performance overhead than processor shutdown operations. Therefore, it is possible to exploit aggressive power management policies for DVS algorithms.

We study task-based DVS algorithms for hard real-time systems in this dissertation. DVS algorithms can be either interval-based or task-based. Interval-based approaches divide the time into fixed-length intervals. DVS operation is activated only at the beginning of each interval. Interval-based schemes are mostly used in general-purpose systems with mixed workloads. However, interval-based DVS schemes are not suitable for real-time systems where each task has its own timing requirement. For example, in a periodic real-time task model, each task is described by its period (P), deadline (d), and worst-case execution time (WCET). A violation of such timing requirements results in either performance degradation for soft real-time systems, or even serious consequences for hard real-time systems. Interval-based DVS schemes do not take this per-task timing information into consideration. Deadline misses cannot be detected until the start of the next interval. A task-based DVS approach, however, adjusts CPU frequency and voltage on a per-task basis instead of at a fixed time interval. DVS functionality is integrated into operating system schedulers so that it can be activated each time a task is dispatched or completed. Therefore, a task-based DVS scheme is a better solution to real-time systems than an interval-based DVS scheme.

Real-time systems bring new opportunities as well as new challenges to DVS algorithms. On one hand, powerful processors are often used in real-time systems to meet the worst-case execution demands, although these demands rarely occur in practice. System utilization is often kept at a low level to ensure operational safety, which creates the opportunity for DVS schemes to reduce CPU frequency. On the other hand, real-time systems expose a fundamental trade-off between energy consumption and timing requirements. Reducing the CPU frequency results in a slower computing speed. A specific operation may consume less power on average but take longer to complete. DVS algorithms need to guarantee that the timing requirements of tasks are maintained. Power consumption is reduced while continuous system services and quick response times are still available. In addition, frequency or voltage switching overhead may cancel out the benefits of DVS schemes. Deploying DVS intensively may influence the timing behavior of the system and actually result in more energy consumption. All these issues must be considered during the design of a DVS algorithm for real-time systems.

## 1.2 Motivation

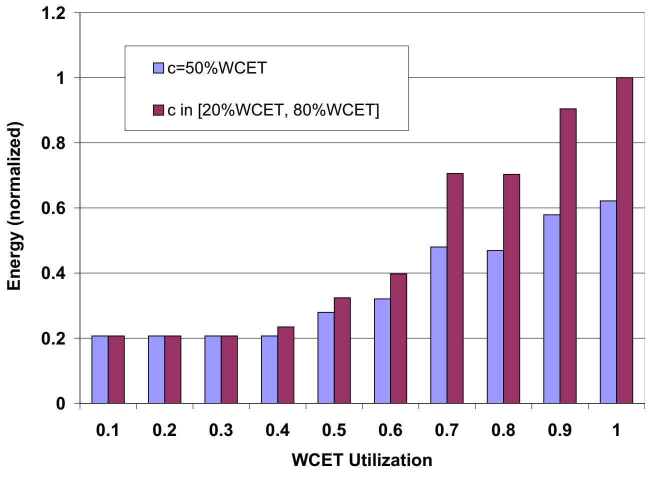


Figure 1.1: Look-ahead RT-DVS Energy for Constant/Fluctuating Workload

The potential to save energy by combining DVS techniques with operating system scheduling has been investigated in previous work. Significant savings have been reported for general-purpose computing systems [13, 16, 30, 40, 52, 69, 55, 15] as well as real-time systems [19, 20, 32, 63, 53, 10, 47, 14, 2, 29]. DVS algorithms for general-purpose systems often use various heuristics to reduce processor voltage or frequency according to the observed system workload [69, 52, 13]. DVS for hard real-time systems, in contrast, requires more subtle control. Timing requirements must be considered by the DVS algorithms to determine the processor frequency.

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 [56, 51, 17, 73, 37, 18, 1, 35, 36, 12, 49, 68]. Prior experiments have shown a wide variation between longest and shortest execution times for many actual applications. For example, actual execution times of real-world embedded tasks are observed to vary by as much as 87% relative to their measured WCET [68]. Budgeting for the WCET may result in excessive energy consumption even though actual utilization is lower than the worst case.

Also, pure DVS techniques do not perform well for dynamic systems where the system workloads vary significantly. 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 [53] between a constant workload and a fluctuating workload, as depicted in Figure 1.1. Both workloads contain three periodic tasks defined as $T_1$={3,8}, $T_2$={3,10} and $T_3$={1,14}, where $T_i$={WCET,Period} for $i = 1...3$. The constant workload consists of tasks whose actual execution times (denoted by $c$) among different jobs are 50% of their WCET. The fluctuating workload consists of tasks with an average execution time of 50% WCET. Their actual execution times fluctuate between 20% and 80% of their WCET (following variation patterns similar to Figure 7.1, discussed later). Figure 1.1 demonstrates that, in the worst case, Look-ahead RT-DVS degrades up to 40% for the fluctuating workload. More adaptable DVS schemes are required for these workloads with dynamic changing execution times.

## 1.3 Contributions

In this thesis, we develop and evaluate a novel DVS technique for dynamic workloads, considering practical design and implementation issues. The novel contributions of this thesis over previous work include:

- A feedback-based DVS framework for dynamic workloads with hard real-time requirements. A feedback controller is integrated into the DVS scheduler to achieve better adaptivity for dynamic task sets with fluctuating execution times. The speed of the processor is adjusted dynamically by the operating system. The feedback technique enables the system to select the appropriate frequency and voltage settings so that energy consumption is significantly reduced. It also helps guarantee the timing requirements of hard real-time tasks so that deadlines are not missed. Different feedback control structures are evaluated in our implementation. To the best of our knowledge, this is the first study of feedback control techniques exploiting DVS for hard real-time systems.

- A combined intra-task and inter-task DVS scheme. In contrast to compiler-directed intra-task DVS algorithms where the speed of a task is changed multiple times during program execution, the combined intra-task and inter-task DVS scheme presented in our work divides the execution budget of a task into at most two portions. Keeping the first portion at a low speed makes our algorithm more aggressive than a pure inter-task DVS algorithm. Changing the speed at most once for each task incurs lower overhead than that of a pure intra-task DVS algorithm.

- Slack passing and preemption-handling schemes for DVS schedulers. These schemes ensure the timing requirements of hard real-time tasks. They follow a greedy policy by passing as much slack as possible to scale the next running task. It speculates on the early completion of each task to aggregate unused slack for other tasks. When preemption occurs, the preempted task relinquishes its remaining slack and passes it on to the next task, while reserving enough slack in the future to avoid deadline misses. Different slack reservation schemes are studied to ensure the schedulability of the system.

- The implementation of our feedback DVS scheme in simulation, as well as the evaluation on a real embedded platform. For task sets with different dynamic execution time patterns, simulation experiments demonstrate the benefits of this scheme with energy savings of up to 29% over previous work. The evaluation on an embedded platform exhibits up to 24% additional energy savings over other DVS algorithms. The comparison of synchronous and asynchronous DVS switching shows that the energy saving under asynchronous switching is not as significant as expected. The experimental results reveal that the $V^2f$ power model (Equation 1.1) works well for DVS performance analysis.

- An extension of the feedback-DVS scheme to embedded architectures where the dynamic power is not dominant. An combined DVS and leakage control scheme is presented to save both static and dynamic power. It automatically alternates between a voltage-scaling mode and a processor sleep mode, according to the execution scenario of tasks. Simulation experiments show that the combined DVS and leakage control scheme saves 15% additional energy on average over a pure sleep policy and 30% additional energy on average over a pure DVS algorithm.

## 1.4 Dissertation Outline

The remaining chapters are structured as follows. In Chapter 2 we present related work. Chapter 3 gives an overview of the proposed feedback-DVS framework. We describe in detail the voltage-frequency selector in Chapter 4, and the feedback controller in Chapter 5. Chapter 6 gives an algorithmic description of the DVS framework, as well as some examples. Chapter 7 presents simulation results to demonstrate the performance of our feedback-DVS scheme under different workload conditions. Chapter 8 evaluates our algorithm on an embedded platform. Chapter 9 discusses the leakage power issue and proposes a leakage-aware DVS scheme. Chapter 10 summarizes this research and indicates future work.

# Chapter 2

# Related Work

The proposed feedback-DVS frame combines feedback real-time control with dynamic voltage scaling techniques. In this chapter we describe some of the related research work.

## 2.1 Dynamic Voltage Scaling

Dynamic voltage scaling has been studied by many researchers for general-purpose systems as well as real-time embedded systems. DVS for general-purpose systems is different from DVS for real-time systems. On one hand, general-purpose systems do not need to maintain any workload timing requirements, which have to be guaranteed by real-time systems. On the other hand, general-purpose systems have no knowledge of the system's worst-case behavior, which is usually available in real-time systems.

A DVS algorithm can be either on-line or off-line. On-line algorithms assign the processor frequency at run-time according to the dynamic state of the system. Off-line algorithms determine the processor frequency statically, before the execution of the system.

Weiser *et al.* [69] and Govil *et al.* [13] are among the first researchers who propose DVS algorithms in operating systems. In 1994, Weiser *et al.* first proposed an interval-based DVS algorithm to monitor CPU utilization constantly on a general-purpose operating system. Processor frequency and voltage are adjusted at the beginning of each interval according to the CPU utilization of previous execution traces. Govil *et al.* compared a number of DVS policies in a simulation environment. Their work suggested that a simple smoothing algorithm was better than a more complex algorithm. Since then, DVS strategies were further evaluated and extended by Pering *et al.* [52] and Grunwald *et al.* [16]. Pering *et al.* examined DVS algorithms through trace-driven simulation. Grunwald *et al.* evaluated DVS policies through physical measurements. Chandrasena *et al.* [7] incorporated the strengths of the conventional workload averaging technique and the rate selection algorithm. System workloads are buffered to estimate the CPU rate until the scaling factor matches the system quantized rates. Saputra *et al.* [61] presented off-line compiler-directed DVS algorithms based on integer linear programming to accommodate energy and performance constraints.

The optimality of DVS algorithms is also studied in some of the previous work. Ishihara *et al.* [24] proved that on a processor with a small number of discrete variable voltages the energy consumption is minimized when the schedule contains at most two voltage levels. The two-speed schedule is only optimal when a task's actual execution time can be determined statically, which is in practice 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. On an ideal processor with continuous voltage and frequency levels, the energy consumption is minimized when the task runs at a constant speed and completes exactly before its deadline. This optimality has also been mentioned by Weiser *et al.* [69] and Lorch *et al.* [40], and been formally proved by Yao *et al.* [71] and Ishihara *et al.* [24]. Qu [57] presented the optimality of a DVS algorithm using a more realistic model where the voltage and frequency switching overhead are considered. When the system's workload requirement is known only probabilistically, Lorch and Smith [40] showed that a constant speed is not optimal anymore. Instead, the energy is expected to be minimized by gradually increasing the CPU frequency as the task progresses. Xie *et al.* [70] explored opportunities and limits of compile-time DVS scheduling. A mixed integer linear program formulation is used to analyze the potential of compiler-directed DVS algorithms. One important result of their work is that as the number of available voltage levels increase, the energy savings decrease significantly. Saewong *et al.* [60] proposed a series of voltage scaling schemes targeting different hardware configurations and task set characteristics. Their results show that some non-optimal schemes may be more suitable than optimal schemes when the system has a high voltage scaling overhead.

When DVS algorithms are applied to real-time systems, timing requirements of real-time applications pose additional challenges. Lee *et al.* [33] presented a branch-and-bound algorithm to determine statically the operating frequency of real-time task sets. Due to the complexity of the algorithm, only two frequency levels are assumed in their model. The algorithm proposed by Liu *et al.* [39] 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. Pillai and Shin [53] proposed a set of dynamic DVS algorithms based on traditional hard real-time mechanisms, namely rate-monotone scheduling and earliest-deadline-first scheduling. They extended the schedulability test of RM and EDF algorithms to incorporate CPU frequency scaling. Static DVS, cycle conservative DVS, and look-ahead DVS are presented. Look-ahead DVS is the most aggressive DVS scheme among the suite of algorithms proposed. Unlike our algorithm which applies frequency scaling to only the current task, they assumed a unified frequency scaling factor on 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 by Dudani *et al.* [11].

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 [2] or static priority scheduling [14]. Aydin *et al.* discuss a series of algorithms, which dynamically reclaim unused computation time of real-time tasks to reduce the processor speed [4]. Energy-aware scheduling of hybrid workloads, including both periodic and aperiodic tasks, are further investigated by Aydin and Yang [3]. Their algorithm is based on early completion of tasks and collects idle time up to the next task's activation. We exploit both the idle time prior to the next task's activation as well as any idle slots up to the deadline of the task in the maximal schedule.

The idea of deriving a feasible dual-level DVS schedule from an ideal case was first proposed by Gruian [14, 15]. It combines off-line and on-line scheduling at both task level and task-set level. Stochastic data derived from previous task execution traces are used to produce energy-efficient schedules. Multiple frequency levels may be assigned to a single task. Our approach, instead, assigns at most two different frequencies for each task. Our algorithm targets dynamic-priority scheduling while Gruian restricts his approach to fixed-priority scheduling. Dual-speed scheduling was also investigated by others. Zhang *et al.* vary the processor speed between high and low whenever non-preemption blocking occurs [72]. Lee *et al.* assume an architecture where only two physical speed levels exist [33]. Our approach considers a more general case where multiple frequency and voltage levels are chosen by subsequent jobs of the same task or even different tasks. Jejurikar and Gupta investigate static and dynamic slowdown factors for periodic tasks [26] and combine them with procrastination scheduling [27] and preemption threshold scheduling [25]. Several of these algorithms were compared in a unified simulation environment, SimDVS [62]. In contrast, we measure power consumption on a concrete micro-architecture for several EDF-based algorithms.

Last-chance scheduling without energy considerations goes back to Chetto *et al.* [8]. We apply their philosophy in a DVS context. We develop a novel DVS variant based on task splitting with exactly two parts. Such a dual-speed 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 with a stochastic approach.

## 2.2 Feedback Real-time Scheduling

There have been a number of efforts of applying feedback techniques on general-purpose control systems. Only recently did researchers begin to incorporate feedback control to real-time scheduling theory [41, 42]. Feedback control for real-time scheduling was first investigated by Stankovic *et al.* [64]. Real-time system performance specifications are analyzed systematically through a control-theoretical method. Lu *et al.* [42] further proposed a feedback control real-time scheduling framework for unpredictable dynamic real-time systems where task execution times diverge from their worst case . 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. While their feedback control framework is for general purpose real-time scheduling, our scheme focuses on feedback control schemes for reducing energy consumption of processors.

For multimedia systems, a formal feedback control algorithm combined with dynamic voltage/frequency scaling technologies was first described by Lu *et al.* [43]. Both continuous and discrete DVS settings are exploited in their scheme to reduce energy consumption. An adaptive set-point is used to achieve fast responses with a stable multimedia throughput.

PID-Feedback control was also proposed for energy-aware computing in previous work. Varma *et al.* [67] presented a feedback-control algorithm where the previous workload execution history is used to predict the future workload behavior by a discrete-time PID function. The combination of the proportional, integral and derivative parts of the PID function provides appropriate estimation across different applications. Poellabauer *et al.* [54] applied a feedback loop on cache miss rates to make more reliable predictions of future task behavior. A general energy management scheme with feedback control was proposed by Minerick *et al.* [46]. An average energy usage is achieved by continuously adjusting the frequency of a processor to meet the energy consumption goal. A PI (proportional and integral) feedback controller is used to change the CPU frequency based on previous energy consumption. While Varma, Lu and Poellabauer's work target soft real-time systems and Minerick's work targets general purpose systems, our feedback DVS scheme focuses on hard real-time systems where timing constraints must not be violated.

## 2.3 Leakage-aware DVS Scheduling

Static power consumption caused by leakage current has incurred much attention in recent years. Conventional DVS scheduling strategies are modified to be leakage-aware. Lee *et al.* [34] proposed greedy methods to maximize the duration of idle and busy periods based on the worst-case execution time [34]. Their algorithms are integrated into conventional dynamic priority scheduling and fixed priority scheduling policies. It is most useful if there are many relatively short inter-task idle periods that can be grouped together. Since actual execution times often diverge considerably from WCET, a conceptual busy period is interspersed with dynamic slack due to early completion of tasks.

Quan *et al.* described an enhanced DVS algorithm to reduce both dynamic and static power consumption [58]. The latest release time of each job in the task set is computed off-line and subsequently used by an on-line scheduler. Their approach is based on fixed-priority scheduling while ours is based on dynamic-priority scheduling. Their online scheduler always delays the release time of a task to its latest start time (last chance) as long as the processor is idle. Such an aggressive scheme is not always the most energy efficient solution. In our algorithm, we make delay decisions based upon the actual execution time of tasks *via* feedback, which is more energy efficient on average.

Jejurikar *at al.* enhanced EDF scheduling with a procrastination algorithm [28]. A delay interval is calculated for each task, which only considers static task information and may result in a pessimistic schedule. Our scheme is integrated with the online scheduler. It converts dynamic slack, generated due to the critical speed threshold or the early completion of tasks, into idle or sleep time. Their approach also assumes that a power manager, implemented as a controller in hardware, handles interrupts and timers when new tasks are released. In contrast, our scheme does not require any special hardware support except for DVS and sleep modes.

Zhang *et al.* presented a compiler-supported solution to reduce leakage energy consumption [74]. Data-flow analysis is employed to identify basic blocks that do not utilize certain functional units. Those functional units are temporarily deactivated by compiler-generated software instructions. While their solution targets micro-architectural effects inside a processor, our approach puts the processor and *all* of its resources into the sleep mode.

# Chapter 3

# Feedback-DVS Framework

In this chapter, we first define the task model used throughout this work. The architecture of the feedback-DVS framework is then described in detail for a better understanding of the scheme.

## 3.1 Task Model

We use a periodic, fully preemptive and independent real-time task model [38] in our framework. Each task $T_i$ is defined by a triple $(P_i, D_i, C_i)$, where $P_i$ is the period of $T_i$, $D_i$ is the relative deadline of $T_i$, and $C_i$ is the worst-case execution time (WCET) of $T_i$, measured at the maximal processor frequency. We always assume $D_i{=}P_i$ in our model. 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 relative deadline is $P_i * j$. We use $c_{ij}$ to represent the actual execution time of job $T_{ij}$. Different instances of a task $T_i$ usually has different actual execution times, which are always bounded by that task's worst case execution time $C_i$. The hyperperiod H of the task set is the least common multiplier (LCM) among the tasks' periods.

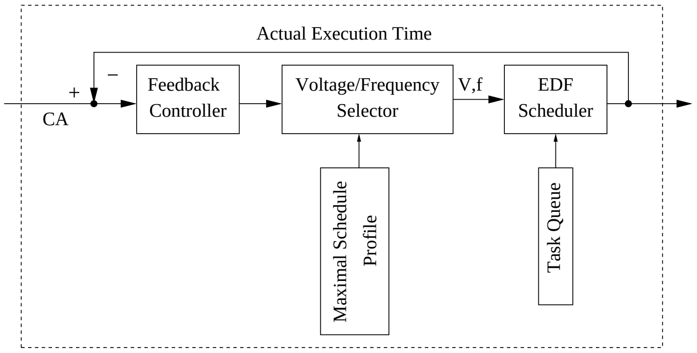


Figure 3.1: Feedback-DVS Framework

## 3.2 Architectural Framework

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 task execution time varies significantly from job to job, a DVS scheduler should 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 speed 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. This framework is based on feedback control that incrementally corrects system behavior to achieve its energy objective, while the hard real-time timing requirements are still preserved. We assume that the processor can operate at several discrete voltage/frequency levels, which reflects contemporary processor technology with support for 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 3.1 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 job and $C_i^A$, the execution time of the first portion of that job (detailed in the task-splitting scheme in the next chapter). 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 job. The maximal schedule profile includes a running scenario of the task set from start time 0 to the end of a hyperperiod. It is generated offline assuming each job's actual execution time always equals the task's worst-case execution time. The voltage-frequency selector uses the information in the maximal schedule profile to choose an appropriate voltage-frequency level without causing any deadline misses. As long as a voltage/frequency level is determined, the EDF scheduler dispatches the ready task at that 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 job is used by the feedback controller to determine the frequency and voltage for successive jobs. The next two chapters detail the mechanism of the voltage-frequency selector and the feedback controller in the feedback-DVS frame.

# Chapter 4

# 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 processor frequency and the square of the voltage [24], minimal energy consumption is obtained by running every task at a uniform processor speed. 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 job. The start point of our scheme is the following inequality, which is a modification of the standard EDF [38] schedulability test:

$$\alpha^{-1}\frac{C_k}{P_k} + \sum_{i\in\{1,\ldots,n\}\setminus\{k\}} \frac{C_i}{P_i} \leq 1 \tag{4.1}$$

Here, $\alpha$ 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 is to anticipate a near-optimal solution using a greedy scheme. In the following, we explain in detail the speed setting scheme used in our voltage-frequency

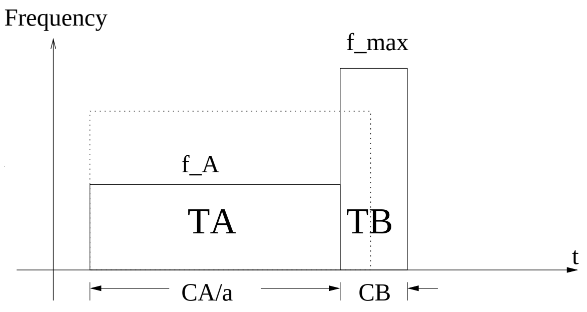


Figure 4.1: Task Splitting

selector.

## 4.1 Task Splitting

For each task, the scaling factor $\alpha$ 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 the expression 4.1 as $P_1(1 - \sum_{i=2}^{n} \frac{C_i}{P_i})$. Its $\alpha$ 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 4.1, $T_B$ always executes at the maximum frequency level $f_m$, while $T_A$ is able to execute at a lower frequency level than the level 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 its deadline. We can safely scale the frequency of $T_A$ using available slack before $T_B$ executes at the maximum frequency following a last-chance approach [8]. 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 incur at most one speed change to each task and therefore keep the impact of voltage and frequency switching overhead to a minimum. Task splitting is transparent to users. It can be implemented as a timer handler. The timer is set up upon the dispatch of $T_A$ and triggered at the end of $T_A$. If the task completes within

$T_A$ or a preemption occurs, the timer can be canceled and no additional overhead will be incurred. Only if execution cannot complete in $T_A$ will the timer go off and trigger the DVS operation to enter the $T_B$ sub-task.

Let $C_k$, $C_k^A$ and $C_k^B$ be the worst-case execution cycles of task $T_k$ and its two subtasks, $T_A$ and $T_B$. Let $s_k$ be the slack available to $T_k$ when $T_k$ is scheduled. We have:

$$C_k = C_k^A + C_k^B, \frac{C_k^A}{\alpha} + C_k^B = C_k + s_k \tag{4.2}$$

we derive $\alpha$ from the above equation:

$$\alpha = \frac{C_k^A}{C_k^A + s_k} \tag{4.3}$$

Equation 4.3 shows that when task splitting is used, the scaling factor $\alpha$ depends not only on the amount of available slack ($s_k$), but also on the number of execution cycles assigned to $T_A$. In the following, we describe the methods used to determine these two values.

## 4.2 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, schedule, or *worst-case* schedule, is the schedule produced by a standard EDF algorithm when the execution time of each job equals its WCET. The actual schedule is the actual execution scenario produced by our feedback-DVS algorithm where the execution time of each task varies from job to job. 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 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,

$$P_{n+1} = P_1, C_{n+1} = P_{n+1}(1-U), c_{n+1} = 0. \tag{4.4}$$

Notice that any other choice of idle task periods is also 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 period as the idle task's period to ensure that there is at least one idle task being released between any 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:

$$idle(t1...t2) = \sum_{t1..t2} idle\ slots \tag{4.5}$$

## 4.3 Dynamic Slack Passing

Dynamic slack passing is a technique to reduce the online complexity of slack computation. It is based on the observation that slack generated by one job is usually not exhausted when the job completes. Instead of computing each job's slack from scratch, the previous job passes its unused amount of slack to the next job. That slack is further augmented by any static idle slots between the deadline of the previous job and the next job.

When there is no preemption, we express dynamic slack passing in terms of the release time $r_{ij}$ of a task $T_{ij}$ in the actual schedule, 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:

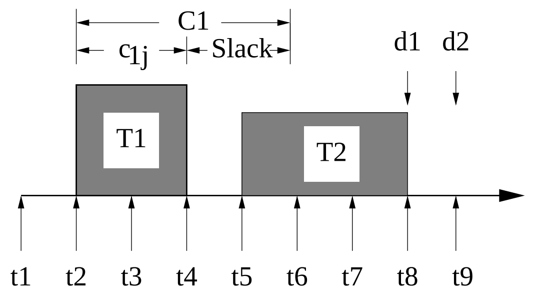


Figure 4.2: Dynamic Slack Passing

$$s_{ij} = \begin{cases} C_p - c_{pk} & if\ r_{ij} \leq I_{pk} + c_{pk} \\ F_{pk} - r_{ij} & if\ I_{pk} + c_{pk} < r_{ij} < F_{pk} \\ 0 & if\ r_{ij} \geq F_{pk} \end{cases} \tag{4.6}$$

An example is depicted in Figure 4.2. Let task $T_1$ with WCET $C_1$ and deadline t8 execute its $j^{th}$ job with an actual execution time of $c_{1j}$. Assume that when $T_1$ is invoked at time t2, it inherits a total slack of $S$ from its previous tasks. $T_1$ is then scaled to a lower frequency with that slack and completes 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 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 4.6 just adds the slack produced by the early completion of $T_1$ into the total slack. If $C_1 < c_{1j}$, Equation 4.6, in fact, reduces the total slack because $c_{1j}$ exceeds its WCET in the maximal schedule (it is feasible under DVS as long as the available slack is not exceeded). 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 is handled specially. In the next section, we derive formulas to compute the slack for a preempted task.

## 4.4 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 other tasks. When preemption occurs, the preempted task relinquishes 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 expected remaining execution time, $L_{ij}$, of the preempted task is:

$$L_{ij} = C_i - c_{ij} \times \alpha^{-1} \tag{4.7}$$

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 expected remaining execution time from its slack:

$$s_{k,r} = s_k - L_{ij} \qquad (\textit{future slots}) \tag{4.8}$$

where $s_k$ is derived from Equation 4.6 and the resulting slack $s_{k,r}$ is passed to the next task.

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

- Forward sweep: When a task $T1$ is preempted and requires $L_{1j}$ slots in the future, the preempting task, $T2$, deducts this amount from its available slack $s$. If $L_{1j} > s$, $T2$ remains without slack. If another task $T3$ is initiated, the calculation repeats itself.

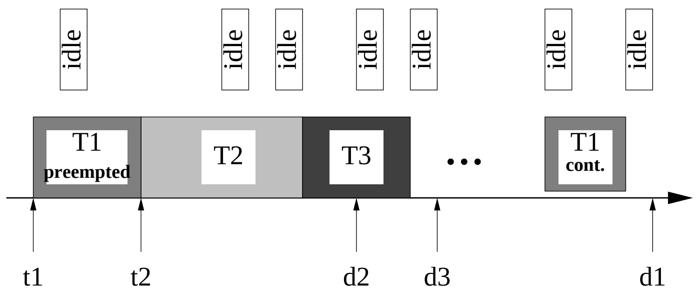


Figure 4.3: Future Slot Reservation

- Backward sweep: Future slots of $T1$ are allocated from $T_1$'s absolute deadline $d1$ backwards. Any of the idle slots in the maximal schedule become unavailable for other tasks, *i.e.*, these slots are excluded in Equation 4.6.

An example is depicted in Figure 4.3. The upper time line of idle slots presents an excerpt of the maximal schedule that depicts idle task allocations. The lower time line shows the dynamic schedule. Upon release of $T2$ at time t2, $T1$ is preempted. Let us assume that $T1$ does not have sufficient static slots (three slots) beyond t2 to finish its execution. It has to rely on future idle slots. During $T2$'s execution, $T3$ is released. Both $T2$ and $T3$ have earlier deadlines than $T1$ ($d2 < d3 < d1$). Subsequently, $T1$ only resumes after $T3$ completes.

Future slot allocation of $T1$ depends on the chosen scheme. The forward sweep results in zero idle slack for $T2$ and $T3$ since idle slots during the tasks' periods are not sufficient to cover $T1$'s future execution budget. The backward sweep, on the other hand, reserves the last 3 idle slots from d1 backwards. $T2$ and $T3$ have two and one idle slots left, which makes frequency scaling still possible.

Overall, the forward sweep is not as greedy as the backward sweep in the sense that tasks released prior to the preemption point may not be scaled due to $T1$'s future slots. A forward sweep is likely to result in zero slack for the preempting task $T2$, if $P2 << P1$, *i.e.*, the period of $T_2$ is much shorter than $T_1$'s period. Fewer idle slots are available in the forward sweep scheme, which may not suffice to cover $T1$'s future requirements. The backward sweep always results in a more greedy solution in delaying the requirements of $T1$ as late as possible. This is consistent with the observation that early completion is likely to generate slack for every task, a property inherent to our algorithm.

# Chapter 5

# Feedback Controller

This chapter first reviews the basic PID feedback control design. It then presents a single proportional-feedback control design, a multi-input control design, and a single-input control design for our DVS algorithm. Finally, the stability of the feedback-DVS system is analyzed.

## 5.1 Basic PID Control

Equation 4.3 shows that the scaling factor $\alpha$ of a task $T_i$ depends not only on the amount of available slack but also on $C_i^A$, the number of execution cycles assigned to the first subtask $T_A$. Static slack utilization and dynamic slack passing, as described in the previous chapters, help us 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 $C_i^A$. Since $C_i^A$ is based on the estimated worst-case execution time of the first subtask $T_A$, our objective is to let $C_i^A$ approximate $T_{ij}$'s actual execution time $c_{ij}$ so that $T_{ij}$ can completes before it enters the second subtask $T_B$. Most of all, when $T_{ij}$'s actual execution time $c_{ij}$ does not exceed $C_i^A$, all of $T_{ij}$ executes at a low frequency corresponding to $\alpha$. It is not necessary for $T_{ij}$ to switch to the maximum processor frequency. Hence, a near-optimal energy consumption is obtained.

In real-time applications, the actual execution time $c_{ij}$ of each task $T_i$ often experiences fluctuations over different jobs. The fluctuations may result in tendencies leading to higher processing demands up to some peak point and receding demands after that 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 [41]. 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. 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 variable and the set point are fed back to the controller repeatedly. Corresponding system states are usually adjusted according to the differences to let the system variables approximate the set points as closely as possible.

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 independent of the proportional gain in the parallel form. We choose the following parallel form as the basis of our PID feedback implementation:

$$output = K_P \times \epsilon_i + \tfrac{1}{K_I} \int \epsilon_i \, dt + K_D \tfrac{d\epsilon_i}{dt} \tag{5.1}$$

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

$$G_P(s) = K_P + \tfrac{K_I}{s} + sK_D \tag{5.2}$$

## 5.2 Proportional Feedback Control Design

A periodic real-time workload may exhibit a relatively stable behavior during a certain interval of time. Thus, the actual execution time of different jobs remains nearly constant or only varies within a very small range. For such workloads, we use a specific PID feedback controller, which includes only a proportional control element. We choose the value of $C_i^A$ as the controlled variable while $c_{ij}$ is chosen as the set point. $C_i^A$ is chosen as 50% of the WCET for the first job of each task. While half of the task's execution is budgeted at a low frequency, half of it is reserved at the maximum frequency. The task can still meet its deadline, even if the worst case is exhibited. Initially, the energy consumption may be significant and is likely to differ from the optimal case due to inappropriate estimations of the actual execution time. Over time, we replace $C_i^A$ with the actual execution time of the task based on the execution time fed back after each task completion. The average value of execution times over past executions is utilized to anticipate future $C_i^A$ portions. On the average, this scheme allows us to complete the entire task's budget at a low frequency level, which closely approximates the optimal energy-saving schedule. Let $C_{ij}^A$ be the anticipated worst case execution time of the first sub-task of job $T_{ij}$. We define the following equations to get $C_{i,j+1}^A$, the anticipated worst case execution time of the first sub-task of job $T_{i,j+1}$:

$$\begin{aligned} C_{i1}^A &= 0.5 \times WCET \\ C_{i,j+1}^A &= (C_{ij}^A \times (j-1) + c_{ij})/j, \qquad j \geq 1 \end{aligned} \tag{5.3}$$

where $c_{ij}$ is the actual execution time of the $j^{th}$ job of task $T_i$. Each time a job completes execution, its actual execution time is fed back and aggregated to anticipate the next job's actual execution time, which is further used to calculate an ideal scaling factor for that task.

Although such a proportional feedback scheme only considers a pure gain adjustment over the anticipated $C_i^A$ value, it works well for real-time task sets where each task either has a constant actual execution time or it has an execution time varying within a small bounded range. For task sets with highly fluctuating execution times, more sophisticated feedback schemes are required, which is detailed in the next sections.

## 5.3 Multi-input Control Design

The proportional feedback control described in the previous section follows a proportional adjustment relative to average execution times. In practice, real-time embedded systems, such as audio and video playback or image processing systems, often experience fluctuating execution times of tasks over a period of time. The fluctuations may result in tendencies leading to higher processing demands up to some point and receding demands after this peak point. In order to devise a DVS algorithm adaptive to such a dynamic environment, more sophisticated feedback schemes are needed. According to the objective described above, we design a feedback scheme presented as a multiple-input (MI) control system. For every task $T_i$ in the system, its $C_i^A$ 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.*,

$$\epsilon_{ij} = c_{ij} - C_{ij}^A. \tag{5.4}$$

The error is measured periodically by the controller. Its output is fed back to the feedback-DVS scheduler to adjust the value for $C_i^A$. For $n$ tasks in the task set, there are altogether $n$ feedback inputs ($\epsilon_{ij}$, i=1...n ) and $n$ system outputs ($C_i^A$, i=1...n).

For each task $T_i$, let $C_{ij}^A$ be the estimated $C_i^A$ value for its $j^{th}$ job. The following discrete PID control formula is used in our feedback-DVS scheduler:

$$\begin{aligned} \Delta C_{ij}^A &= K_P \times \epsilon_{ij} + \tfrac{1}{K_I} \textstyle\sum_{IW} \epsilon_{ij} + K_D \tfrac{\epsilon_{ij} - \epsilon_i(t-DW)}{DW} \\ C_{i,j+1}^A &= C_{ij}^A + \Delta C_{ij}^A \end{aligned} \tag{5.5}$$

where $K_P$, $K_I$ and $K_D$ are proportional, integral, and derivative parameters, respectively. $\epsilon_{ij}$ is the monitored error. The output $\Delta C_{ij}^A$ is fed back to the scheduler and is used to regulate the next anticipated value for $C_i^A$. $IW$ 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 compromised 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.

This multi-input model achieved significant energy savings as shown in Chapter 7, but it also exhibited some drawbacks when we implemented it on real embedded platforms. The multi-input control structure increases the total memory requirements of the system, since the DVS scheduler needs to create an individual feedback controller for every task in the task set. Each feedback controller maintains a queue structure in order to store the execution time history of previous jobs, which requires additional memory space proportional to the length of the queue as well as the total number of tasks. Such per-task memory requirements limit the maximal number of tasks an embedded system can sustain. Furthermore, the multi-input model manipulates multiple inputs and multiple outputs simultaneously, which increases the complexity of the scheduler design and implementation. Given the difficulty of precisely characterizing the behavior of a control system, it also adds complexity to the theoretical analysis of the system.

In order to address the drawbacks brought by the complexity of the MI control system, we transform the above MI model into a single-input (SI) control model in the following.

## 5.4 Single-input Control Design

We now present a simplified design for the system model.

Instead of using $C_i^A(i = 1...n)$ as the controlled variable for each task $T_i$ and creating $n$ different feedback controllers for $n$ different tasks, we now define a single variable r as the controlled variable for the entire system as:

$$r_j = \frac{1}{n}\sum_{i=1}^{n}\frac{C_{ij}^A - c_{ij}}{c_{ij}} \tag{5.6}$$

where j is the index of the latest job of task $T_i$ before the sampling point. $r_j$ describes the average difference between tasks' actual execution times and their corresponding $C_i^A$ values. Our objective is to make $r$ approximate 0 (*i.e.,* the set point). The system error becomes

$$\epsilon(r_j) = r_j - 0. \tag{5.7}$$

where $\epsilon(r_j)$ reflects the error of the entire task set and is not a function of a particular task $T_i$ anymore. $\epsilon(r_j)$ is further fed back to the PID scheduler to regulate the controlled variable r. The PID feedback controller is now defined as:

$$\begin{aligned} &\Delta r_j = K_P \times \epsilon(r_j) + \tfrac{1}{K_I}\textstyle\sum_{IW}\epsilon(r_j) + K_D\tfrac{\epsilon(r_j)-\epsilon(r_{j-DW})}{DW} \\ &r_{j+1} = r_j + \Delta r_j \end{aligned} \tag{5.8}$$

where $K_P$,$K_I$ and $K_D$ are the PID parameters. IW and DW are the integral and derivative window sizes.

When job $T_{ij}$ completes, we adjust the $C_A$ value for $T_{i(j+1)}$ by $C_{i,j+1}^A = r_j \times c_{ij} + c_{ij}$, which is used by the DVS scheduler to calculate the scaling factor $\alpha$ and to determine a processor frequency and voltage for the next job. Such a single controller mechanism is easy to implement because one feedback controller suffices for the entire system, which reduces the complexity and overhead of the feedback DVS algorithm. It reduces the memory requirement of the system since only one global feedback queue needs to be created instead of n different queues for n different tasks in the multi-input feedback scheme. Such a transformation simplifies the control system so that there is only one system input $\epsilon(t)$ and one system output $r$. It eases the analysis and implementation of the feedback controller in our scheduler. But a drawback of the model is that it does not provide direct feedback of the $C_i^A$ value for each individual task. A zero value of $r$ may not necessarily imply that each task $T_i$'s $C_i^A$ 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.

## 5.5 Stability Analysis of the Single-input Feedback Control

Stability is an important metric for real-time control systems. A control system is stable if its controlled variables are always bounded for bounded input performance references and disturbances. In order to analyze the stability of the above single-input control model, we compute its transfer function in the Laplace domain. The transfer function of the PID controller is defined as:

$$G_{PID}(s) = K_P + \frac{K_I}{s} + K_D s \tag{5.9}$$

The transfer function between $r_j$ and $C_i^A$ can be derived by taking derivative of both sides of the equation 5.6:

$$G_r(s) = Ms \tag{5.10}$$

where $M = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{c_i}$. Therefore, the transfer function of the entire closed-loop feedback system can be computed as:

$$\frac{G_{PID}(s)G_r(s)}{1 + G_{PID}(s)G_r(s)} = \frac{MK_P s + MK_I + MK_D s^2}{1 + MK_P s + MK_I + MK_D s^2} \tag{5.11}$$

According to control theory, a system is stable if and only if all the poles (the denominator of its transfer function) are in the negative half-plane of the s-domain. From Equation 5.11, we infer the poles of our system as

$$\frac{-MK_P \pm \sqrt{MK_P^2 - 4MK_D(MK_I + 1)}}{2MK_D} \tag{5.12}$$

Note that $-MK_P + \sqrt{MK_P^2 - 4MK_D(MK_I + 1)}$ is always less than 0 when $MK_P^2 - 4MK_D(MK_I + 1) > 0$. Hence, all the poles are in the negative half-plane of the s-domain. Therefore, the stability of the above system is ensured.

# Chapter 6

# Algorithm and Its Correctness

This chapter presents an algorithmic description of the feedback-DVS scheme in pseudo-code. Some examples are then given to explain the scenario when the algorithm is applied on real-time task sets.

## 6.1 Algorithm Description

An algorithmic description of our feedback-DVS scheme with the PID feedback control is given in Algorithm 1. The following notations are used in the algorithm description:

- $T_{ij}$: the j-th job of task $T_i$
- *prev*: the index of the previous job immediately scheduled before $T_{ij}$
- *now*: the current time
- $P_i$: the Period of $T_i$
- $d_{ij}$: the absolute deadline of $T_{ij}$
- $C_i$: the WCET of $T_i$ (without scaling)
- $C_i^A$: the anticipated worst-case execution time of the first sub-task (low frequency portion) of $T_i$
- $C_i^B$: the anticipated worst-case execution time of the second sub-task (high frequency portion) of $T_i$

- $c_{ij}$: 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
- $L_{ij}$: the worst-case remaining execution time of $T_{ij}$ (without scaling)
- $slack$: the current slack of the system
- $idle(t1..t2)$: 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
- $\alpha$: the frequency scaling factor

This algorithm integrates the PID feedback scheme and preemption-handling with future slot reservation. Only the MI control model is presented in the pseudo-code. The SI model is implemented in a similar way. The online complexity of our algorithm is $O(n)$ for $n$ tasks, because the length of slots in the maximal schedule during the interval between the release time and deadline of the current task has to be updated when a task is released or completes. The number of slots in this interval is bounded by the number of tasks because only a constant number of jobs for each task and a constant number of preemptions may occur in this interval.

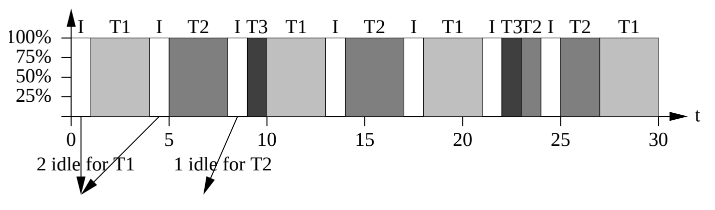
Next, let us see some examples of applying the algorithm on real-time task sets.

## 6.2 Examples

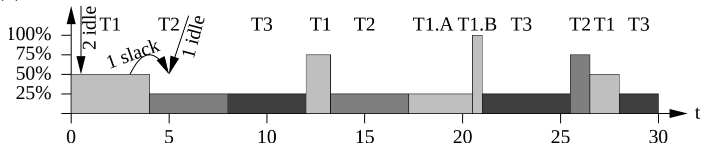
Figure 6.1(i) is an example of a static maximal EDF schedule, constructed offline. The example includes a task set of three tasks T1={3,8}, T2={3,10} and T3={1,14}, where $T_i = \{C_i, P_i\}$ denotes task $T_i$'s worst case execution time $C_i$ and its period $P_i$. An idle task I={1,4} is also included in the maximal schedule to fill underutilized processor time niches. Every task's actual execution time is one except the first job

**Algorithm 1: Feedback-DVS**

```
Procedure Initialization
begin
    foreach $T_k \in \{T_1, T_2, \ldots, T_n\}$ do
        $C_k^A \leftarrow C_k/2; \quad L_{k0} \leftarrow C_k; \quad t_i \leftarrow 0$
    $U \leftarrow \frac{C_1}{P_1} + \frac{C_2}{P_2} + \ldots + \frac{C_n}{P_n}$
    $P_{n+1} \leftarrow P_1; \quad C_{n+1} \leftarrow P_1 \times (1 - U)$
    $c_{n+1} \leftarrow 0; \quad slack \leftarrow 0$
end

Procedure TaskActivated ($T_{ij}$)
begin
    if $processor\ was\ idle\ for\ d$ then
        $slack \leftarrow slack - d$
    if $T_{prev}\ was\ prempted/interrupted$ then
        $L_{prev} = C_{prev} - c_{prev} \times \alpha\prime$
        $slack \leftarrow slack - idle(d_{ij}..d_{prev})$
        if $L_{prev} > slots(T_{prev}, now..d_{prev})$ then
            $reserve_{prev} \leftarrow L_{prev} - slots(T_{prev}, now..d_{prev})$
            $allocate\ reserve_{prev}\ in\ [now...d_{prev}]$
    else
        if $now > d_{prev}$ then
            $slack \leftarrow slack - idle(d_{prev}, now)$
        $slack \leftarrow slack + idle(d_{prev}..dij)$
    $\alpha\prime \leftarrow \min\{\frac{f_1}{f_m}, \ldots, \frac{f_m}{f_m} | \frac{f_i}{f_m} \geq \frac{C_{Aij}}{C_{Aij} + slack}\}$
    if $\alpha\prime = 1$ then
        $C_{ij}^A \leftarrow 0$
    else
        $C_{ij}^A \leftarrow slack \times \alpha\prime/(1 - \alpha\prime)$
    SetInterrupt($T_i, C_{ij}^A/\alpha\prime$)
    SetFrequency($\alpha\prime$)
end

Procedure Taskcompleted ($T_{ij}$)
begin
    $slack \leftarrow slack - c_{ij} + C_i$
    $\epsilon \leftarrow c_{ij} - C_{ij}^A$
    $\Delta C_{ij}^A \leftarrow K_p * \epsilon(t_i) + \frac{1}{I}\sum_{IW} \epsilon(t_i) + D\frac{\epsilon(t_i) - \epsilon(t_i - DW)}{DW}$
    $C_{i,j+1}^A = C_{ij}^A + \Delta C_{ij}^A$
    $t_i \leftarrow t_i + 1; \quad L_{i(j+1)} = C_i$
    if $reserve_{ij} > 0$ then
        $release\ up\ to\ |reserve_{ij}|$
end

Procedure SetInterrupt ($T_{ij}, C_{ij}^A$)
begin
    $Set\ timer\ interrupt\ for\ T_{ij}\ at\ C_{ij}^A\ time\ units$
end

Procedure SetFrequency ($\alpha\prime$)
begin
    $f \leftarrow \alpha\prime \times f_m$
end
```



(i) Static Worst-Case EDF Schedule with Idle Task I



(ii) Our Feedback DVS at Beginning of 1st Hyperperiod

Figure 6.1: Discrete Scaling Levels for 3 Tasks

of T1, who has an actual execution time of two. 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.

Next, the task set is scheduled according to our algorithm (without the idle task). Additional operations to calculate slack and to set the CPU frequency/voltage are inserted at scheduling points. As shown in Figure 6.1(ii), when the first task $T_1$ (with the earliest deadline) is activated at time 0, its initial slack is assigned according to Equation 4.6. The initial slack $s_{1,0}$ is set to 0 since no previous task had been scheduled. The value of $idle(0..d_1)$ is obtained from the pre-calculated maximal EDF schedule. Then, a frequency scaling factor $\alpha$ is set according to Equation 4.3: $\alpha = C_k^A/(C_k^A + s_k)$. The CPU frequency is set to $\alpha * f_m$. When the first task completes, unused slack is adjusted and passed on to the next task according to Equations 4.7 and 4.8. The estimated value of $C_1^A$ for the first task is updated according to our feedback scheme. When the second task is scheduled, its slack is again determined by Equation 4.6, this time with a non-zero slack on the right-hand side of the equation (since the first task passes no unused slack). The frequency level is determined in a similar way as the first task. For later task instances, the feedback scheme chooses $C_i^A$ 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 4.4, is also applied but not shown here to simplify the example.

The effect of the PID feedback scheme is shown in the following example. Consider a task set of three tasks T1={12,32}, T2={12,40} and T3={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 7.1. Figure 6.2(a) is a snapshot of the feedback-DVS schedule for this task set without PID-feedback. Figure 6.2(b) depicts the feedback-DVS schedule for the same task set using feedback with PID parameters CP=0.9, CI=0.08 and CI=0.1.

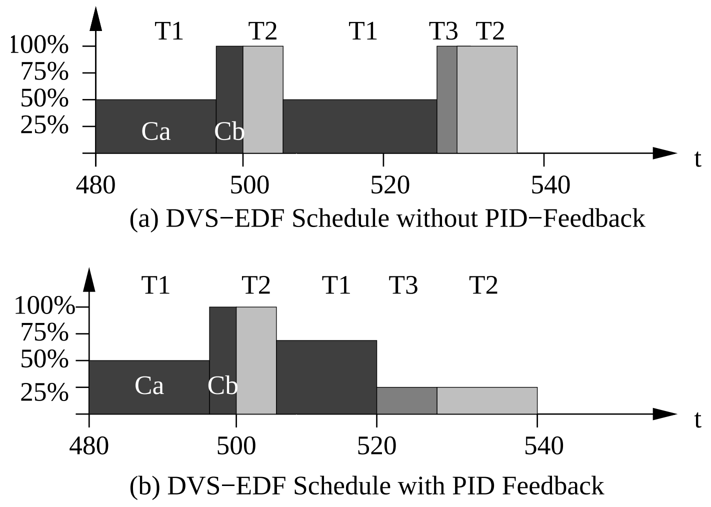


Figure 6.2: Schedules: Simple and PID Feedback

We can see from the figures that the first job of $T_3$ 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, while it starts at time 520 in the PID feedback schedule. The PID feedback scheme gets an execution time of 3.06 for its $C^A_{3,1}$ according to Equation 5. With the closer approximation of $c_{3,1}$, 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 $C^A_{2,2} = 6.76$, which is again closer to $T_{2,2}$'s actual

Table 6.1: Sample Task Set

| 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 |

execution time. This demonstrates the superiority of our feedback-DVS scheme in adapting to dynamic workloads resulting in additional energy savings.

## 6.3 Correctness of the Algorithm

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 extend a job's execution time to be longer than its WCET, 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 Table 6.1. Its worst-case schedule with an idle task and its actual schedule under feedback-DVS are shown in Figure 6.3(a) and Figure 6.3(b), 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 scales $T_3$ 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.** *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.*

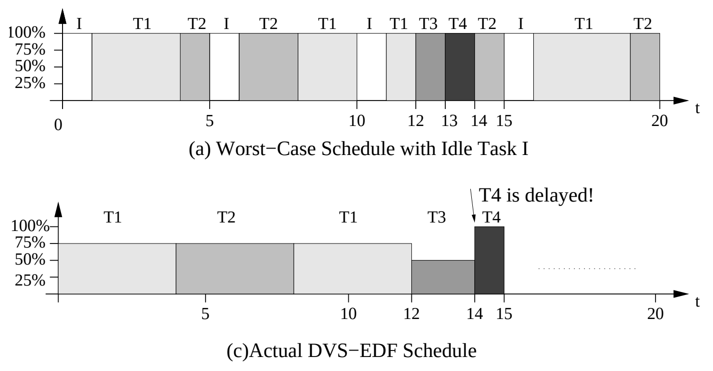


Figure 6.3: Delayed Start of Tasks due to Scaling

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:*

$$s_i - s_i^+ \leq idle(f_{i-1}^+, d_{i-1}) + \sum_{T_l \in [f_{i-1}^+, d_{i-1}]; d_l > d_i} C_l \tag{6.1}$$

*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 $T_i$.*

*Proof of Lemma 1* 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,

$$s_1 - s_1^+ = 0. \tag{6.2}$$

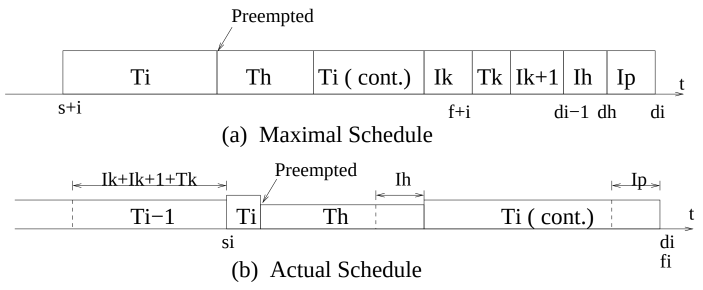


Figure 6.4: Maximal vs. Actual Schedule

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 6.4 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 :

$$s_i - s^+_i \leq \sum_{l=k}^{k+m-1} I_l + \sum_{l=k}^{k+q-1} C_l = idle(f^+_{i-1}, d_{i-1}) + \sum_{T_l \in [f^+_{i-1}, d_{i-1}]; d_l > d_i} C_l \qquad (6.3)$$

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 6.4(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:

$$f_i - f_i^+ \leq idle(f_i^+, d_i) - C_R + \sum_{T_l \in [f_i^+, d_i];\ d_l > d_{i+1}} C_l \tag{6.4}$$

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:

$$s_{i+1} - s_{i+1}^+ \leq f_i - f_i^+ \leq idle(f_i^+, d_i) + \sum_{T_l \in [f_i^+, d_i];\ d_l > d_{i+1}} C_l \tag{6.5}$$

Hence, inequation 6.1 also holds for $T_{i+1}$, and we proved the lemma. □

*Proof of Theorem 1* Lemma 1 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:

$$s_{i+1}^+ + C_{i+1} + idle(f_i^+, d_i) + \sum_{T_l \in [f_i^+, d_i];\ d_l > d_{i+1}} C_l \leq d_{i+1} \tag{6.6}$$

From Inequation 6.5 and 6.6, we derive:

$$s_{i+1} + C_{i+1} \leq s_{i+1}^+ + C_{i+1} + idle(f_i^+, d_i) + \sum_{T_l \in [f_i^+, d_i];\ d_l > d_{i+1}} C_l \leq d_{i+1} \tag{6.7}$$

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

# Chapter 7

# Simulation Experiments

This chapter presents the simulation experiments to evaluate the performance of the feedback-DVS scheme. Some of the experimental results are presented and the algorithmic performance is analyzed.

## 7.1 Experimental Method

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 [53]. We assume a processor model capable of operating at four different voltage and frequency levels, as depicted in Table 7.1. Comparable frequency and voltage settings were also used in the Look-ahead RT-DVS work [53] and the experimental work with StrongARM processors [55]. The results discussed hereafter are also consistent in their trends for power savings on a concrete DVS-capable architecture. 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.

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

Table 7.1: Processor Model for Scaling

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

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 [44]. 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 7.1.

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 the WCET and 100% of WCET. After the peak value is reached, the actual execution time of the following jobs drop exponentially (modeled as $c_i = 1/2^{(t-c_m)}$) until it reaches 50% of 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

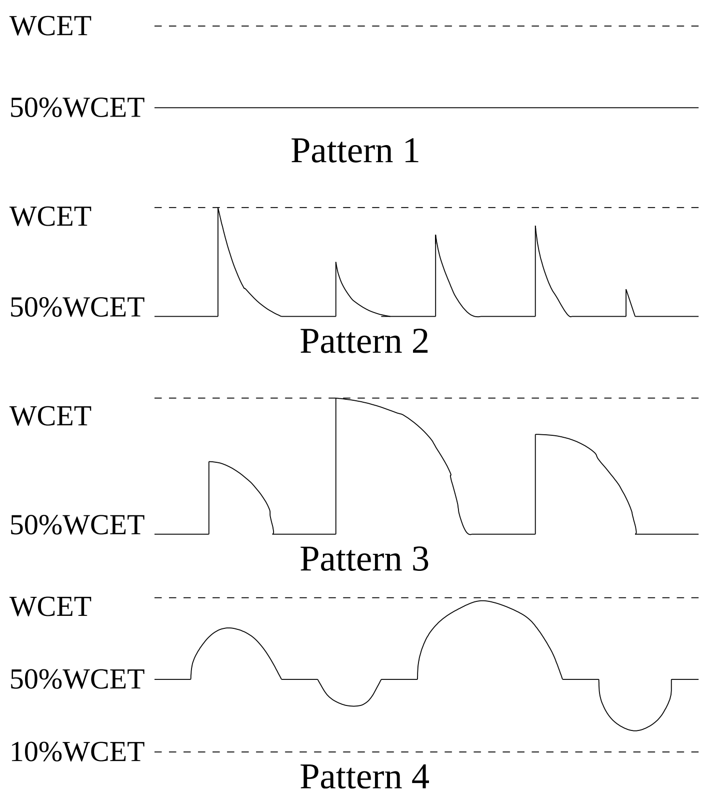


Figure 7.1: Task Actual Execution Time Pattern

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 multi-input (MI) feedback control model and the simplified single-input (SI) feedback control model were evaluated in our experiment. The corresponding feedback-DVS schedulers are referred to as MI Feedback-DVS and SI Feedback-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 $C_i^A$. 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.

## 7.2 Results

Figure 7.2 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 for all schemes. With the increase of the worst-case utilization, our feedback-DVS scheme started saving more energy than Look-ahead RT-DVS. MI Feedback-DVS adapts to the changing workload better than Look-ahead RT-DVS and costs 8% to 24% less energy than it. The maximal energy saving (24%) is observed at 80% utilization. SI Feedback-DVS works almost as good as MI Feedback-DVS, which shows that the simplified model still captures the dynamic system behavior

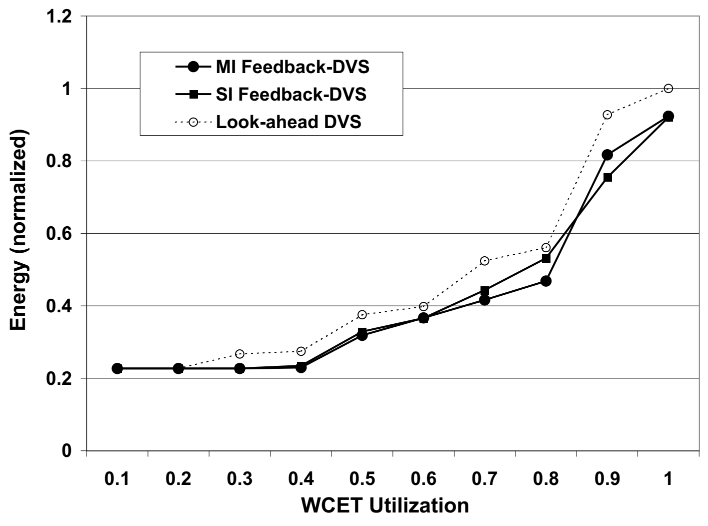


Figure 7.2: Execution Time Pattern 1

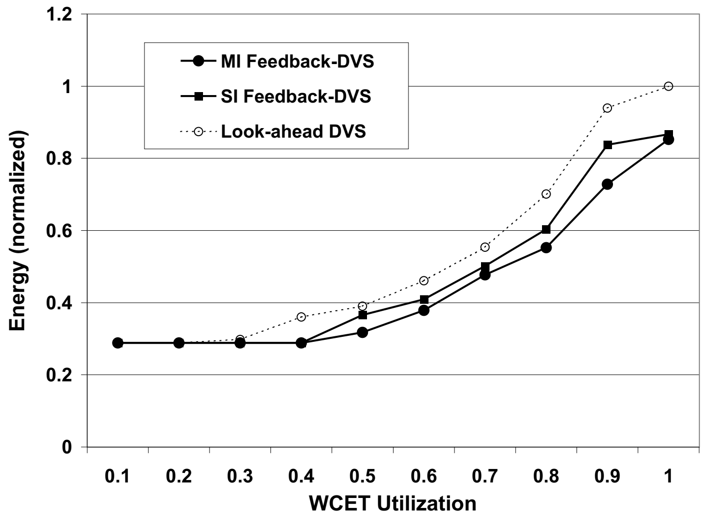


Figure 7.3: Execution Time Pattern 2

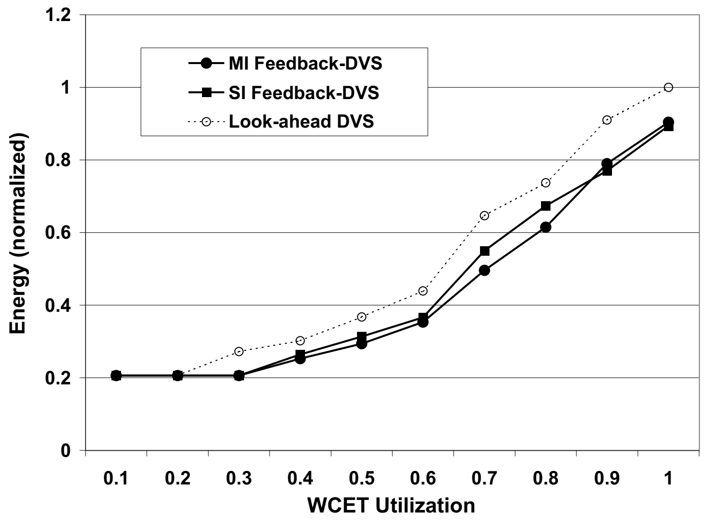


Figure 7.4: Execution Time Pattern 3

and adapts to the changing workload efficiently.

Similar results can be observed for execution time pattern 2 and 3, as depicted in Figures 7.3 and 7.4. The maximal energy saving of MI Feedback-DVS, 22% and 16%, appears at 0.5 and 0.9 utilizations, respectively. Its average energy saving over Look-ahead RT-DVS is around 15%. SI Feedback-DVS costs a little more energy than MI Feedback-DVS in some cases because SI Feedback-DVS usually takes longer time to respond to tasks' execution time variations than MI Feedback-DVS does. But overall, SI Feedback-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 different patterns of fluctuating workloads.

In order to further assess 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 7.1 has a 50% WCET baseline, the other two task sets have baselines of 75% and 25% WCET, respectively. Shifting the baseline among different task sets also results in a change of their actual utilization. Figure 7.5 and Figure 7.6 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

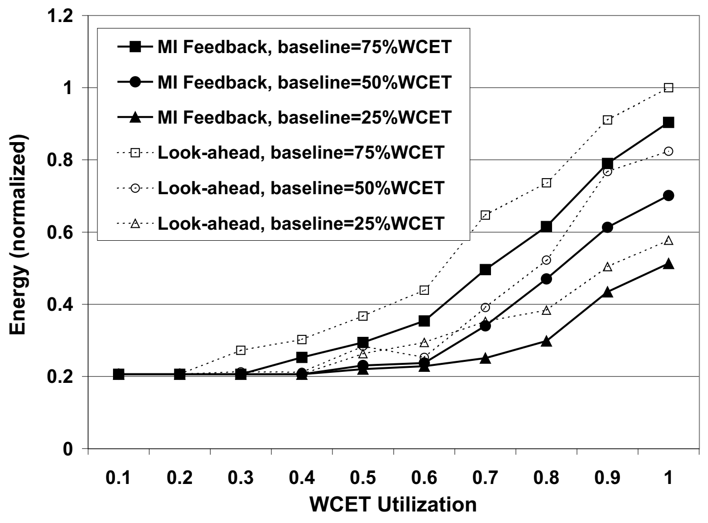


Figure 7.5: Multi-input Feedback-DVS, Varying Baseline

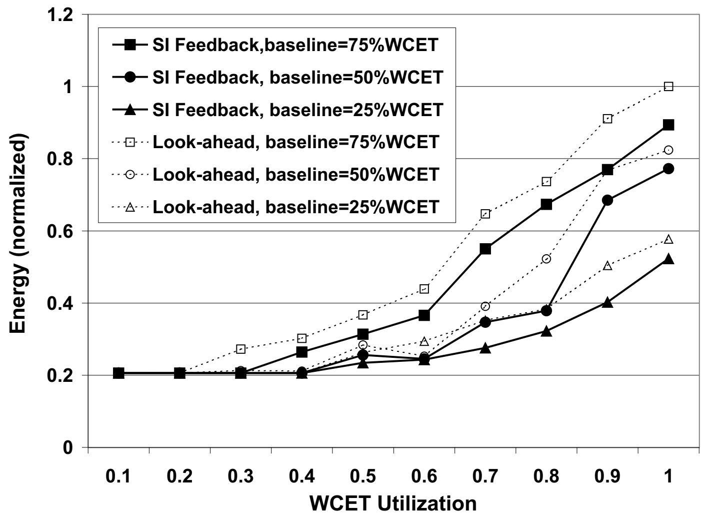


Figure 7.6: Single-input Feedback-DVS, Varying Baseline

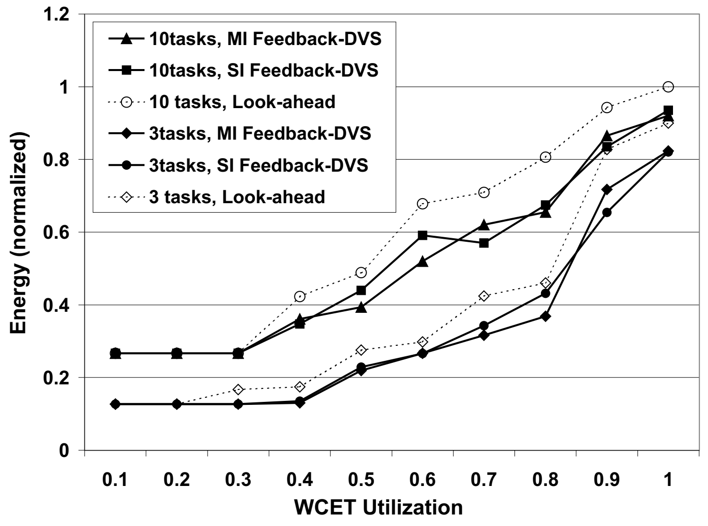


Figure 7.7: 10-task vs. 3-task under Pattern 1

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. MI Feedback-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 SI Feedback-DVS. A maximum energy saving of 26% over Look-ahead are observed at the 50% WCET baseline case. Both our schemes, MI Feedback-DVS and SI Feedback-DVS, are able to adapt to workloads with baseline variations.

Figure 7.7 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 MI Feedback-DVS and SI Feedback-DVS are able to save about 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

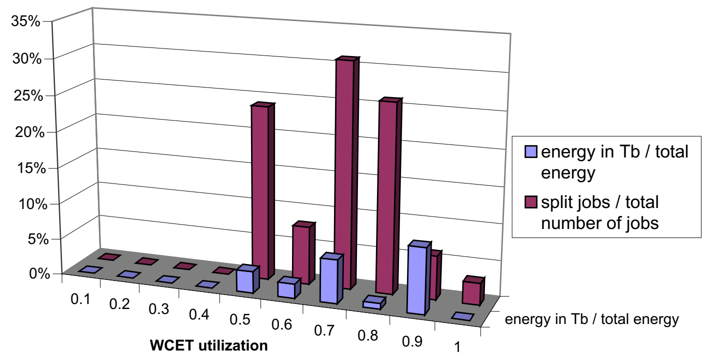


Figure 7.8: Pattern 1, Percentage of subtask(energy) in $T_B$

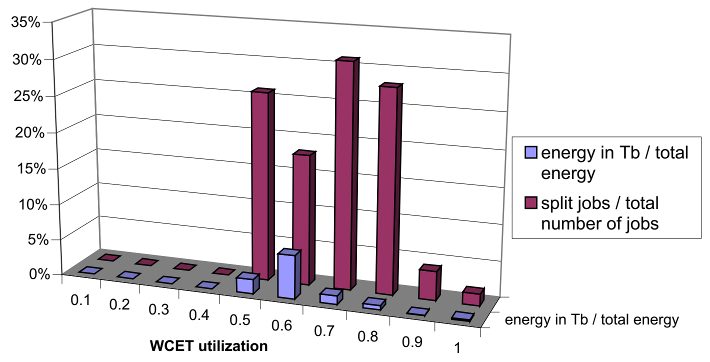


Figure 7.9: Pattern 2, Percentage of subtask(energy) in $T_B$

consumption.

The feedback-DVS scheme obtains lower energy consumption than previous non-feedback approaches because PID control makes most of the tasks complete within the $T_A$ subtask without getting into the high-frequency $T_B$ portion. In order to substantiate this claim, for each of the three task execution time patterns we measured the percentage of jobs which get into the high frequency $T_B$ subtask, as well as the percentage of energy consumed in the $T_B$ portion. The results, as depicted in Figures 7.8,7.9 and 7.10, show that the number of jobs which get into the high-frequency $T_B$ portion is constrained to be less than 31% of the total number of released jobs. The amount of energy consumed in those $T_B$ portions is even less, from 1% to 9%, compared to the total energy consumption. When a task enters its high-frequency

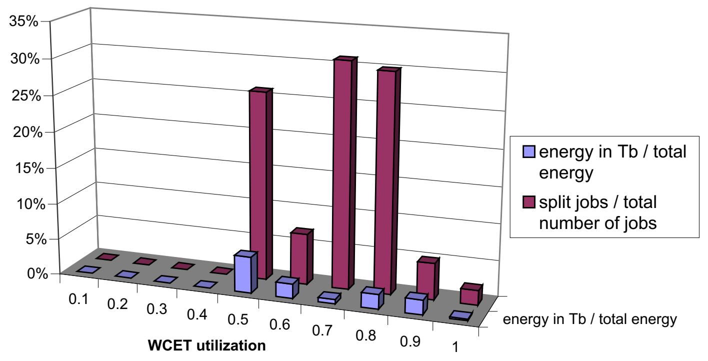


Figure 7.10: Pattern 3, Percentage of subtask(energy) in $T_B$

$T_B$ portion, it usually completes within a very short period of time. Even if $C_k^A$, the execution time produced by the PID controller for the low frequency $T_A$ subtask, is not equal to or greater than a task's actual execution time, the difference between $C_k^A$ and the actual execution time is still very close to its predicted value. These results give us insight about the benefit of adding a small amount of padding to $C_k^A$ values to improve the energy saving performance. Since most of the time $C_k^A$ is only a bit larger than a task's actual execution time, adding a small amount of padding on $C_k^A$ may remove the high-frequency $T_B$ portion and reduce the task splitting overhead. But the price we may pay for using padding is likely an increase on the total amount of energy consumption, because padding may lead to overestimated $C_k^A$ values and makes the algorithm less aggressive. How to choose an optimal amount of padding is still an open question and requires further studies. Overall, these results demonstrate the power of the PID feedback controller, which is able to adjust the system behavior dynamically according to task workload variations.

Besides the execution time patterns listed in Figure 7.1, 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.

## 7.3 Summary

Overall, our Feedback feedback-DVS algorithm is able to provide considerable energy savings for different task sets. The simplified feedback control model, SI Feedback-DVS, captures the characteristics of the overall system behavior and leads to acceptable performance comparable with MI Feedback-DVS. Feedback control in conjunction with DVS scheduling makes the system more adaptive to dynamically changing workloads and saves more energy than less adaptive schemes.

# Chapter 8

# Real Architecture Evaluation

In order to evaluate the energy saving potential of our algorithm in an actual system as opposed to a simulation environment, we implemented our feedback DVS algorithm as well as several other DVS algorithms, namely static DVS, cycle-conserving DVS, look-ahead-1/2 DVS (all by Pillai and Shin [53]), DR-OTE and AGR-2 (by Aydin *et al.* [2]) on an embedded development board. Look-ahead-1 and look-ahead-2 are the original and a modified version of the look-ahead DVS algorithm in [53], respectively. Look-ahead-1 updates each task's absolute deadline immediately when a task instance completes. Look-ahead-2 delays such updates till the next task instance is released, which results in additional energy savings. AGR-2 follows the most aggressive scheme with an aggressiveness parameter k of 0.9. In these experiments, we use simple feedback on constant workloads and single-input PID feedback for dynamic fluctuating workloads, if not stated explicitly. We compared the energy consumption as well as DVS overhead of different algorithms. We also wanted to determine if the lower frequencies and voltages chosen by our feedback scheme outweigh the higher computational overhead required to make scheduling decisions.

## 8.1 Platform and Methodology

The embedded platform used in our experiment is a PowerPC 405LP embedded board running on a diskless MontaVista Embedded Linux variant, which is based on the 2.4.21 stock kernel but has been patched to support DVS on the PPC 405LP. This board provides the hardware support required for DVS and allows software to scale voltage and frequency via user-defined operation points ranging from a high end of 266 MHz at 1.8V to a low end of 33 MHz at 1V [50, 6, 21]. The board has also been modified for 50% reduced capacitance, which allows DVS switches to occur more rapidly, *i.e.*, switches are bounded by at most a 200-microsecond duration from 1V to 1.8V. The DVS algorithms (static, cycle-conserving, look-ahead [53] and our feedback DVS) were exposed to the DVS capabilities of the 405LP board. The scheduling algorithms can choose any frequency/voltage pair from the set depicted in Table 8.1.

This set of pairs was constrained by a need to have a common phase lock loop (PLL) multiplier of 16 relative to the 33MHz base clock, and a divider of two or any multiple of 4. Changing the multiplier incurs additional overhead for switching, which we wanted to eliminate in this study. A dynamic power management (DPM) facility [6] is developed as an enhancement to the Linux kernel to support DVS features. DPM *operating point* defines stable frequency/voltage pairs (as well as related system parameters), which we experimentally determined.

In order to assess power consumption, we need to monitor processor core voltage and current at a high rate. Hence, we used a high-frequency analog data acquisition board to gather data for (a) the processor core voltage and (b) the processor current. The latter was measured as a voltage level over a resistor with a 1V drop per 360mA. Power consumption was computed by multiplying the CPU voltage with its current. The data acquisition board allowed us to experiment with longer-running applications to assess the energy consumption of the processor, which is the integration of power over time. We also employed an oscilloscope for visualizing the voltages and currents

Table 8.1: Valid Frequency/Voltage Pairs

| Setting | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| CPU freq. (MHz) | 33 | 44 | 66 | 133 | 266 |
| bus freq. (MHz) | 33 | 44 | 66 | 133 | 133 |
| CPU voltage (Volts) | 1.0 | 1.0 | 1.1 | 1.3 | 1.7 |

with high precision in readings.

We implemented an EDF scheduler as a user-level thread library under Linux on the 405LP board. A user-level library was chosen over a kernel-level solution because of the simplicity of its design and the fact that the operating system background activity is minimal on the embedded board infrastructure. Different DVS scheduling schemes were integrated into the EDF scheduler as independent modules.

## 8.2 Synchronous vs. Asynchronous Switch

We first assessed the overhead of different DVS techniques supported by the test board and the dynamic power management extensions of the operating system.

A unique DVS feature supported by the IBM PPC 405LP embedded board is that frequency switching can be done either synchronously or asynchronously. Synchronous switching is the traditional approach for processor frequency/voltage transitions, where applications have to stop execution during the transitional interval. Asynchronous switching, on the contrary, allows applications to continue execution during the frequency/voltage transitions. Figure 8.1 depicts the changes in current (lower curve) and voltage (upper curve) of the PPC 405LP processor core during an asynchronous switch.

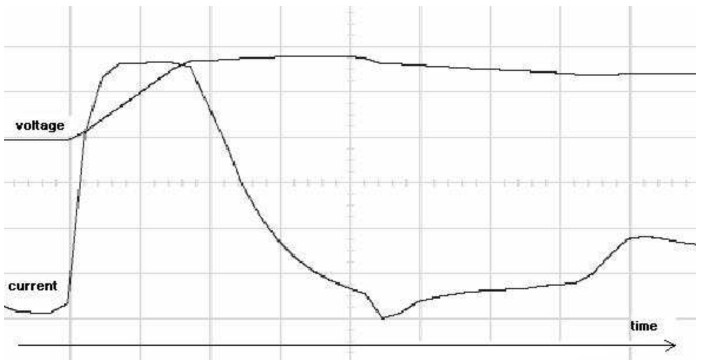


Figure 8.1: Current and Voltage Transition During Asynchronous Frequency Switching

This unique feature of asynchronous switching is achieved by a system call that, when switching to a higher voltage/frequency, first reprograms the voltage to ramp up towards the maximum as fast as possible (the 30 degree voltage ramp on the upper curve of Figure 8.1). Meanwhile, the time to reach a voltage level at least as high as required by the new frequency is estimated. A high-resolution timer is programmed to interrupt when this duration expires, prior to which the application can still continue execution. Once the timer interrupt triggers its handler (at the peak after the 30 degree ramp on the upper curve), the power management unit is reprogrammed to settle at the target voltage level, and the new processor frequency is activated before returning from the handler. The voltage then settles (in case it overshot) in a controlled manner to the new operating point. The current also settles in a controlled manner depending on processing activity.

Table 8.2 reports the overhead for synchronous and asynchronous switching in a time range bounded by two extremes: (a) Switching between adjacent frequency/voltage levels and (b) switching between the lowest and highest frequency/voltage levels. Furthermore, the overhead of the subsequent signal handler associated with each asynchronous switch is also measured for a range of the highest and the lowest processor frequencies. In order to make a comparison, the execution time of a system call getpid() is also measured. The results indicate that a synchronous DVS switch has about an order of a magnitude higher overhead than an asynchronous switch. In contrast, the asynchronous DVS switch is almost as efficient as a null system call. The timer interrupt handler triggered at each asynchronous switch has a negligibly small impact on the DVS switching operation. Overall, triggering an asynchronous DVS switch only has the cost of a light-weight system call.

Table 8.2: Frequency/Voltage Switch Overhead

| sync. switch | async. switch | signal handler | syscall |
|---|---|---|---|
| 117-162 $\mu$sec | 8-20 $\mu$sec | 0.07-0.6 $\mu$sec | 3-8 $\mu$sec |

Table 8.3: Overhead of DVS-EDF Scheduler

| | DVS scheduling overhead[$\mu$sec] | | | | | |
|---|---|---|---|---|---|---|
| CPU freq. | Static | CC | LA | feedback | | |
| | | | | simple | PID | |
| | | | | | SI | MI |
| 33 MHz | 217 | 487 | 2296 | 3207 | 3612 | 3633 |
| 44 MHz | 170 | 366 | 1714 | 2433 | 2943 | 3012 |
| 66 MHz | 100 | 232 | 1112 | 1568 | 1728 | 1739 |
| 133 MHz | 52 | 120 | 546 | 725 | 801 | 796 |
| 266 MHz | 36 | 76 | 229 | 413 | 472 | 477 |

Table 8.4: Task Set

| | Task Set 1 [$ms$] | | Task Set 2 [$ms$] | | Task Set 3 [$ms$] | |
|---|---|---|---|---|---|---|
| task | Period | WCET | Period | WCET | Period | WCET |
| 1 | 2,400 | 400 | 600 | 80 | 90 | 12 |
| 2 | 2,400 | 600 | 320 | 120 | 48 | 18 |
| 3 | 1,200 | 200 | 400 | 40 | 60 | 6 |

## 8.3 DVS Scheduler Overhead

We compared the timing overhead of our feedback-DVS algorithm with several other dynamic DVS algorithms. We first measured the execution time of these DVS scheduling algorithms under different frequencies on the embedded board, as depicted in Table 8.3. The overhead was obtained by measuring the amount of time when a task issues a yield() system call till another task was dispatched by the scheduler. The table shows that static DVS has the lowest overhead among the four while our PID-feedback DVS has the highest one. This is not surprising since static DVS uses a very simple strategy to select the frequency and voltage falling short in finding the best energy saving opportunities. Cycle-conserving (CC) DVS, look-ahead (LA) DVS and our PID-feedback DVS use more sophisticated and aggressive algorithms for lower energy consumption, albeit at higher overheads. The single-input (SI) feedback scheme and the multi-input (MI) feedback scheme have almost the same timing overhead at high frequencies, since they require constant time to update the feedback information. But the single-input scheme imposes slightly less overhead than the multi-input scheme scheme at low frequency cases.

Next, we assessed if our feedback-DVS algorithm, although incurring the largest overhead among the four, gives the best energy saving results in the real embedded environment. We measured the actual energy consumption of these DVS algorithms when executing three medium utilization task sets depicted in Table 8.4 using both synchronous and asynchronous DVS switchings. As a baseline for comparison, we also implemented a naïve DVS scheme where the maximum frequency is always chosen whenever a task is scheduled, and the minimum frequency is always chosen whenever the system is idle.

The first task set in Table 8.4 is harmonic, *i.e.*, all periods are integer multiples of the smallest period, which facilitates scheduling. This often allows scheduling algorithms to exhibit an extreme behavior, typically outperforming any other choice of periods. The second and third task sets are non-harmonic with longer and shorter periods, respectively. Actual execution times were half that of the WCET for each task for this experiment.

Table 8.5: Energy $[mW - hrs]$ consumption per RT-DVS algorithm

| algorithm | naïve | static DVS(%) | CC DVS(%) | LA DVS(%) | PID DVS(%) |
|---|---|---|---|---|---|
| **Task Set 1** | | | | | |
| syn. | 4.47 | 3.2 (**28.41%**) | 2.38 (**46.61%**) | 2.21 (**50.56%**) | 2.04 (**54.21%**) |
| asyn. | 4.43 | 3.13 (**29.35%**) | 2.327 (**47.51%**) | 2.12 (**52.07%**) | 2.00 (**54.70%**) |
| savings | 0.89% | 2.19% | 2.51% | 3.92% | 1.95% |
| **Task Set 2** | | | | | |
| syn. | 0.544 | 0.5056 (**7.06%**) | 0.4713 (**13.36%**) | 0.424 (**22.06%**) | 0.4089 (**24.83%**) |
| asyn. | 0.5276 | 0.5025 (**4.76%**) | 0.4622 (**12.40%**) | 0.4218 (**20.05%**) | 0.4064 (**22.97%**) |
| savings | 3.01% | 0.61% | 1.93% | 0.52% | 0.61% |
| **Task Set 3** | | | | | |
| syn. | 0.595 | 0.5616 (**5.61%**) | 0.4799 (**19.34%**) | 0.4043 (**32.05%**) | 0.3708 (**37.68%**) |
| asyn. | 0.5802 | 0.5496 (**5.27%**) | 0.4547 (**21.63%**) | 0.3912 (**32.57%**) | 0.3671 (**36.73%**) |
| savings | 2.49% | 2.14% | 5.25% | 3.24% | 1.00% |
| **Task Set 2 vs. Task Set 3** | | | | | |
| change | 9.07% | 8.57% | -1.65% | -7.82% | -10.71% |

Table 8.5 depicts the energy consumption in a unit of mWatt-hours. The naïve DVS algorithm serves as a base of comparisons for each of the subsequent DVS algorithms. For task set one, static DVS reduces energy consumption by about 29% over the naïve scheme. Cycle-conserving DVS saves 47% energy. Look-ahead RT-DVS saves over 50%, and our feedback method saves about 54% energy compared to naïve DVS. This clearly shows the tremendous potential in energy savings for real-time scheduling. The savings of each algorithm are lower for task set two, peaking at 23% in our feedback scheme. As mentioned before, task set one is harmonic, which typically results in the best scheduling (and energy) results since execution is more predictable. Task set three lies in between the other two with peak savings of 37% for our feedback scheme. The results also demonstrate that the overhead for calculations inherent to scheduling algorithms is outweighed by the potential for energy savings. This is underlined by the increasing overhead in execution time for each of the scheduling algorithms (from left to right in Table 8.5) while energy consumption decreases.

Another noteworthy result is the comparison between synchronous and asynchronous DVS switching depicted in the last row for each task set in Table 8.5. For each of the scheduling algorithms, we see additional savings of 1-5% on asynchronous switch due to the ability to commence with a task's execution during frequency and voltage transitions. We also ran experiments with task sets that had an order of a magnitude smaller periods and execution times. Surprisingly, the synchronous *vs.* asynchronous savings remained approximately the same, even though DVS switches occur ten times as often. We believe that the periods and execution times used in our experiments are still large compared to the execution time of a synchronous or asynchronous switch. If we only save about 100 $\mu$sec at each frequency switch (as has been shown in Table 8.2) but later on spend more then 10-100 msec in running a task, the benefit of the asynchronous DVS switching becomes insignificant. These results seem to indicate that the benefit of continuous execution during DVS switching, although not negligible, is secondary to trying to minimize the overhead of DVS scheduling itself.

We also compared task sets two and three in terms of their absolute energy readings, which is valid since they executed for the same amount of time (ten seconds), the same actual to worst-case execution time ration and the same utilization, albeit at seven times more context switches. This change is depicted in the last row of Table 8.5 for the asynchronous case. Not surprisingly, the energy with naïve DVS is about 9% higher for task set three than for set two due to the higher context switch overhead of the latter. Quite interestingly, this overhead turns into a reduction in energy as DVS schemes become more aggressive.

## 8.4 Impact of Different Workloads

We now examine the behavior of our DVS algorithm on different workloads in more detail. A suite of task sets with synthetic CPU workloads was created. Each task set contains three independent periodic tasks whose worst-case execution time varies from 0.1 to 0.9 with an increment of 0.1. The actual execution time of a task is determined by timing the body of each task plus the scheduler overhead (see Table 8.3) of the corresponding DVS algorithm under the lowest CPU frequency. We dynamically changed the number of instructions inside each task body among different invocations (jobs) to approximate the workload fluctuation behavior of actual real-time applications.

We still study the four synthesized execution patterns as introduced in Chapter 7, Figure 7.1, with the same feedback control settings. Asynchronous switching is exploited in this experiment, since it has shown better energy saving performance than the synchronous switching in previous experiments.

Figures 8.2 - 8.5 present the energy consumption of our feedback-DVS algorithm, as well as four other dynamic DVS algorithms under the four dynamic execution time patterns. Each task set contains three tasks. For pattern 1, we compare our simple feedback scheme with the multi-input feedback scheme. For dynamic pattern 2, 3 and 4, we compared our single-input feedback scheme with the multi-input feedback scheme. All energy values are normalized to the naïve DVS results under corresponding task set configurations. DR-OTE and AGR-2 dynamically reclaim unused slack up to the next arrival time of any task instance, thereby saving about 50% extra

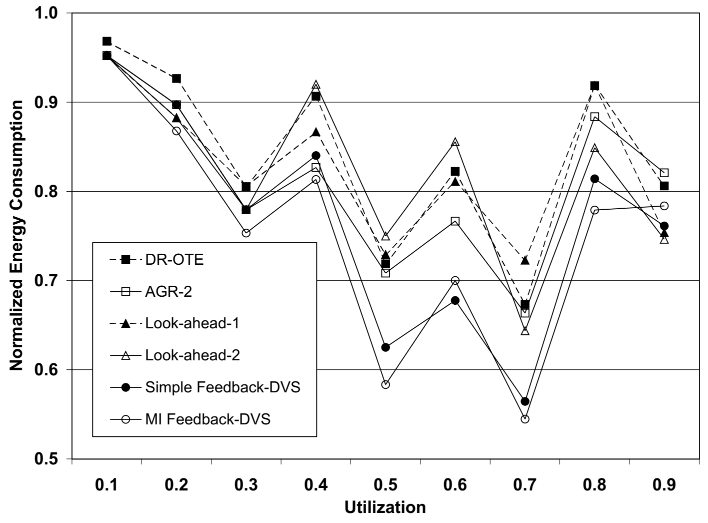


Figure 8.2: Energy Consumption for Set of 3 Tasks, Pattern 1

energy than naïve DVS. AGR-2 is not as good as Look-ahead-1/2 DVS in pattern 1 and 3, but beats Look-ahead-1/2 in pattern 2 and 4 for some cases. Look-ahead-1/2 is aggressive in frequency scaling, but it has to overcome the fact that the frequency is occasionally lowered too aggressively so that it has to be subsequently raised to a high level. We avoid such behavior in our algorithm *via* feedback.

In Figure 8.2, our simple feedback scheme performs almost as well as the multi-input feedback scheme. The difference of normalized energy between our algorithm and others ranges from an additional 5% to 15% energy savings over the best scheme published previously. Considering the low overhead of the simple feedback scheme, it is a good choice for tasks whose execution time does not vary over multiple instances.

In Figures 8.3, 8.4 and 8.5, our single-input and multi-input feedback schemes save an additional 5%-18% energy over other schemes due to the algorithm's self-adaptation to a job's actual execution time. Single-input feedback performs slightly worse than the multi-input feedback scheme, because its modeling method is not as precise as the multi-input feedback scheme. But the energy consumption differs by about 4% for all cases between these two schemes. There are even cases, *e.g.*, at 0.6 utilization in Figure 8.4, where single-input feedback outperforms the multi-input feedback. In extremely low or extremely high task utilization cases, our feedback-

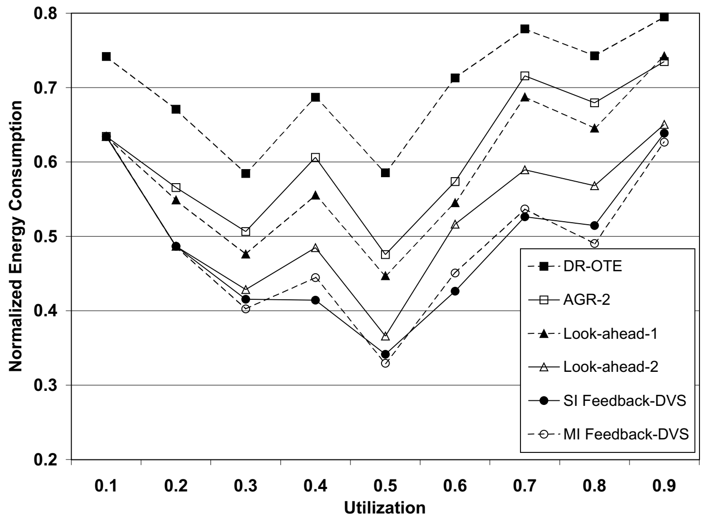


Figure 8.3: Energy Consumption for Set of 3 Tasks, Pattern 2

DVS, Look-ahead DVS and AGR algorithm result in comparable energy consumption. In these cases, tasks either have enough slack to always run at the minimum speed, or they do not have slack at all preventing them to lower their speed. This results in virtually the same frequency choices for a schedule irrespective of the DVS algorithm used.

To better assess the scalability of our feedback-DVS algorithm, we further ran two experiments. One experiment increases the number of tasks in the task set from 3 to 30, as shown in Figure 8.6. AGR-2 benefits from such a smaller task granularity in 30-task sets and outperforms Look-ahead-1 and Look-ahead-2 in some utilization cases. The small task granularity also reduces the gap between our feedback DVS algorithms and the other algorithms. But we still save around 3% to 8% additional energy than others. The second experiment fixes the execution-time pattern of the task set, while varying the baseline (the average execution time) of different task instances, as shown in Figure 8.7. The average execution time in Figure 8.7 is set as 75%WCET, 50%WCET and 30%WCET, respectively. All energy values are normalized to the naïve DVS values. We see from these figures that our single-input feedback scheme scales equally well for loose (0.3WCET case) and tight (0.75WCET case) actual execution times. In all three cases, 14% to 24% additional energy is saved

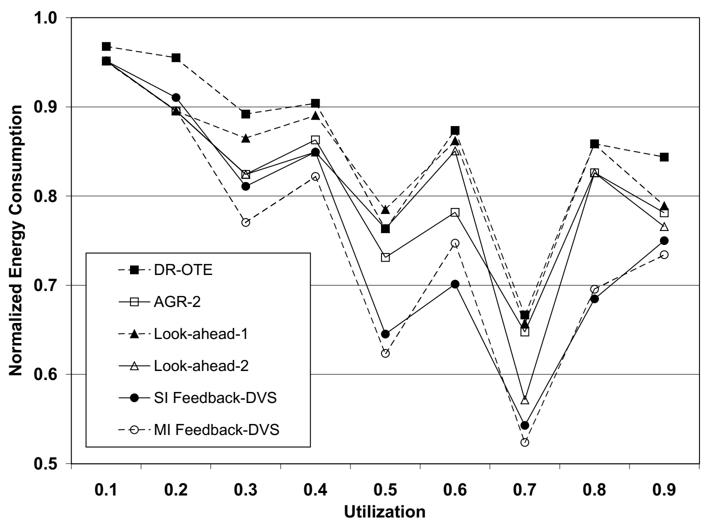


Figure 8.4: Energy Consumption for Set of 3 Tasks, Pattern 3

over look-ahead-2 DVS. The feedback schemes show larger improvements for median execution times than the loose or tight ones. In this range, there is enough slack to distinguish itself from the other algorithms. When comparing our feedback algorithm with the naïve DVS scheme, we observe even more significant energy savings. The largest saving is shown in Figure 8.6, where up to 70% additional savings are achieved by our algorithm over the naïve one at the 0.3 utilization case.

We also visualized voltage and current switches using an oscilloscope. Figures 8.8 and 8.9 depict the screen-shots of voltage and current obtained from the oscilloscope for the phase just after a simultaneous release of all tasks at the beginning of a hyperperiod. In Figure 8.8, every task has a loose WCET, which is two times of its actual execution time. In Figure 8.9, a tight WCET equal to a task's actual execution time is used. Static DVS shows two levels of voltages (busy/idle time) whereas cycle-conserving DVS differentiates three levels on a dynamic base. Even lower voltage and current readings are given by look-ahead DVS, which not only distinguishes more levels but also exhibits much lower power levels during load. The lowest results were obtained by our feedback DVS, which defers execution even more aggressively than any of the other methods. However, our feedback scheme can only reduce power consumption occasionally as sufficient static or dynamic slack exists to be reclaimed.

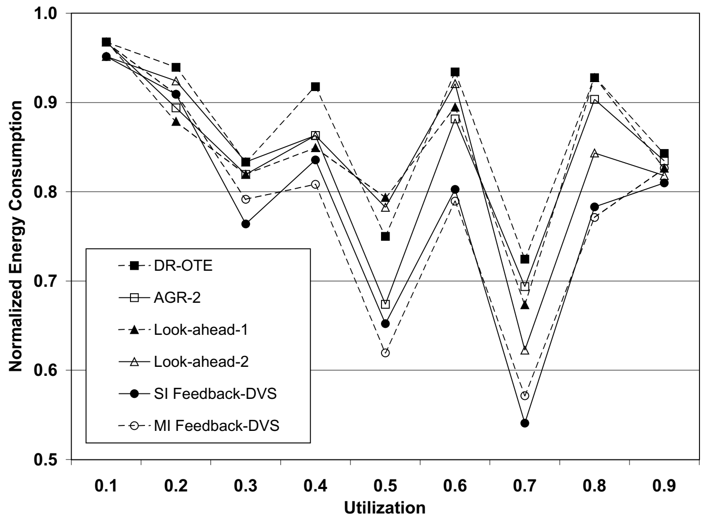


Figure 8.5: Energy Consumption for Set of 3 Tasks, Pattern 4

Dynamic slack is recovered in increasing orders by the latter three schemes.

## 8.5 Comparison with Simulation Results

When we compare the energy saving results obtained from the IBM 405LP embedded board with our previous simulation results presented in Chapter 7, we clearly see the advantage and disadvantage of simulation for power-aware studies. The advantage of simulation lies in its ease of implementation and predictability of performance trends. The energy consumption of different DVS algorithms show a consistent trend under both simulation and the actual embedded platform. But the quantitative results differ. Our previous simulation results reported 5%-10% higher savings on average. For example, the best energy saving of our feedback DVS over look-ahead DVS was reported as 29% in simulation while the best result we measured from the test board is around 24%. It is non-trivial to model the actual power/energy consumption in simulation without considering actual hardware details. This is also the case when evaluating the overhead. Since the overhead of DVS algorithms was not included in our previous simulation experiment, we still observed 7%-10% energy savings over look-ahead DVS even at high utilization cases. But the actual energy measurements

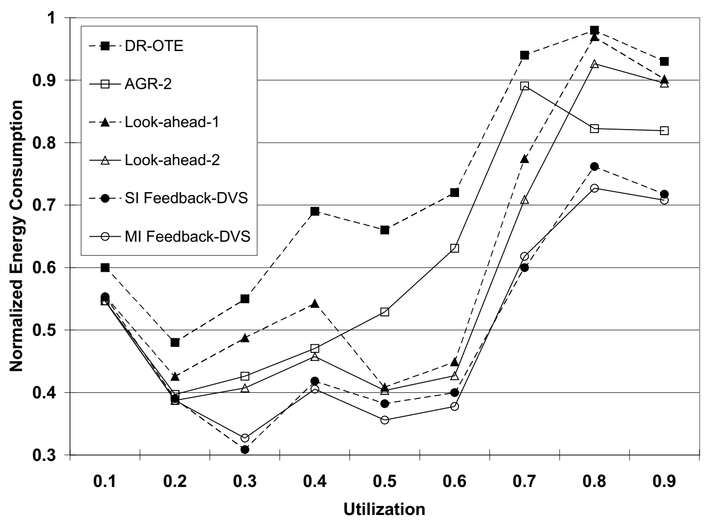


Figure 8.6: Energy Consumption for Set of 30 Tasks, Pattern 2

from the test board show only 3%-6% savings for these cases.

Overall, our experiments on the embedded platform quantitatively show the potential of our feedback DVS algorithm and its ability to scale power even more aggressively than previous DVS algorithms.

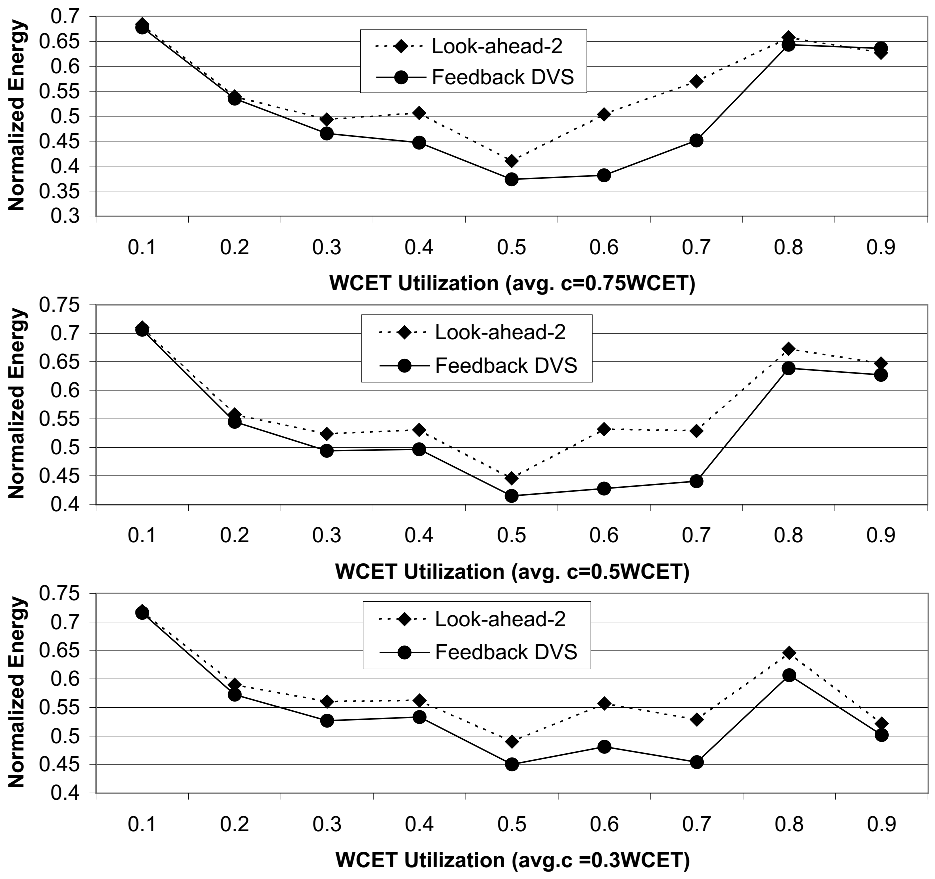
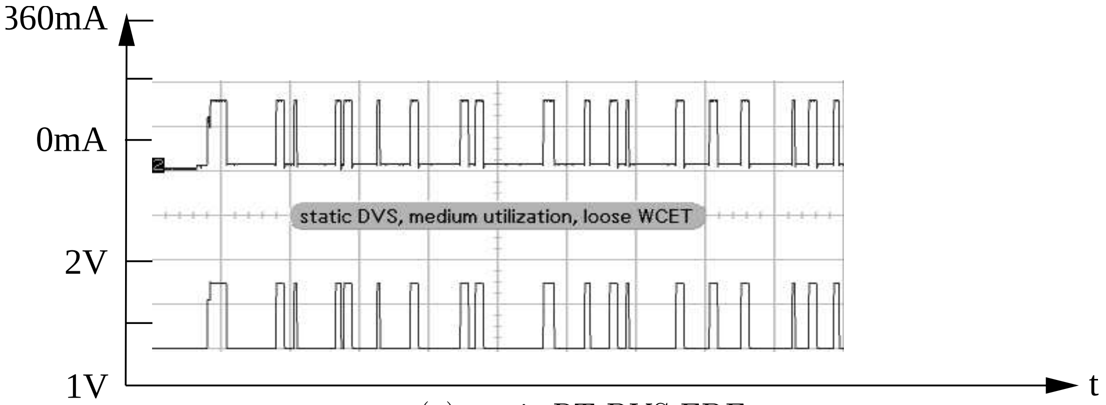
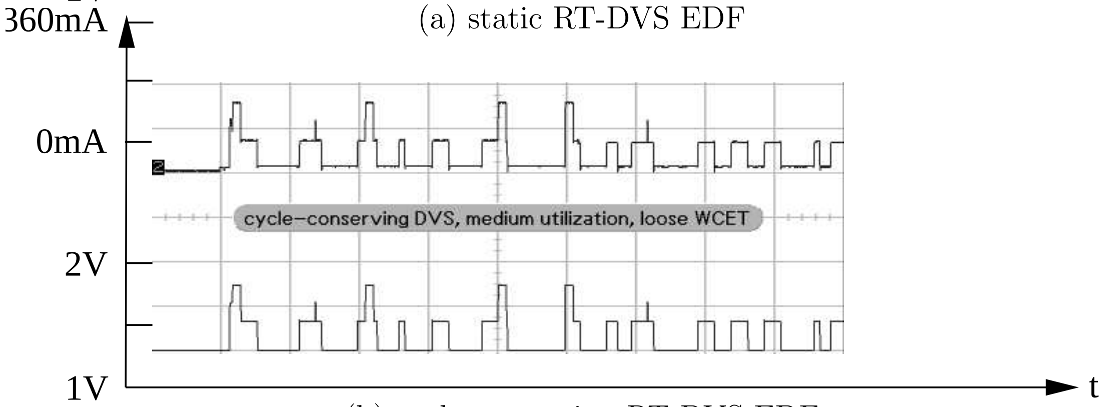


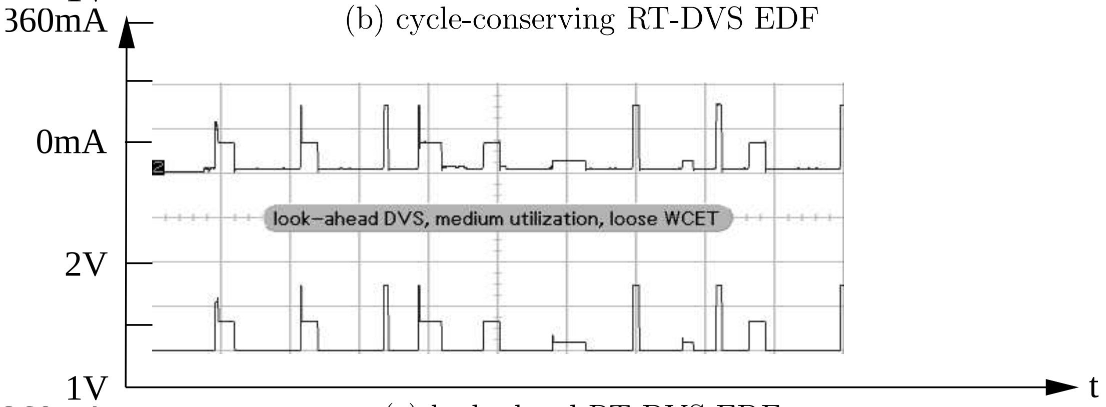
Figure 8.7: Energy Consumption for Set of 3 Tasks, Pattern 4, with Varying Baseline



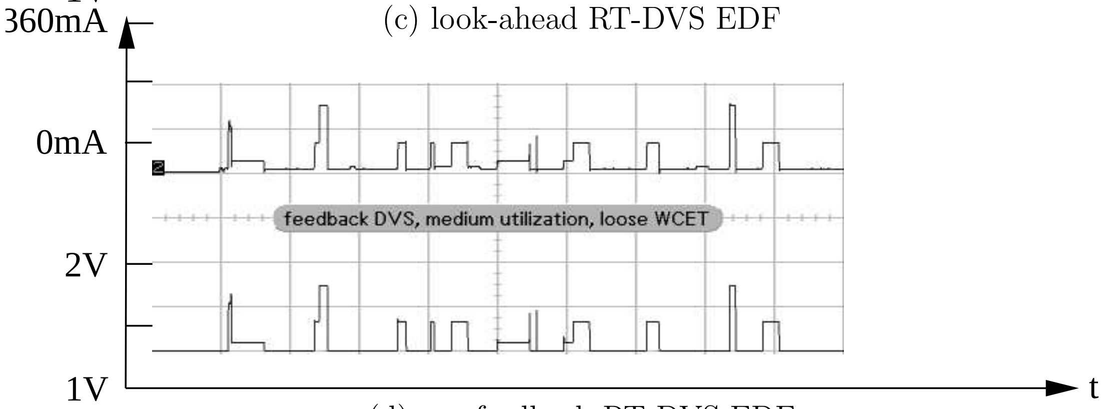
(a) static RT-DVS EDF



(b) cycle-conserving RT-DVS EDF

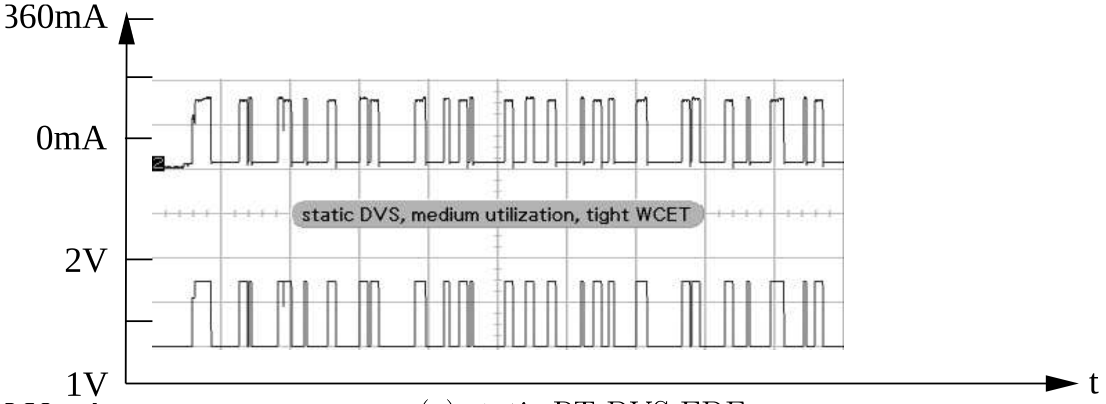


(c) look-ahead RT-DVS EDF

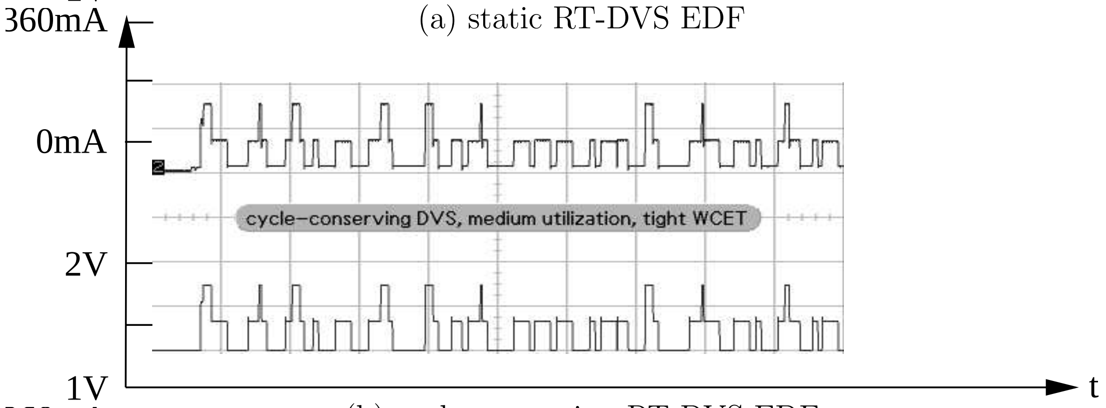


(d) our feedback RT-DVS EDF

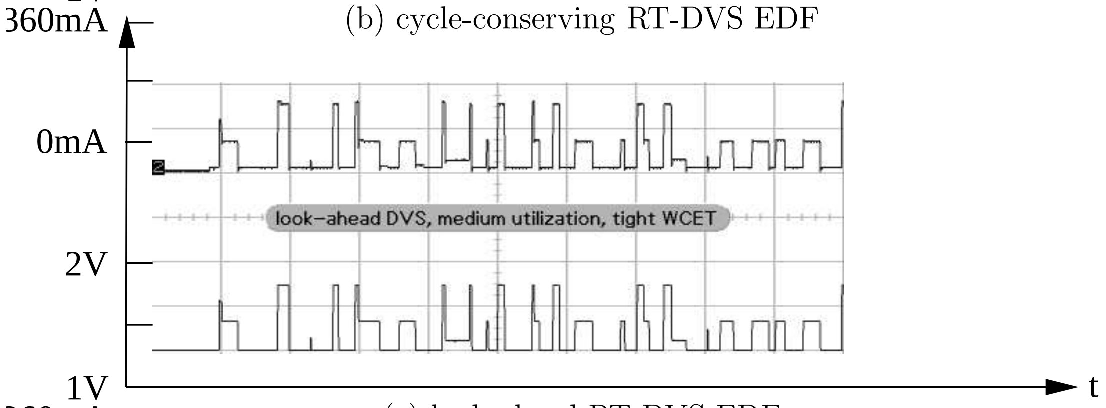
Figure 8.8: Voltage/Current Oscilloscope Shot, Loose WCET= 2× ActualExecTime, U=0.5



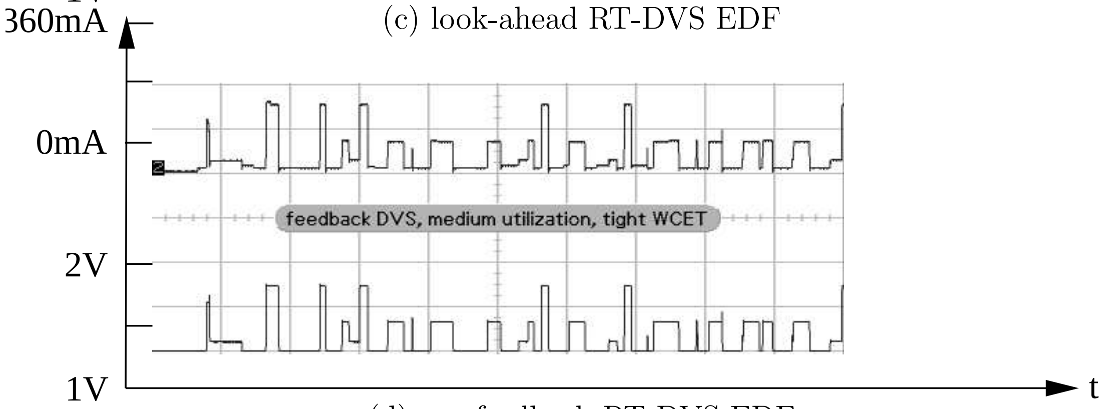
(a) static RT-DVS EDF



(b) cycle-conserving RT-DVS EDF



(c) look-ahead RT-DVS EDF



(d) our feedback RT-DVS EDF

Figure 8.9: Voltage/Current Oscilloscope Shot, Tight WCET= ActualExecTime, U=0.5

# Chapter 9

# Leakage-Aware Feedback-DVS

In this chapter, we extend our feedback-DVS scheme considering not only the dynamic power consumption but also the static power consumption, which is caused by leakage current existing in a CMOS-based circuit.

## 9.1 Motivation

Power consumption in a CMOS-based processor consists of three elements, *i.e.*, dynamic, static, and short-circuit power [45]. Static power consumption stems from leakage current that exists even in the absence of logic operations of a circuit. The Feedback-DVS scheme presented in previous chapters works well when dynamic power dominates in the CMOS processors, while static and short-circuit power can simply be ignored. This is also the assumption in most of the previous dynamic voltage scaling work [71, 53, 4, 29, 13, 16, 52, 69]. Although this is true on traditional CMOS-based circuits, it is not true anymore for some of the contemporary and future processors when we consider the trends of CMOS circuit technologies. For example, the sub-threshold leakage current, which contributes to the static power consumption , is $0.01\mu A/\mu m$ for the $130nm$ technology and is anticipated to be $3\mu A/\mu m$ for the $45nm$ technology [28]. A five-fold increase in the leakage power is estimated with each technology generation [5]. Static power is not ignorable and is expected to be comparable with dynamic power. When the voltage supplied to a CMOS-based processor is reduced below a certain threshold value, static power exceeds dynamic power and becomes the dominant cause of power dissipation *per se*. The processor frequency associated with this threshold voltage is called the *critical speed*. Above the threshold voltage, the total energy per cycle increases as the processor voltage scales up. But below the threshold voltage, the total energy per cycle *also increases* as the voltage scales down, even through only *lower frequencies* with lower performance can be sustained. This result leads us to re-consider two issues in the design of a DVS algorithm:

1. It is not energy-efficient to scale down processor voltage and frequency to an extremely low level, if that level is below the threshold value / below the critical speed.

2. Because of the existence of leakage power consumption, forcing the processor into sleep mode may be more energy-efficient than keeping the system idle at a low frequency as long as the idle period is long enough to compensate for the shutdown overhead.

Any combined DVS/leakage policy has to take into account the above issues and makes decisions according to the actual power consumption characteristics. These issues have been addressed in previous work where both static and dynamic power consumption are reduced [23, 34, 28]. These approaches either assume that all tasks are running at the same speed to conserve static power. Or, they use off-line schemes without fully exploiting the power saving potentials. Lee *et al.* [33] used a greedy method to locally maximize the duration of alternating idle and busy periods based on the worst-case execution time. Since actual execution times often diverge considerably from the WCET, a conceptually busy period is actually interspersed with dynamic slack due to early completion of jobs. The potential of dynamic slack remains unused. Jejurikar *et al.* [28] assume that a power manager, implemented as a hardware controller, handles interrupts and timers when new tasks are released. In contrast, our scheme does not require any special hardware support beyond DVS and sleep modes, nor does it assume execution times equal to their worst-case bounds.

In the following, we present an on-line combined DVS/leakage control scheme that saves both static and dynamic power. This scheme profits from our feedback-DVS algorithm that exploits a modified earliest-deadline-first (EDF) scheduling variant. It automatically chooses between voltage scaling and a processor sleep mode according to the run-time execution scenario of tasks. Voltage scaling is used when dynamic power dominates the total power consumption. A processor sleep mode is entered when static power dominates the total power consumption. Our scheme also locally adjusts the dispatch time of a task so that adjacent tasks are either bundled together or scattered apart to increase the opportunity of getting into the sleep mode.

## 9.2 Power Model

We use the power model of a CMOS circuit first presented by Martin *et al.* [45]. The power consumed in a processor consists of three portions: dynamic power $P_{AC}$, static power $P_{DC}$, and short-circuit power. Short-circuit power is only consumed during signal transitions and, in practice, is generally negligible [45]. Similar power models are also used in related work [28, 58]. Hence, we only consider static and dynamic power in our model.

Dynamic power is given by:

$$P_{AC} = C_{eff} V_{dd}^2 f \tag{9.1}$$

where $C_{eff}$ is the average switched capacitance per cycle, $V_{dd}$ is the supply voltage, and $f$ is the processor clock frequency. Static power consumption is given by:

$$P_{DC} = V_{dd} I_{subn} + |V_{bs}|(I_{jn} + Ibn) \tag{9.2}$$

where $I_{subn}$ is the sub-threshold leakage current, $I_{jn}$ and $I_{bn}$ are the drain and source to body junction leakage currents.

Static and dynamic power can be traded-off against each other in practice. It has been shown that there exists a threshold voltage $V_{th}$, below which it is no longer energy efficient, *i.e.*, the processor voltage should not be scaled below this threshold value [28]. From the threshold voltage $V_{th}$, one can derive a corresponding threshold frequency $f_{th}$, the *critical speed*. Instead of operating at a speed below the threshold value, it is more power efficient to execute tasks at or above the critical speed.

The transition into and out of a sleep mode does not come without cost. Such a transition incurs additional energy consumption, termed sleep overhead from here on. This overhead is mostly due to warm-up of resources (particularly caches) when resuming execution. Hence, sleeping is only a viable option when the energy saved by sleeping exceeds that of the sleep overhead itself.

In the following, we assume a deep sleep mode during which only the interrupt line of a processor remains receptive. Other parts of the processor, including caches, are turned off and will loose their state. In our model, we assume that the processor consumes a negligible amount of energy when in sleep mode. Power consumption in the sleep mode is documented as being three orders of a magnitude lower than the power consumption in an active mode[22].

Transitioning into and out of a sleep mode incurs, as a side-effect, cold misses in cache among other resource refresh overheads. Let $E_{sd}$ be the additional energy per sleep overhead. Although entering and exiting the sleep mode also take a small amount of time, we ignore it here since it can be incorporated into our sleep threshold derived below.

Let $p_{idle}$ be the power consumption when the system is idle. Then, $t_{th} = E_{sd}/p_{idle}$ defines a sleep threshold. It is energy efficient to enter sleep mode if and only if the slack time in the schedule exceeds $t_{th}$. Otherwise, the processor should remain idle at a power-efficient DVS level. These parameters are platform dependent but are available to the scheduler at system initialization.

In the following section, we describe the DVSleak algorithm, which is integrated into the task scheduler and contains policies for reducing both static and dynamic power.

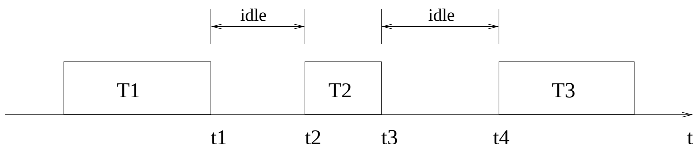
## 9.3 DVSleak Algorithm

To make the DVS algorithm leakage aware, our feedback-DVS scheme takes into account the impact of static power as well as the threshold voltage to consider the effect of static power. A naïve scheme is to mark all voltage and frequency levels below the threshold as invalid, so that whenever the DVS algorithm wants to assign a speed below that threshold, it uses the threshold value instead. A task then runs at a higher speed than its original assignment. It completes earlier providing more slack (idle time) prior to its deadline. As long as the slack is long enough to compensate for the shutdown overhead, the DVS scheduler can put the processor into a sleep mode during that interval to further reduce the impact of the static power consumption.

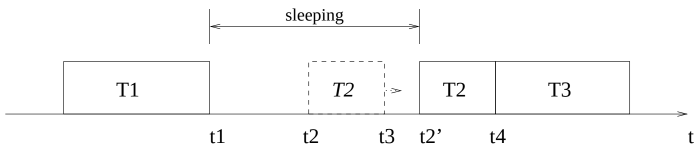
Unfortunately, such a naïve scheme does not fully exploit the energy saving potential. For example, consider the three adjacent tasks depicted in Figure 9.1(a). Task $T_1$ completes at time $t_1$. Task $T_2$ is released at time $t_2$ and completes at time $t_3$. Task $T_3$ is released at time $t_4$. Let the lengths of both idle intervals $[t_1, t_2]$ and $[t_3, t_4]$ be less than the threshold $t_{th}$. Hence, the processor is kept in an idle state during the above two intervals instead of entering a sleep mode. Both static and dynamic power consumption exist in the idle state. The processor energy consumption in an idle state, although lower than the energy consumption in a non-idle running state, is still significantly larger than the energy consumed in a sleep mode.

In order to further exploit the savings for both static and dynamic power, we adapt the schedule of the system to reduce static leakage as much as possible. Consider shifting task $T_2$ to line up with the release time of $T_3$ in Figure 9.1(b). $T_2$ is now released at time $t2'$. The interval $[t_1, t_2']$ then exceeds the sleep threshold value $t_{th}$ so that the processor enters a sleep state during that interval. Static power is almost eliminated while sleeping. The only energy consumption the processor pays is dynamic power as well as the sleep overhead. Figure 9.1(b) is the ideal case where $T_2$ completes exactly before the release of $T_3$, thus maximizing the processor sleep period. Even if $T_2$ takes less cycles than expected and completes earlier, delaying the release time of $T_2$ costs less energy than the non-delay policy. As shown in Figure 9.1(c), if $T_2$ completes earlier, the processor enters the idle state till the release time of $T_3$. The energy saved in $[t_1, t_2']$ due to sleeping makes the delay policy superior to the non-delay schedule, as shown in Figure 9.1(a).

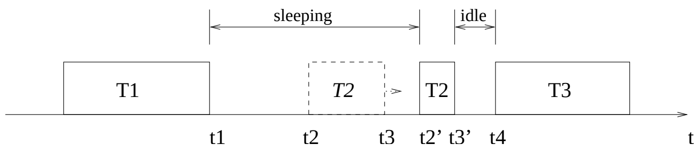
The above example illustrates the benefit of the delay policy in terms of reduced leakage in a DVS-aware system. In the following, we present an algorithm that



(a) DVS without Delay Policy



(b) DVS with Delay Policy using WCET



(c) DVS with Delay Policy using Actual Execution Time

Figure 9.1: Combining DVS and Leakage Savings

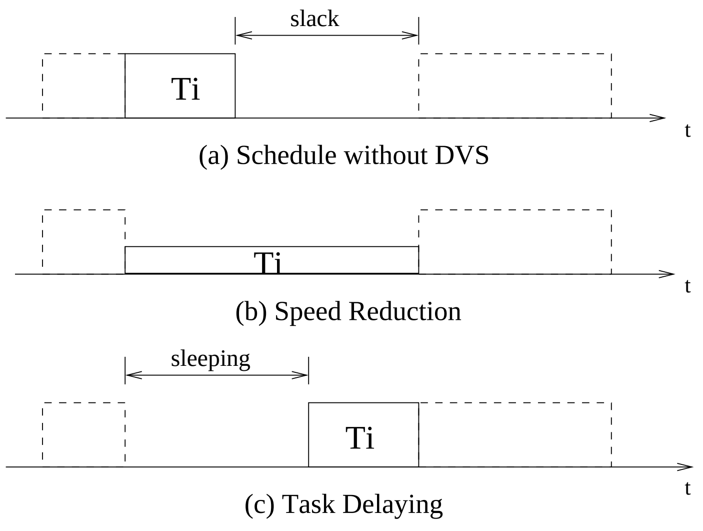


Figure 9.2: Speed Reduction vs. Task Delaying

combines this delay policy with dynamic slack reclamation and feedback of actual execution times.

### 9.3.1 Speed Reduction vs. Task Delaying

DVS technology modulates the processor speed according to the amount of slack or idle time in the schedule. A dynamic voltage scaling algorithm, when integrated with leakage power saving schemes, needs to address two issues. First, it needs to determine how to distribute the amount of slack between speed reduction and task delaying. Second, it needs to decide if the release time of a job should be delayed. This section focuses on the first issue while the next section addresses the second one.

Consider the example in Figure 9.2(a). Lowering the processor voltage and frequency, *i.e.*, reducing the application speed, decreases the amount of slack available in the schedule, as depicted in Figure 9.2(b). Similarly, delaying the activation time of a task by putting the processor into a sleep mode also decreases the amount of slack, as depicted in Figure 9.2(c). At any point of time during execution, the amount of slack is always a shared resource between these two competing operations. The DVS algorithm has to define a policy to determine the distribution of the slack between these two schemes.

This dilemma can be solved based on the critical speed (frequency). We prefer a lower frequency over delaying a task as long as the resulting frequency is higher than the critical speed. Conversely, when our frequency scaling scheme suggests a speed lower than the critical speed, we default to the critical speed and activate the task delaying scheme. This policy reflects a best effort to reduce power as much as possible.

According to the above analysis, whenever a task completes and a new task $T_i$ is released, our DVS algorithm uses a feedback-EDF scheme to calculate a frequency level $f_i$. The actual frequency $f$ assigned to task $T_i$ is defined by:

$$f = min(f_i, f_{th}) \tag{9.3}$$

Given the actual frequency of task $T_i$, a corresponding voltage can also be determined. But before task $T_i$ is released, the DVS scheduler has to decide whether or not the release time of the task needs to be delayed. This issue is detailed in the following section.

### 9.3.2 Delay Policy

The example in the motivation section seems to imply that a task should always be delayed as much as possible against its deadline. This is also the strategy used in previous work [28, 58]. Such an intuitive approach, however, is not always the best solution. This is due to the variability of the actual execution time of tasks. Figure 9.1(b) shows the case where the execution time of task $T_2$ equals its worst-case execution time. In the real world, the actual execution time of a task is generally shorter than its worst-case execution time.

A schedule without delaying $T_i$'s release time leaves the processor idle in the beginning. Some time later the processor enters a sleep mode, as shown in Figure 9.3(a).

Figure 9.3(b) depicts the effect of a delayed schedule, where the processor enters a sleep mode first and later on incurs a potentially longer idle period, thereby consuming more power than case (a). This effect is due to the delay policy, which relies on the

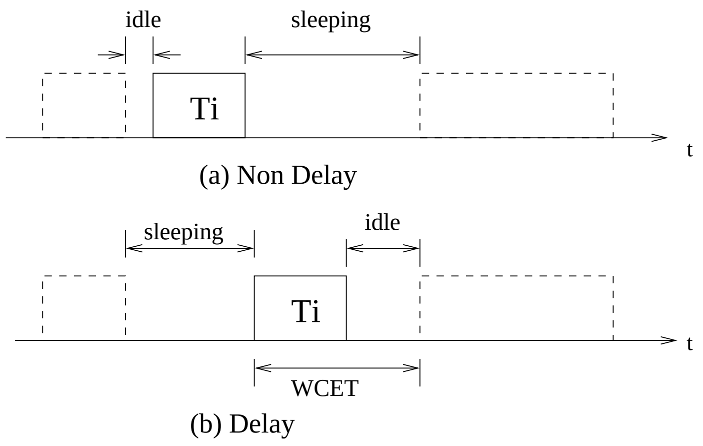


Figure 9.3: Delay vs. Non-delay

WCET instead of the actual execution time of $T_i$ to determine the delay time. When a task completes earlier than expected, it produces additional dynamic slack, which significantly reduces the benefit of the delay policy.

Taking this short-coming into consideration, we present the following delay policy as part of our DVSleak algorithm. We observe that at any time $t$, the DVS algorithm can infer the amount of slack $s_t$ in the schedule. If the ready queue of the scheduler is not empty, the delay policy remains inactive. As shown in Figure 9.4, the next task $T_i$ will be released at time $t_r$ $(t_r \geq t)$ according to the standard EDF scheduling algorithm. With the knowledge of $s_t$, the feedback-DVS algorithm assigns a processor frequency $f_A$ and a scaling factor $\alpha_A$, as defined in Equation 4.1, for $T_A$, which is the first subtask of $T_i$ in the task splitting scheme. Since the number of execution cycles of $T_i$ is also split into two parts, we have

$$\frac{C_i^A}{\alpha_A} + C_i^B = \frac{C_i}{\alpha_i} \tag{9.4}$$

where $\alpha_i$ is a unified scaling factor of the entire task (as if the task had not been split). By introducing $\alpha_i$, the delay policy of the following task can be easily integrated into any DVS algorithms. From Equation 9.4, we derive $\alpha_i$:

$$\alpha_i = \frac{C_i \alpha_A}{C_i^A + C_i^B \alpha_A} \tag{9.5}$$

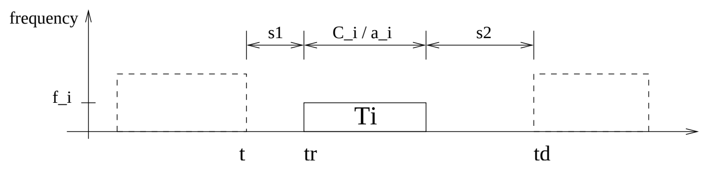


Figure 9.4: Rules for Task Delaying

Let $c_i$ be the expected actual execution time of $T_i$ provided by a feedback scheme based on the execution times of previous instances of task $T_i$. The latest time that $T_i$ can complete without missing its deadline is given by:

$$t_d = t + s_t + C_i \tag{9.6}$$

where $C_i$ is the WCET of $T_i$. Time $t_d$ can also be represented as the minimum of the absolute deadline of the task and the release time of the next task in EDF after $T_i$, *i.e.*,

$$t_d = min(d_i, t_{r_{i+1}}) \tag{9.7}$$

Notice that if the next task is released together with $T_i$, there will only be one idle period prior to $T_i$.

We use the following rule to determine the modified release time of $T_i$.

1. Task $T_i$ is released at time $t_r$ (as under standard EDF) if and only if

   (a) $t_d - t - C_i/\alpha_i \leq t_{th}$, or,

   (b) $t_d - t - C_i/\alpha_i > t_{th}$ and $t_r - t < t_{th}$ and $C_i/\alpha_i - c_i \geq t_r - t$.

2. Task $T_i$ is released at time $t_d - C_i/\alpha_i$ (later than under standard EDF) if and only if

   (a) $t_d - t - C_i/\alpha_i > t_{th}$ and $t_r - t \geq t_{th}$, or,

   (b) $t_d - t - C_i/\alpha_i > t_{th}$ and $t_r - t < t_{th}$ and $C_i/\alpha_i - c_i < t_r - t$.

Rule 1 covers the cases where the release time of task $T_i$ is not delayed. Conversely, Rule 2 captures the cases where it should be delayed. Rule 1(a) applies when the total amount of slack time in $[t, t_d]$ (equivalent to $s_1 + s_2$ in Figure 9.4) is less than the sleep threshold $t_{th}$. Task $T_i$ is not delayed since there is not enough slack to benefit from sleeping, regardless of whether or not the task is delayed. Rule 2(a) applies when the total amount of slack is greater than the sleep threshold $t_{th}$ and the initial slack $s_1$ is at least as large as this threshold, which ensures that sleeping will be beneficial. By delaying $T_i$'s release time to $t_d - C_i/\alpha_i$, we increase the amount of slack prior to T's execution as much as possible to prolong the initial sleep duration.

Rule 1(b) and Rule 2(b) capture cases where the length of the first slack $s_1$ is less than the threshold $t_{th}$ while the overall slack $s_1 + s_2$ exceeds this threshold. In these cases, delaying the release time of task $T_i$ does not always result in the longest sleep duration. Figure 9.3(a) and (b) illustrates the best efforts reflected by Rules 1(b) and 2(b), respectively. The decision is, in fact, based on the anticipated portion of unused execution time (WCET - actual execution time). If this portion is equal or larger than slack $s_1$, it is beneficial to accumulate more slack (due to early completion within $C_i/\alpha_i - c_i$) with $s_2$, which does not require the task to be delayed, as reflected in Rule 1(b). Conversely, if the unused portion is less than slack $s_1$, late slack ($s_2$) is merged with early slack ($s_1$) by shifting the execution of T to the latest possible point in time, which lengthens the beneficial sleep duration prior to the shifted task, as reflected in Rule 2(b). This heuristic approach is relatively simple but still yields promising results. Notice that $c_i$, the expected actual execution time of task $T_i$, is provided by the feedback controller according to previous execution history.

We combine this task delay policy with the existing DVS algorithm. By enhancing the algorithm with the delay policy, we still guarantee the feasibility of the schedule for the task set, as stated by the following theorem.

**Theorem 2.** *If a feasible schedule exists for a task set under EDF scheduling, the modified schedule after applying the delay Rules 1 and 2 is guaranteed to be feasible as well.*

*Proof.* For any task $T_i$ in the task set, let $d_i$ be its absolute deadline. If T meets its deadline under the EDF, then its release time $t_r$ satisfies:

$$t_r + C_i + s_t \leq d_i \tag{9.8}$$

According to the above relationship and Equation 9.6, we know that $t_d \leq d_i$. Delay Rules 1 and 2 either release $T_i$ at its original EDF time $t_r$ or at time $t'_r = t_d - C_i/\alpha_i$. In the former case, $T_i$ will not miss its deadline since $T_i$ is scheduled as in conventional EDF. In the later case, let $T_i$'s worst case execution time after frequency scaling be $C'_i$. Then, $C'_i = C_i/\alpha_i$. In the worst case, $T_i$ will complete before

$$t'_r + C'_i = t_d - C_i/f_t + C_i/\alpha_i = t_d \leq d_i \tag{9.9}$$

Since it will be activated at its new release time $t'_r$ and no other tasks are ready in $[t, t_d]$ due to Equation 9.7. Hence, $T_i$ again completes before its deadline and the task set can still be feasibly scheduled. □

## 9.4 Simulation Experiment

We implemented the above leakage-aware DVSleak algorithm in a simulation environment, using the power model described in Section 9.2. We assume the processor has four discrete frequency levels, which are 25% , 50%, 75% and 100% of $f_m$, which is the maximal frequency supported by the processor. We use the same approach as in [28] to compute the corresponding power consumption as $550mW$, $650mW$, $990mW$ and $1480mW$ for frequency levels 25%, 50%, 75% and 100%, respectively. The processor enters an idle frequency whenever none of the tasks are ready. As in [28, 58], the idle power consumption is assumed to be 240mW, $E_{sd}$ is at $483\mu J$, and $t_{th}$ is $2ms$. The threshold frequency level is set to 41% of $f_m$.

In order to assess the energy saving potential of our combined leakage-aware DVSleak algorithm, three different algorithms are implemented in the simulator.

1. A pure feedback-DVS algorithm without any leakage power saving schemes. The algorithm does not observe trade-offs due to the threshold frequency, *i.e.*, the frequency is freely scaled below this threshold.

2. A feedback-DVS algorithm with a sleep policy. This algorithm puts the processor into sleep mode whenever the idle slack in the schedule is longer than the sleeping threshold. The algorithm exploits the threshold frequency in that no

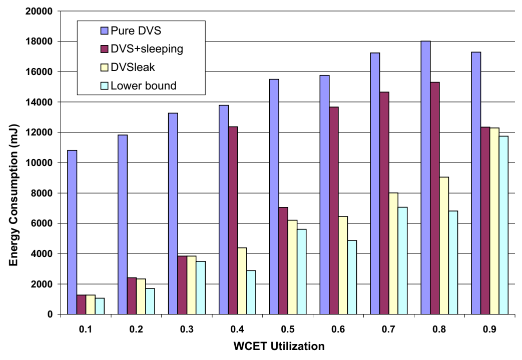
tasks will be scaled below that frequency. Hence, a frequency of 25% of $f_m$ will never be used. However, this algorithm does not contain any delay policy to postpone the release of a task.

3. DVSleak, a feedback-DVS algorithm with delayed sleep policy, as outlined in the last section. This algorithm is the most aggressive one. It not only puts the processor into a sleep mode, it further delays the release time of tasks according to our delay rules. The delay rules increase the length of the sleep duration, which saves more energy than other algorithms. Our experimental results show that this is mostly (but not always) the case. DVSleak also exploits knowledge about the threshold frequency.

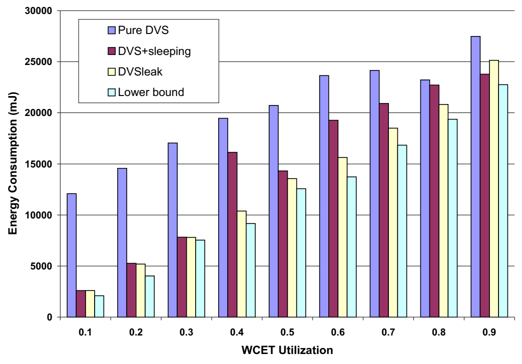
We use the same task sets as described in Chapter 7, Figure 7.1. For each execution pattern, the task sets' WCETs were uniformly distributed in the range [10,1000]. When tasks' WCETs were generated, each task's period was chosen so that the worst case utilization of the task set varies from 0.1 to 1.0 in increments of 0.1.

In order to make a comparison, we also calculate a lower bound on energy for each utilization case. In the lower bound schedule, the entire task set runs at either the ideal optimal speed or the critical speed, whichever the greater. The number of times the processor gets into the sleeping or idle state is also minimized. We assign a longest busy interval for the task set which equals the maximal response time of the task with the longest period. Such assumptions make it possible to derive a lower bound energy overhead for processor state transitions.

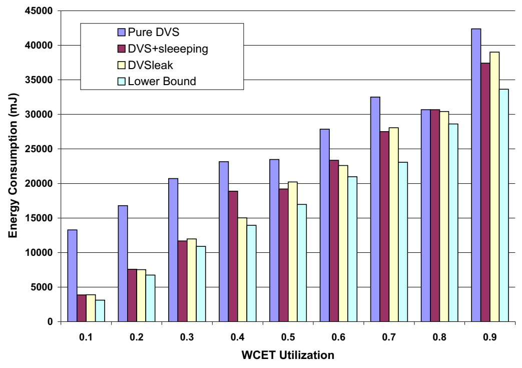
Figure 9.5 depicts the energy consumption of the three different algorithms with execution pattern one, as well as the lower bound energy consumption for each utilization case. We see significant energy savings at low utilization because of the existence of large amounts of slack. Putting the processor into sleep mode saves as much as 80% more energy than the pure DVS algorithm, which only lets the processor idle during the entire slack period, sacrificing both dynamic and static energy. When the utilization increases to 0.6 and larger, the sleep policy alone is not attractive since there is not enough *ad-hoc* slack in the schedule anymore. On average, only 10% more energy is saved over pure DVS. DVSleak with its combined sleep and delay policy,



(a) c=25%WCET



(b) c=50%WCET



(c) c=75%WCET

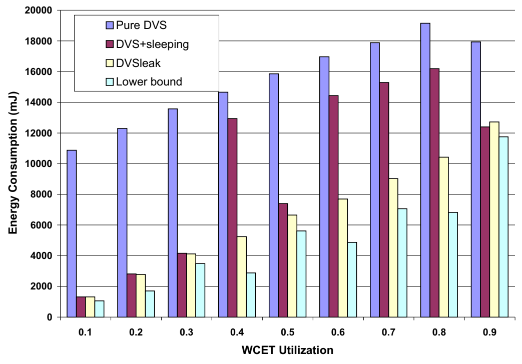
Figure 9.5: Energy Savings for 3 Tasks, Pattern One under Different Actual Execution Times (Constant) and Utilization

in contrast, shows its strength by saving 50% more energy on average than the pure DVS scheme and 40% more energy on average than the pure sleep policy. It is interesting to note that the delay policy performs at par with the sleeping policy for several cases, such as the 0.1 and 0.2 utilization cases. Figure 9.5(b) and 9.5(c) also show that the delay policy even costs more energy than the non-delay policy at 0.9 utilization, where already limited slack is further reduced by the delay policy, which results in higher processor frequencies than that of a pure sleep policy.

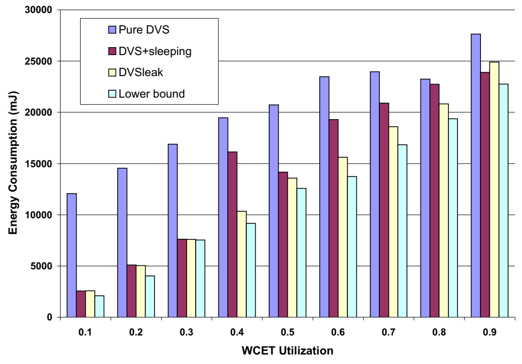
Figure 9.6 depicts the energy consumption of these three DVS algorithms under execution pattern two. In contrast to pattern one, execution times vary dynamically among different jobs, which results in higher energy consumption than pattern one in corresponding cases. When we look at the energy savings of the three DVS algorithms, we see that DVSleak again shows its advantage under medium and high utilization. Even with varying workloads, the delay policy generates more opportunities for sleeping than any of the other policies. It saves 40% more energy on average than the pure DVS algorithm. For both execution patterns, the energy consumption produced by DVSleak is also very close to the lower bound in most of the utilization cases.

We further increase the number of tasks in a task set from 3 to 10. The increase in number of tasks limits the effectiveness of the delay policy. It cannot produce sufficiently long intervals to benefit from sleeping. Nonetheless, Figure 9.7 illustrates that DVSleak exhibits stable savings in energy irrespective of the number of tasks. It achieves almost the same amount of savings over the pure DVS as observed for 3 tasks. The energy saving over the pure sleeping algorithm is not as significant as that of 3 tasks. Still, DVSleak saves 10% more energy on average than the sleep policy for 10 tasks. These results clearly show the adaptiveness and stability of DVSleak under different workloads.

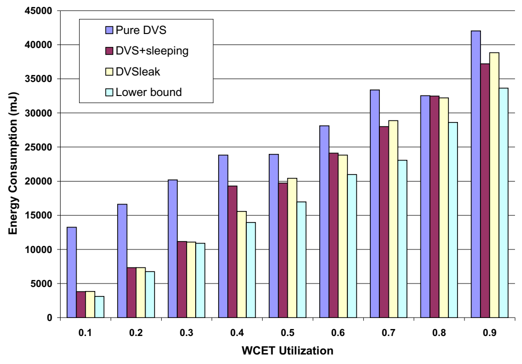
Figure 9.8 compares the performance of difference algorithms under three different dynamic patterns when the ratio of average execution time to WCET is fixed. Although the three patterns (patterns 2, 3, and 4) follow different fluctuations in execution time, DVSleak works equally well for all patterns. It saves 15% more energy on average than the sleep policy and 30% more energy on average than the pure DVS



(a) Avg. c=25%WCET

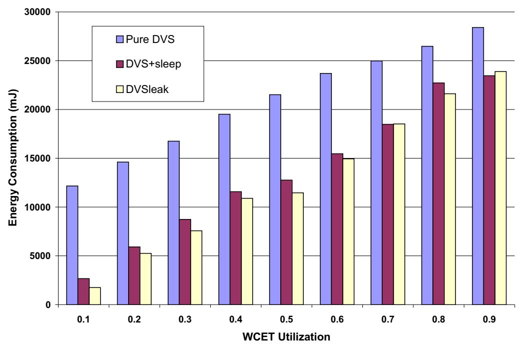


(b) Avg. c=50%WCET

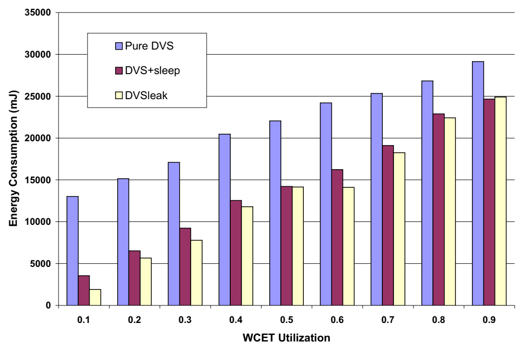


(c) Avg. c=75%WCET

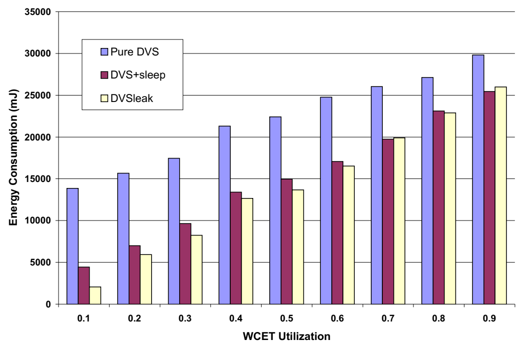
Figure 9.6: Energy Savings for for 3 Tasks, Pattern Two under Different Actual Execution Times (Variable) and Utilization



(a) Avg. c=25%WCET



(b) Avg. c=50%WCET

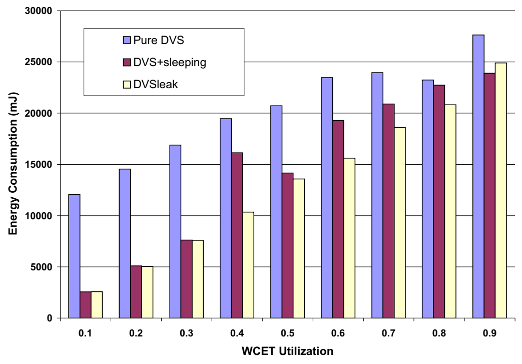


(c) Avg. c=75%WCET

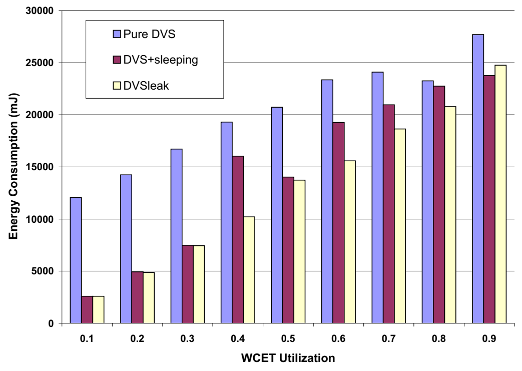
Figure 9.7: Energy Savings for 10 Tasks, Pattern Two under Different Actual Execution Times (Variable) and Utilization

algorithm. Overall, the combined sleep and delay algorithm, DVSleak, exhibits stable performance under different patterns due to the feedback control scheme used in our DVS algorithm, which adjusts automatically according to workload variations.

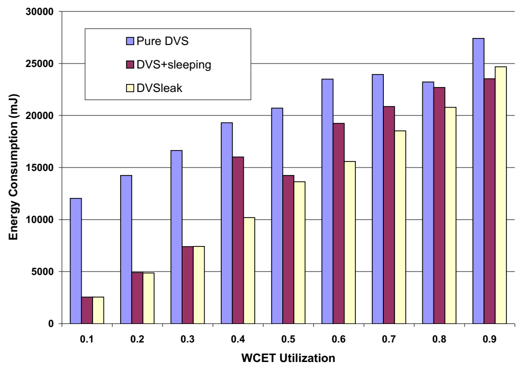
These experiments provide a better understanding of the three policies. The pure sleep policy and DVSleak do not show much difference under extremely low or extremely high utilization cases. In the former case, there is always enough slack for turning off the processor, no matter whether we delay the release time of a task or not. In the later case, there is hardly any slack at all, no matter how the release time of a task is delayed. Using the sleep policy alone in such a case is sufficient by itself to achieve virtually the same reduction in energy as the combined policy, albeit at a lower algorithmic complexity. At medium utilization, DVSleak excels due to its combined sleep and delay policy to shows its true potential of energy savings.



(a) Pattern 2



(b) Pattern 3



(c) Pattern 4

Figure 9.8: Energy Savings for 3 Tasks, Dynamic Pattern 2/3/4 when Average Execution Time = 50% WCET

# Chapter 10

# Conclusion and Future Work

In this dissertation, we present a novel energy management framework for CMOS-based processors, combining DVS technology with feedback control and leakage-aware schemes. This framework extends the traditional real-time EDF scheduling algorithm. It considers not only the timing requirements of real-time tasks but also energy consumption issues. Both the DVS scheduler and the feedback controller are implemented within the operating system.

We first introduce a task-splitting scheme. In order to obtain a low CPU frequency, each job is divided into a low-speed portion and a high-speed portion. While the high-speed portion always executes at the maximum CPU frequency, the low-speed portion is allowed to execute at a reduced CPU frequency. Voltage scaling on the low-speed portion benefits from the variation of jobs' actual execution times. Early completion of a job makes it possible to scale the entire job at a low frequency. If, however, the job requires its full worst-case execution time, it enters the high-speed portion with the maximal CPU frequency and voltage. Such a scheme aggressively exploits available slack in the schedule without violating the system's timing requirements.

We then present a preemption-handling technique. Preemption handling follows a greedy scheme by passing as much slack as possible to scale the next running task. We speculate on the early completion of the running task to aggregate more slack for other tasks. Forward sweep and backward sweep strategies are discussed as two available slot reservation methods. When preemption happens, the preemption-handling scheme maintains the timing behavior of the system by reserving necessary slack in the future for those preempted tasks.

A feedback control scheme is also integrated into our DVS algorithm to improve algorithmic performance for dynamic workloads, where the execution time of a periodic task may vary significantly from job to job. The feedback scheme facilitates scheduling behavior so that the DVS scheduler is more adaptable to dynamically changing workloads. With the information provided by the feedback controller, the WCET budget of the low-speed portion of a job is able to approximate its actual execution time. A proportional feedback structure, a multi-input feedback structure, and a single-input feedback structure are presented in this dissertation. For dynamically fluctuating execution-time patterns, the feedback DVS scheme achieves energy savings of up to 29% over previous work. The scheme is not sensitive to particular workload characteristics, *i.e.*, different execution-time patterns, and is equally scalable for both small and large number of tasks.

We evaluate the proposed DVS algorithm, as well as several other real-time DVS algorithms, not only in a simulation environment, but also on a real embedded development platform. Real-time task sets with varying execution times are assessed under different feedback schemes. The performance of our algorithm and its adaptability to dynamic workloads are evaluated. Asynchronous switching, which is a unique feature provided by the IBM 405LP embedded board, is assessed with real-time task sets. The experiments exhibit 5% additional energy savings with asynchronous switching, as opposed to traditional synchronous switching mechanisms. The experimental results also indicate a considerable potential for real-time DVS scheduling algorithms with up to 70% energy savings over a naïve DVS scheme. When we compare the feedback DVS algorithm with some of the best dynamic DVS algorithms presented in previous work, 24% additional energy savings are observed, which clearly shows the the strengths of our algorithm. To the best of our knowledge, this is the first comparative study of real-time DVS algorithms on a concrete micro-architecture. It is also the first evaluation of asynchronous DVS switching implemented in DVS algorithms.

We further study the potential of energy saving on CMOS-based processors when dynamic power is not dominant. Since leakage current increases significantly with every generation of processor technology, static power is anticipated to be a dominant factor in the total power consumption. Because of the existence of leakage power, forcing the processor into a sleep mode may be more energy-efficient than keeping the system idle at a low frequency, as long as the idle period is long enough to compensate for the shutdown overhead. We propose a combined DVS/leakage reduction scheme to minimizes both static and dynamic power consumption within a unified framework. This scheme profits from our feedback-DVS algorithm with a combined DVS and CPU sleeping policy. It automatically alternates between a frequency scaling state and a processor sleeping state, according to the run-time execution scenario of tasks. Frequency scaling is chosen when dynamic power is dominant. After the CPU voltage is reduced below a certain threshold, a sleep mode is entered, which is the most energy efficient state since static power dominates at that time. We point out that greedily delaying the start time of a task to put the processor into the sleep mode, as proposed in previous work, does not necessarily yield the most energy-efficient solution. The delaying decision needs to consider a task's dynamic execution behavior as well as its static behavior. DVSleak, the leakage-aware feedback-DVS algorithm, is implemented in a simulation environment. The combined sleep and delay algorithm exhibits stable performance under different task patterns due to the feedback control scheme used in the algorithm.

Overall, the main contributions of this dissertation include:

- A feedback-based DVS framework for dynamic workloads with hard real-time requirements.
- A combined intra-task and inter-task DVS scheme.
- Slack-passing and preemption-handling schemes for DVS schedulers.
- The implementation of our feedback DVS scheme in simulation, as well as the evaluation on an actual embedded platform.
- An extension of the feedback-DVS scheme for embedded architectures where static power is dominant.

The research presented in this dissertation also suggests many interesting research directions for future work.

One direction for future research is to study more sophisticated PID tuning methods, such as the Ziegler-Nichols approach[75]. Those methods are necessary to determine the PID parameters for dynamic system workloads instead of a manual tuning approach. It would further be of interest to investigate the development of feedback schemes for DVS algorithm with more complex execution characteristics. When the variation of tasks' behavior is non-linear, it cannot be handled by the classical linear control scheme. A more advanced control system is required for building a robust and adaptive DVS system in extremely unpredictable environments. It would also be of interest to evaluate the leakage-aware DVS algorithm on real embedded architectures. The leakage power on next-generation processor architectures keeps growing with lower threshold voltages. Although the simulation experiments in this thesis show promising results, an evaluation on real embedded platforms is required to justify the trade-off we have made during the design of the DVSleak algorithm.

Another direction for future research is to investigate DVS scheduling algorithms for resources-sharing tasks. This dissertation only focuses on independent task model where different tasks do not share any resources. When the execution of tasks depends on each other, reducing the speed of one task may influence the execution of another task. More sophisticated DVS algorithms need to be developed to coordinate among multiple non-independent tasks.

Instead of a CPU-oriented DVS algorithm, system-wide DVS algorithms are another interesting direction for future work. With the development of new hardware technology, CPU, memory, system buses and I/O devices are all expected to have frequency-scaling functionality. A DVS algorithm needs to be aware of the speed difference of each hardware component. A system-wide power model is required for the DVS algorithm to make its scheduling decisions.

# Bibliography

[1] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[2] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and agressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time Systems Symposium*, December 2001.

[3] H. Aydin and Q. Yang. Energy-responsiveness tradeoffs for real-time systems with mixed workload. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2004.

[4] Hakan Aydin, Rami Melhem, Daniel Mosse, and Pedro Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Comput.*, 53(5):584–600, 2004.

[5] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[6] B. Brock and K. Rajamani. Dynamic power management for embedded systems. In *IEEE International SOC Conference*, September 2003.

[7] Lama H. Chandrasena, Priyadarshana Chandrasena, and Michael J. Liebelt. An energy efficient rate selection algorithm for voltage quantized dynamic voltage scaling. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 124–129, New York, NY, USA, 2001. ACM Press.

[8] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[9] Wu chun Feng, Michael S. Warren, and Eric Weigle. The bladed beowulf: A cost-effective alternative to traditional beowulf. In *IEEE International Conference on Cluster Computing (CLUSTER 2002), 23-26, Chicago, IL, USA*, 2002.

[10] J. Kim D. Shin and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. In *IEEE Design and Test of Computers*, March 2001.

[11] A. Dudani, F. Mueller, and Y. Zhu. 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)*, pages 213–222, June 2002.

[12] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techiniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 37–46, June 1997.

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

[14] F. Gruian. 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*, Aug 2001.

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

[16] D. Grunwald, P. Levis, C. Morrey III, M. Neufeld, and K. Farkas. Policies for dynamic clock scheduling. In *Symp. on Operating Systems Design and Implementation*, Oct 2000.

[17] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium*, pages 68–77, December 1992.

[18] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

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

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

[21] IBM and MontaVisa Software. Dynamic power management for embedded systems. white paper.

[22] Intel. *Intel PXA255 Processor: Electrical, Mechanical, and Thermal Specification*, February 2004.

[23] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 37–46, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[24] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202. ACM Press, 1998.

[25] R. Jejurikar and R. Gupta. Integrating preemption threshold scheduling and dynamic voltage scaling for energy efficient real-time systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA '04)*, 25-27 Aug 2004.

[26] R. Jejurikar and R. Gupta. Optimized slowdown in real-time task systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS '04 )*, Jun30-Jul2 2004.

[27] R. Jejurikar and R. Gupta. Procrastination scheduling in fixed priority real-time systems. In *Proceedings of the Language Compilers and Tools for Embedded Systems*, Jun 2004.

[28] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 275–280, New York, NY, USA, 2004. ACM Press.

[29] D. Kang, S. Crago, and J. Suh. A fast resource synthesis technique for energy-efficient real-time systems. In *IEEE Real-Time Systems Symposium*, December 2002.

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

[31] Kanishka Lahiri, Sujit Dey, Debashis Panigrahi, and Anand Raghunathan. Battery-driven system design: A new frontier in low power design. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 261, Washington, DC, USA, 2002. IEEE Computer Society.

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

[33] Yann-Hang Lee and C. M. Krishna. Voltage-clock scaling for low energy consumption in fixed-priority real-time systems. *Real-Time Syst.*, 24(3):303–317, 2003.

[34] Yann-Hang Lee, Krishna P. Reddy, and C. M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *EuroMicro Conf. on Real Time Systems*, pages 105–112. IEEE Computer Society, 2003.

[35] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, December 1995.

[36] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[37] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[38] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[39] Yanbin Liu and Aloysius K. Mok. 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*, May 2003.

[40] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.

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

[42] C Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Syst.*, 23:85–126, 2002.

[43] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In

*Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 156–63, 2002.

[44] Michael Mächtel and Helmut Rzehak. Measuring the Influence if Real-Time Operating Systems on Performance and Determinism. *Control Eng. Practice*, 4(10):1461–1469, 1996.

[45] Steven M. Martin, Krisztián Flautner, Trevor N. Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Intl. Conference on Computer Aidded Design*, pages 721–725. ACM, 2002.

[46] R. Minerick, V. W. Freeh, and P. M. Kogge. Dynamic power management using feedback. In *Proceedings of Workshop on Compilers and Operating Systems for Low Power*, 2002.

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

[48] Trevor N. Mudge. Power: A first class design constraint for future architecture and automation. In *HiPC '00: Proceedings of the 7th International Conference on High Performance Computing*, pages 215–224, London, UK, 2000. Springer-Verlag.

[49] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[50] K. Nowka, G. Carpenter, and B. Brock. The design and application of the powerpc 405LP energy-efficient system on chip. *IBM Journal of Research and Development*, 47(5/6), September/November 2003.

[51] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[52] T. Pering, T. Burd, and R. Brodersen. The simulation of dynamic voltage scaling algorithms. In *Symp. on Low Power Electronics*, 1995.

[53] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium on Operating Systems Principles*, 2001.

[54] Christian Poellabauer, Leo Singleton, and Karsten Schwan. Feedback-based dynamic frequency scaling for memory-bound real-time applications. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, March 2005.

[55] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. *Technical report,Delft University of Technology*, 2000.

[56] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[57] Gang Qu. What is the limit of energy saving by dynamic voltage scaling? In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 560–563, Piscataway, NJ, USA, 2001. IEEE Press.

[58] Gang Quan, Linwei Niu, Xiaobo Sharon Hu, and Bren Mochocki. Fixed priority scheduling for reducing overall energy on variable voltage processors. In *25th IEEE Real-Time System Symposium*, pages 309–318. IEEE Computer Society, 2004.

[59] R. Rajamony and R. Bianchini. Energy management for server clusters. In *Tutorial, 16th Annual ACM International Conference on Supercomputing*, June 2002.

[60] Saowanee Saewong and Ragunathan Rajkumar. Practical voltage-scaling for fixed-priority RT-systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2003.

[61] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In

*LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 2–11, New York, NY, USA, 2002. ACM Press.

[62] D. Shin, W. Kim, J. Jeon, J. Kim, and S. L. Min. SimDVS: An integrated simulation environment for performance evaluation of dynamic voltage scaling algorithms. In *Workshop on Power-Aware Computer Systems*, February 2002.

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

[64] John A. Stankovic, Chenyang Lu, Sang H. Son, and Gang Tao. The case for feedback control real-time scheduling. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, June 1999.

[65] Vivek Tiwari, Deo Singh, Suresh Rajgopal, Gaurav Mehta, Rakesh Patel, and Franklin Baez. Reducing power in high-performance microprocessors. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 732–737, New York, NY, USA, 1998. ACM Press.

[66] Osman S. Unsal and Israel Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055–1069, 2003.

[67] Ankush Varma, Brinda Ganesh, Mainak Sen, Suchismita Roy Choudhury, Lakshmi Srinivasan, and Jacob Bruce. A control-theoretic approach to dynamic voltage scheduling. In *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, pages 255–266. ACM Press, 2003.

[68] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

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

[70] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 49–62, New York, NY, USA, 2003. ACM Press.

[71] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, page 374, Washington, DC, USA, 1995. IEEE Computer Society.

[72] Fan Zhang and Samuel T. Chanson. Processor voltage scheduling for real-time tasks with non-preemptable sections. In *IEEE Real-Time Systems Symposium*, December 2002.

[73] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.

[74] W. Zhang, M. Kandemir, N. Vijaykrishnan, M. Irwin, and V. De. Compiler support for reducing leakage energy consumption, 2003.

[75] J. G. Ziegler and N. B. Nichols. Optimum settings for automatic controllers. *ASME Transaction*, 64:759–768, November 1942.