# Abstract

In modern digital system designs, energy consumption is emerging as a key issue, especially for battery-powered portable systems. Dynamic voltage scaling (DVS), which adjusts dynamically the processor clock frequency and supply voltage, is one of the most effective approaches in reducing the energy consumptions of digital systems because energy consumption has a quadratic dependency on the supply voltage. However, this lead to a performance degradation of the system because the maximum allowable clock frequency is proportional to supply voltage. For hard real-time systems where timing constraints must be strictly satisfied, this energy-performance tradeoff makes it more challenging to adjust the supply voltage dynamically so that the energy consumption is minimized while not violating the timing requirements.

In this dissertation, clock and voltage scheduling algorithms for real-time applications are addressed. We propose a voltage scheduling model within task boundary, called IntraDVS. The clock frequency and supply voltage is controlled according to the execution flow of a task within the task boundary. By fully exploiting all the slack times, a scheduled program by the proposed technique always completes its execution near the deadline,

thus achieving a high energy reduction ratio.

First, we formulate the IntraDVS problem and propose an IntraDVS algorithm based on static timing analysis. It exploits the information about worst-case execution path. We also introduce a software tool that automatically converts a DVS-unaware program into an equivalent low-energy program using the IntraDVS techniques.

Second, two techniques to improve the energy performance of the IntraDVS are introduced. One is to use profile information to optimize the voltage scheduling for the average-case execution path. The other is to use data flow analysis technique to optimize the locations of scaling points.

Third, how to cooperate with the OS-level voltage scheduler is described. Though IntraDVS exploits all slack times within a task boundary, there are cases where it is better to transfer slack times to following tasks. We propose hybrid DVS algorithms, which determine the slack distribution observing the current system status.

Each algorithm proposed in this dissertation is evaluated in terms of energy consumption using simulations and measurements. We compare our voltage scheduling algorithms with other task-level or OS-level voltage scheduling algorithms. The experiments show the efficiencies of the newly proposed voltage scheduling techniques.

**Keywords**: dynamic voltage scaling, variable-voltage processor, real-time systems, power management, low-power design

**Student Number**: 2000-30298

# Contents

# List of Figures

# List of Tables

# Glossary

**ACEP** (Average-Case Execution Path): An execution path which has the highest probability to be executed.

**AVS** (Automatic Voltage Scaler): A software tool which converts a normal program into a DVS-enabled program.

**B-type VSE**: A VSE for the slack time generated at a branch statement.

**Data flow analysis**: A program analysis technique which provides global information about how a program manipulates its data. Using data flow analysis, we can know where a variable is defined and used.

**Data predecessor**: A statement which assigns a value to be used at a specified program location for a specified variable.

**Down-VSE**: A VSE where clock speed and voltage level are lowered.

**Hybrid DVS**: A DVS algorithm which uses both the intra-task voltage scheduling and the inter-task voltage scheduling.

**InterDVS**: A DVS algorithm which schedules the supply voltage between tasks.

**IntraDVS**: A DVS algorithm which schedules the supply voltage within task boundary.

**LaIntraDVS** (Look-ahead IntraDVS): An Intra-task DVS algorithm which

selects voltage scaling points using data flow analysis.

**LaVSP** (Look-ahead Voltage Scaling Point): A voltage scaling point determined by Look-ahead IntraDVS.

**Look-ahead point** : An earliest program point from which until a specified program location the values of specified variables are not changed.

**L-type VSE**: A VSE for the slack time generated at a loop statement.

**RAEC** (Remaining Average-case Execution Cycles): A number of cycles required to execute the average-case execution path starting from a specified point.

**RAEP** (Remaining Average-case Execution Path): An execution path which has the highest probability to be executed among the paths starting from a specified point.

**Reference Path**: An execution path whose execution cycles are used to determine the clock speed in IntraDVS.

**RPEC** (Remaining Predicted Execution Cycles): A number of cycles required to execute the predicted execution path starting from a specified point.

**RWEC** (Remaining Worst-case Execution Cycles): A number of cycles required to execute the longest execution path starting from a specified point.

**RWEP** (Remaining Worst-case Execution Path): An execution path which has the longest execution cycles among the all paths starting from a specified point.

**Slack Factor**: A value representing how much static slack times a task has comparing its WCET and deadline. (deadline-WCET)/deadline.

**SUR** (Speed Update Ratio): A ratio between two clock speeds before and after voltage scaling edge.

**Up-VSE**: A VSE where clock speed and voltage level are raised.

**Variable-voltage processor**: A processor where the clock speed and supply voltage can be adjusted.

**VSE** (Voltage Scaling Edge): A program location where the clock speed and voltage is adjusted.

**WCEC** (Worst-Case Execution Cycles): A number of cycles required to execute the worst-case execution path.

**WCEP** (Worst-Case Execution Path): An execution path which has the longest execution cycles among the all paths starting from the program entry.

**WCET** (Worst-Case Execution Time): An execution time required to execute the worst-case execution path.

**WCPU** (Worst-Case Processor Utilization): A maximum system utilization assuming that all tasks demand the worst-case execution cycles.

**Workload-variation slack time**: Dynamic slack time. A slack time generated dynamically when a task execution time is smaller than its worst-case execution time.

**Worst-case slack time**: Static slack time. A extra time identified statically when the worst-case processor utilization is smaller than 1.

# Chapter 1

# Introduction

## 1.1 Motivation

Recently, the reduction of energy consumption is emerging as a key technology in the VLSI system design, especially for battery-powered portable systems such as digital cellular phones, personal digital assistants, and mobile videophones. For these systems, the low energy consumption is a primary design goal, since the battery operation time is one of the most significant performance measures. Even for non-portable VLSI systems such as high performance microprocessors, the energy consumption is an important design constraint, because large heat dissipations in high-end microprocessors often result in the device thermal degradation, system malfunction, or in some cases, non-recoverable crash. These problems demand low-power technologies over a wide range of hardware and software design abstractions, including device, circuit, logic, architecture, compiler, operating system,

and application levels. Recently, there have been many researches on the system or software-level low-power techniques using compiler and operating systems because the opportunities for power reduction at such levels are larger than those of hardware-level techniques.

Dynamic voltage scaling (DVS) [1] is one of the most effective approaches in reducing the power consumption of embedded systems, where supply voltage can be dynamically reduced to the lowest possible extent that ensures a proper operation when the required performance of the target system is lower than the maximum performance. Since the dynamic energy consumption of CMOS circuits, which dominates total power consumption, is proportional to the square of the supply voltage $V_{dd}$, a significant energy reduction is possible with the DVS scheme. Recently, many commercial variable-voltage microprocessors (e.g., [2, 3, 4]) have been introduced to the mobile embedded market, reflecting the effectiveness of DVS techniques.

In the past, various OS-level voltage scheduling algorithms have been proposed for hard real-time systems [5, 6, 7, 8, 9, 10, 11, 12]. Given multiple tasks, these algorithms assign the proper speed to each task dynamically while guaranteeing all their deadlines. In real-time systems, since the real execution time of each task may smaller than the worst-case execution time (WCET), workload-variation slack times [13] are generated at run time even though the worst-case utilization of the processor is 1. However, it is difficult to utilize the workload-variation slack times because we cannot know the exact amount of slack time before the completion of a task. Therefore, most of DVS scheduling algorithms [7, 8, 10, 11, 12] transfer the slack time to the following tasks which can utilize it. These techniques exploit the "slack estimation and distribution" strategy for the supply voltage determination,

which can be summarized as follows:

1. run the current task.
2. estimate the slack time due to early completion of the current task.
3. distribute the slack time to the next tasks and determining the operating speed of the tasks.
4. run the next tasks.

These techniques determine the supply voltage on *task-by-task basis*. For each task activation, only one supply voltage is assigned to the task, and it is not changed during the task execution. In this paper, we call these techniques as *inter-task dynamic voltage scheduling* (InterDVS).

While generally effective in reducing energy consumption of multi-task real-time systems, InterDVS has several practical limitations. For example, since a task scheduler in OS determines the supply voltage of a task, it requires OS modifications. Furthermore, it cannot be applied to a single-task environment because there is no another task which can utilize the slack time generated by the completed task. Considering many small-size embedded mobile applications are based on a single-task model, this can be detrimental to a wide adoption of variable-voltage processors in practice.

Even in a multi-task environment, InterDVS may not be effective in reducing the energy consumption if one task is dominant in both the slack times and execution times. In this case, a dominant task (with the highest energy consumption) exploits slack times from other tasks (with small slack times), thus ineffective in reducing the energy consumption. For example,

Table 1.1: A typical videophone application.

| | | MPEG4 Video Encoding | MPEG4 Video Decoding | VSELP Speech Encoding | VSELP Speech Decoding |
|---|---|---|---|---|---|
| Period (= Deadline) (msec) | | 66.667 | 66.667 | 40.000 | 40.000 |
| WCET (msec) | | 50.386 | 9.826 | 1.844 | 1.383 |
| Average Execution Time (msec) | | 13.099 | 1.460 | 0.907 | 0.680 |
| Energy Consumption | InterDVS [8] | 0.826 | | | |
| | Off-line Optimal | 0.106 | | | |

*In normalizing energy consumption values, the base case of normalization is DVS-unaware systems using only power-down mode.

consider a typical mobile videophone application with four tasks shown in Table 1.1. Using an InterDVS algorithm of [8], only 17% of energy reduction is observed while an off-line (theoretical) optimal voltage scheduling can achieve 90% power reduction[1].

The limitations of InterDVS algorithms come from their slack estimation techniques. InterDVS estimates a task's workload-variation slack time at the completion of the task. However, it may be too late if there is no or small task workload to exploit the slack time as shown in the videophone application. Therefore, we need a new voltage scheduling technique which can identify the workload-variation slack time instantly during the task execution.

[1]The energy reduction by the InterDVS algorithm of [8] was estimated using a simulation. The off-line optimal schedule was calculated by assuming that the real execution times of tasks are known *a priori*.

## 1.2 Dissertation Goals

In this dissertation, we are to provide voltage scheduling algorithms which can identify the workload-variation slack time within task boundary. The scheduling algorithm should reduce the energy consumption of real-time tasks guaranteeing the timing constraints of them. Our goals can be described as follows:

- Proposing new voltage scheduling algorithms which can identify and exploit workload-variation slack times of a task during the task execution.
- Optimizing the proposed voltage scheduling algorithms to improve the energy performance.
- Integrating the proposed voltage scheduling algorithms with OS-level InterDVS algorithms.
- Providing an automatic conversion tool that converts DVS-unaware programs into DVS-aware ones based on the proposed voltage scheduling algorithms.
- Examining the energy performance of the proposed voltage scheduling algorithms comparing with other voltage scheduling algorithms.

## 1.3 Contributions

This dissertation proposes the voltage scheduling technique called the *intra-task dynamic voltage scheduling* (IntraDVS) because it adjusts the voltage

and clock speed within a task. The technique identifies the slack time generated from the workload variation and adjusts the clock/voltage to utilize the slack time at run time. To enable the run-time clock/voltage adjustment, the application code is preprocessed based on the static timing analysis. The workload-variation slack times are identified by observing the changes of the remaining execution times due to the control flow. Specifically, we present:

- A new IntraDVS algorithm using static timing analysis. The algorithm, entitled **RWEP-based IntraDVS** [14], finds the voltage scaling points from a target program and determines the clock speed at the points. It uses the worst-case timing analysis technique to know the static timing information of the target program. The proposed IntraDVS algorithm has the following features: (1) It fully exploits *all* workload-variation slack times, achieving a significant improvement in the energy consumption. (2) It is applicable to a single-task environment, since it controls the supply voltage *within* each task. (3) It provides an *automatic* conversion tool that converts DVS-unaware programs into DVS-aware ones. This means that a programmer requires no knowledge on DVS, making the proposed algorithm very practical. (4) It enables each individual task to control supply voltage independent of other tasks, without any support from operating systems. Therefore, it can be directly applied to a conventional *DVS-unaware* OS without any modification.

- An extension of the IntraDVS algorithm using profile information. The algorithm, entitled **RAEP-based IntraDVS** [15], determines

the clock speed based on the average-case execution information guaranteeing the timing constraint of the target program.

- An extension of the IntraDVS algorithm using data flow analysis. The algorithm, called **Look-ahead IntraDVS**, optimizes the locations of scaling points by searching the earliest points where we can detect the change of workload using a data flow analysis technique.

- Techniques for cooperation with InterDVS algorithms. We propose **Hybrid IntraDVS** [16] algorithms which cooperate with the InterDVS technique to balance the slack consumption among tasks.

- A software tool for IntraDVS techniques. Based on the proposed IntraDVS algorithms, we developed a software tool called **Automatic Voltage Scaler** (AVS) that automatically converts a DVS-unaware program into an equivalent low-energy program.

## 1.4 Dissertation Structure

The dissertation has seven chapters and two appendices. The first chapter is this introduction. The last chapter offers conclusions and describes avenues for future work. The five intermediate chapters address the following.

Chapter 2 provides the background for real-time systems and dynamic voltage scaling. We introduce the worst-case timing analysis technique for real-time systems, which is closely related to our work. The fundamental relations between clock frequency, supply voltage, power and energy are described. We also serves the related works on dynamic voltage scheduling.

Chapter 3 describes the overall algorithm of the proposed IntraDVS and the formulation of IntraDVS problem.

Chapter 4 presents details of the IntraDVS algorithm which uses the static timing analysis technique. We describe how to select the voltage scaling points and how to transform a target program to make a DVS-aware program.

Chapter 5 describes two improvement techniques for IntraDVS, using the execution profile information and using the data flow analysis.

Chapter 6 shows how to cooperate for IntraDVS with OS-level InterDVS algorithms. We show that there are cases where the pure IntraDVS or the pure InterDVS dose not work well and propose hybrid IntraDVS techniques which use both IntraDVS and InterDVS.

Appendix provides the information about several variable-voltage processors and the detail description about the automatic voltage scaler, the software tool for IntraDVS.

# Chapter 2

# Background

## 2.1 Real-Time Systems

Real-time systems are considered to be those types of systems which have to respond to certain stimuli within a finite and specified delay. In other words, the correctness such systems depends not only on the logical result of the computations, but also on the time at which the results are produced. For hard real-time systems, it is imperative that responses occur within the specified deadline, any exception leading to a total failure of the system. In soft real-time systems, response times are important, but the system will still function correctly if some deadlines are occasionally missed.

Various scheduling techniques have been proposed to ensure that tasks finish before their deadlines. These scheduling algorithms generally require that the worst-case execution time (WCET) of each task in the system be known a priori. The worst-case execution time is a possible longest

execution time of a program code on a given hardware. The WCET of a task depends on many factors including the source code of a target application, the target machine's architectural features, compiler and operating system, etc. There have been several researches focusing on the estimation of the WCETs of tasks. The estimation of WCET must be safe and accurate. The safeness means that the estimated WCET is not shorter than the real WCET. The accuracy means that the estimated WCET is close to the real WCET. An accurate and safe estimation of a task's WCET is crucial for reasoning about the timing properties of real-time systems.

One of most popular static WCET analysis techniques is to use a timing schema [17]. In this approach, we first build the control flow graph of basic blocks of a program. Next, we determine the time of each basic block by adding up the execution time of the machine instructions. Finally, we determine the WCET of a whole program by using timing schema. The timing schema is a set of formulas for computing execution time bounds of language constructs. For example, consider a conditional statement

$$S : \textbf{if}(E) \ \ \textbf{then} \ \ S_1; \ \ \textbf{else} \ \ S_2.$$

We can represent the WCET of the statement $S$, $WCET(S)$, as

$$WCET(E) + max(WCET(S_1), WCET(S_2))$$

assuming a simple single-issue architecture without pipeline and cache for the target machine.

However, for modern RISC processors, such timing information is not sufficient to accurately account for timing variations resulting from pipelined execution and cache memory. So, there have been intensive studies about

the WCET estimation for modern processors [18, 19, 20]. Especially, Lim *et al.* [21] proposed an extended timing schema of RISC processors, which models pipeline and cache. Using the timing schema, they built a timing analysis tool. We use the timing analysis tool for our IntraDVS algorithms to know the static timing information of a target program.

## 2.2 Dynamic Voltage Scaling

### 2.2.1 Power and Energy

In order to design energy-efficient systems, one has to understand first the sources of energy consumption. It is also important to examine the relations between power, energy, and signal delay in digital CMOS circuits. The power dissipated on a CMOS circuit can be decomposed into two basic types, static and dynamic [22]:

$$P_{CMOS} = P_{static} + P_{dynamic} \tag{2.1}$$

In the ideal case, CMOS circuits do not dissipate static power, since in steady state there is no open path from source to ground. In reality, there are always leakage currents and short circuit currents which yield the static component of the CMOS power consumption. Although the static power is today about two orders of magnitude smaller than the total power, the typical chip's leakage power increases about 5 times each generation, and will soon become a significant portion of the total power [23].

The dynamic component of the CMOS power is dissipated during the

transient behavior, i.e. during switching between logic levels and is represented as following equation:

$$P_{dynamic} = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f_{clk} \tag{2.2}$$

$\alpha$ is the switching activity factor (the average number of high-to-low transitions in one clock period), $C_L$ is the load capacitance, $V_{dd}$ is the supply voltage and $f_{clk}$ is the clock frequency. In CMOS circuits, this component of power dissipation accounts for at least 85-90% of the total power consumption [22].

From all the considerations made above, we can approximate the power dissipated on a CMOS circuit node using the following equation:

$$P_{CMOS} \approx P_{dynamic} = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f_{clk} = C_{eff} \cdot V_{dd}^2 \cdot f_{clk} \tag{2.3}$$

where $C_{eff}$ is the effective load capacitance.

This means that the power consumption in a CMOS circuit is proportional to the switching activity, capacitive load, clock frequency, and the square of the supply voltage. All the power and energy reduction techniques try to minimize one or more of these factors. Especially, supply voltage ($V_{dd}$) reduction appears to be the most promising, because of its quadratic dependency to power. A decrease in voltage by a factor of two yields a decrease in power by a factor of four. We call the technique which adjust the supply voltage to minimize the power consumption as the *dynamic voltage scaling*.

In this dissertation, we focus on energy rather than power consumption. Although low power and energy efficiency are often perceived as overlapping goals, there are certain differences when designing for one or the other.

Formally, the energy consumed by a system is the amount of power used during a certain period of time. We can denote the energy consumption $E$ during the time interval $T$ as follows:

$$E = \int_0^T P(t)dt \propto V_{dd}^2 \cdot f_{clk} \cdot T = V_{dd}^2 \cdot N_{cycle} \tag{2.4}$$

$P(t)$ is the power consumption at the time $t$ and $N_{cycle}$ is the number of clock cycles during the time interval $T$. Equation (2.4) tells that reducing only the clock frequency makes no change in the energy consumption (though it reduces the power dissipation).

Unfortunately, we can not reduce the supply voltage for free. The circuit delay $\Delta$, which sets the clock frequency, depends on the supply voltage [24]:

$$\frac{1}{f_{clk}} \propto \Delta \propto \frac{V_{dd}}{(V_{dd} - V_t)^\gamma} \tag{2.5}$$

where $V_t$ is the threshold voltage and $\gamma$ is the saturation velocity index ($\gamma$ is between 1 and 2.). For a sufficiently small $V_t$ we can rewrite the relation between clock frequency and supply voltage as:

$$f_{clk} \propto V_{dd}^{(\gamma-1)} \tag{2.6}$$

For this reason supply voltage and clock frequency should be scaled together. Consequently, dynamic voltage scaling provides the energy reduction but lead to a slow system. Real-time scheduling and energy minimization are therefore closely related problems, that should be tackled in conjunction for best results.

Table 2.1: Variable voltage processors.

| Processor | | Clock Range | Voltage Range | Transition Time |
|---|---|---|---|---|
| Commercial | Transmeta Crusoe [26] | $200 - 700$MHz | $1.1 - 1.65V$ | $300\mu$s |
| | AMD Mobile K6 [3] | $192 - 588$MHz | $0.9 - 2.0V$ | $200\mu$s |
| | Intel PXA250 [4] | $100 - 400$MHz | $0.85 - 1.3V$ | $500\mu$s |
| | IBM PowerPC 405LP [27] | $152 - 380$MHz | $1.0 - 1.8V$ | $400\mu$s |
| | Compaq Itsy [28] | $59.0 - 206.4$MHz | $1.0 - 1.55V$ | $189\mu$s |
| | TI TMS320C55x [25] | $6 - 200$MHz | $1.1 - 1.6V$ | $300\mu s(1.1 \rightarrow 1.6V)$<br>$3.3ms(1.6 \rightarrow 1.1V)$ |
| Academic | UC Berkely lpARM [1] | $5 - 80$MHz | $1.2 - 3.8V$ | $70\mu$s |
| | TU Delft LART [29] | $59 - 221$MHz | $0.8 - 1.5V$ | $140\mu s(59 \leftrightarrow 221\text{MHz})$<br>$40\mu s(0.8 \rightarrow 1.5V)$<br>$5.5ms(1.5 \rightarrow 0.8V)$ |

### 2.2.2 Variable-Voltage Processors

Recently, many variable-voltage processors have been announced. Table 2.1 shows the representative commercial variable-voltage processors and academic trials to implement variable-voltage processors. These processors provide finite numbers of voltage and clock levels within the voltage/clock range specified. Each processor requires a time delay to change the voltage/clock level. Most of variable-voltage processors except Transmeta's Crusoe provide the software mechanisms for users to be able to control the voltage and clock level such as TI's Power Scaling Library [25]. Appendix A contains the detail descriptions of variable-voltage processors. Throughout this dissertation, we make the following assumptions on a target variable-voltage processor:

1. The processor provides a special instruction, `change_f_V(`$f_{clk}$`)`, that dynamically controls clock frequency $f_{clk}$ and its corresponding voltage $V_{dd}$ of the processor based on Equation (2.6).

2. $f_{clk}$ and $V_{dd}$ have continuous values within the operational range of the processor.

3. When the processor changes ($f_{clk1}$,$V_{dd1}$) to ($f_{clk2}$,$V_{dd2}$), there is a clock/voltage transition overhead period of $C_{VTO}$ cycles[1].

4. During clock/voltage transition, the processor stops running.

Assumptions (1), (3) and (4) are valid for most of variable processors. Especially, assumption (4) is conservative considering recent variable-voltage processors such as TI's TMS320C55x which stops during the clock transition but operates during the voltage transition. Assumption (2) is not realistic because most of variable-voltage processors provide only discrete voltage/clock levels. However, our work can support the processors with a finite number of voltage/clock levels with a slight modification of speed selection algorithm.

Generally, dynamic voltage scaling is applied only to processor and on-chip memory (on-chip cache). Other peripherals and external memory have their own clock frequencies different from internal processor frequency and sustain the clock frequencies despite of the clock scaling of processor. In this dissertation, we consider only the processor's energy consumption and assume that the system performance is proportional to the processor's clock frequency. The DVS techniques considering other external devices as well

[1] The clock/voltage transition time is different depending on the source voltage and the target voltage. However, we assumed there is a fixed voltage transition time for a simple explanation. Since we represent the fixed clock/voltage transition overhead period by the number of cycles, it can vary depending on the current clock frequency. For a simpler analysis, we assume that $C_{VTO}$ cycles were counted under the maximum clock frequency.

as processor is one of our future works.

## 2.3 Related Works

For hard real-time systems where timing constraints must be strictly satisfied, a fundamental energy-delay tradeoff makes it more challenging to adjust the supply voltage dynamically while minimizing the energy consumption and guaranteeing the timing requirements. For this reason, extensive studies have been recently carried out on the DVS problems. For hard real-time systems, there are two kinds of voltage scheduling approaches depending on the voltage scaling granularity: intra-task DVS (IntraDVS) and inter-task DVS (InterDVS). The intra-task DVS algorithms [14, 30] adjust the voltage within an individual task boundary, while the inter-task DVS algorithms determine the voltage on a task-by-task basis at each scheduling point. The main difference between them is whether the slack times are used for the current task or for the tasks that follow. InterDVS algorithms distribute the slack times from the current task for the following tasks, while IntraDVS algorithms use the slack times from the current task for the current task itself.

### 2.3.1 InterDVS Algorithms

The InterDVS algorithms are classified depending on slack estimation method. Slack times generally come from two sources; *worst-case slack times* are the extra times identified statically when the worst-case processor utilization is smaller than 1, while *workload-variation slack times* are caused from run-

time variations of the task executions.

The worst-case slack estimation methods is to compute the speed of each task, which is defined as the clock speed to minimize the energy consumption guaranteeing the feasible schedule of a task set [5, 6, 31, 30, 32]. For example, in EDF scheduling, if the worst case processor utilization (WCPU) $U$ of a given task set is lower than 1.0 under the maximum speed $f_{max}$, the task set can be scheduled with the speed $f = U \cdot f_{max}$.

Even though a given task set is scheduled such that there are no worst-case slack times, since the actual execution times of tasks are usually much less than their WCETs, the tasks usually have workload-variation slack times. One simple method to estimate the workload-variation slack time is to use the arrival time of the next task [8, 31]. (The arrival time of the next task is denoted by NTA.) Assume that the current task $\tau$ is scheduled at time $t$. If NTA of $\tau$ is later than ($t$+WCET($\tau$)), task $\tau$ can be executed at a lower speed so that its execution completes exactly at the NTA.

In the priority-driven scheduling such as RM and EDF, we can exploit the basic properties of the scheduling to estimate workload-variation slack times. The basic idea is that when a higher-priority task completes its execution earlier than its WCET, the following lower-priority tasks can use the slack time from the completed higher-priority task. It is also possible for a higher-priority task to utilize the slack times from completed lower-priority tasks. However, the latter type of slack stealing is computationally expensive to implement precisely. Therefore, the existing algorithms are based on heuristics [10, 12].

Another method is to use the processor utilization. The actual processor

utilization during run time is usually lower than the worst case processor utilization. We can estimates the required processor performance at the current scheduling point by recalculating the expected worst case processor utilization using the actual execution times of completed task instances [11]. When the processor utilization is updated, the clock speed can be adjusted accordingly. The main merit of this method is its simple implementation, since only the processor utilization of completed task instances have to be updated at each scheduling point. Kim *et al.* [33] evaluated the energy efficiencies of state-of-the-art InterDVS algorithms.

There have been studies on InterDVS for special configurations. Im *et al.* [34] proposed an InterDVS technique for multimedia applications. Their algorithm fully utilizes the idle intervals with buffers in a variable speed processor and determines the minimum buffer size to achieve the maximum energy saving. The energy performance of the technique is strongly dependent on the available buffer size. This technique has a limitation that it can be used only when a system can buffer multiple input data or output results.

Hong *et al.* [35] have proposed a set of heuristic algorithms to schedule a mixed workload of periodic and sporadic tasks. Their algorithm optimizes the energy consumption while ensuring that all periodic tasks meet their deadlines and accept as many sporadic tasks, which can be guaranteed to meet their deadlines, as possible.

Shin and Kim [36] have proposed scheduling algorithms for the systems which have both periodic tasks and aperiodic tasks. The algorithms tries to minimize the energy consumption guaranteeing the timing constraints

of periodic tasks while bounding the maximum increase of aperiodic tasks' response time. They exploited the behaviors of the bandwidth-preserving servers [37] for aperiodic task scheduling such as deferrable server, sporadic server, total bandwidth server and constant bandwidth server.

### 2.3.2 IntraDVS Algorithms

Comparing with InterDVS techniques, few approaches have been introduced for IntraDVS techniques. Lee and Sakurai [13] proposed an intra-task voltage scheduling where each task is partitioned into fixed-length segments. After the completion of each segment, the supply voltage is adjusted depending on the slack time made by the previous segment. This is the same idea of InterDVS if we regard the segment as a task. Although [13] shows a significant improvement in the energy reduction, it provides no systematic methodology for developing DVS-aware intra-task applications. For example, there exists no systematic guideline of selecting the best program locations where voltage scaling code is inserted. Consequently, the programmer himself should find out proper locations based on his own knowledge. It implies that the technique described in [13] is very difficult to be applied to practical applications, since average programmers are generally not familiar with low-energy software issues as well as timing analysis techniques.

Another approach of IntraDVS is based on the stochastic method [30, 38]. This technique is motivated by the idea that it is usually better to *start at low speed and accelerate execution later when needed* than to *start at high speed and reduce the speed later when the slack time is found* in the program execution. However, it requires the probability density function of execution

times of a task to calculate the optimal speed schedule. Furthermore, the stochastic IntraDVS requires OS modification like InterDVS and cannot utilize all the slack times. In Chapter 4, we compare the energy efficiency of the stochastic IntraDVS technique with our proposed IntraDVS techniques.

Hsu and Kremer [39] also proposed a different kind of IntraDVS. While our IntraDVS estimates the slack times of a task based on the changes of the remaining execution cycles due to the control flow, their technique finds the slack times based on the architectural characteristic of microprocessor. By identifying the program regions in which the CPU is mostly idle due to memory stalls, their technique slows down the clock speed of CPU in the regions for energy reduction with negligible performance loss. However, this technique should be carefully used in real-time systems due to the performance degradation. Since this technique is different from our IntraDVS in the kind of slack times exploited, it can be used together with our IntraDVS for a better energy performance.

# Chapter 3

# Intra-Task Voltage Scheduling Framework

## 3.1 Basic Idea

Consider a hard real-time program $P$ with the deadline of 2 sec shown in Figure 3.1(a). The control flow graph (CFG) $G_P$ of the program $P$ is shown in Figure 3.1(b). In $G_P$, each node represents a basic block of $P$ and each edge indicates the control dependency between basic blocks. The number within each node indicates the number of execution cycles of the corresponding basic block. The back edge from $b_5$ to $b_{wh}$ models the **while** loop of the program $P$.

In developing hard real-time systems where tasks have strict timing constraints (i.e., deadlines), the worst-case execution times (WCETs) of the tasks are estimated in advance (prior to run time) to guarantee that re-

quired timing constraints are met. Such WCETs can be predicted by existing analysis tools that produce safe and accurate prediction results [40, 21]. Using a WCET analysis tool, we can find the path $p_{worst}$ = ($b_1$, $b_{wh}$, $b_3$, $b_4$, $b_5$, $b_{wh}$, $b_3$, $b_4$, $b_5$, $b_{wh}$, $b_3$, $b_4$, $b_5$, $b_{wh}$, $b_{if}$, $b_{call4}$, $b_8$, $b_{10}$, $b_{11}$, $b_7$) as the worst-case execution path (WCEP) for the example program $P$, assuming that the maximum number of **while** loop iterations is set to 3 by user. The predicted execution cycles of $p_{worst}$ is, therefore, $200 \times 10^6$ cycles[1], which is the worst-case execution cycles (WCEC). If a target processor operates at the maximal clock frequency of 100 MHz, the program $P$ completes its execution in 2 sec, resulting in no slack time.

However, there are large execution time variations among different execution paths. In particular, the average-case execution paths (ACEPs) complete their executions much earlier than the WCEP(s) does [8]. For the example program shown in Figure 3.1(b), there exist 51 different execution paths. While the WCEP $p_{worst}$ takes $200 \times 10^6$ cycles, twelve of 51 possible execution paths take less than $100 \times 10^6$ cycles. For such short execution paths, the workload-variation slack times are generated. If we were able to identify them in the early phase of its execution, we can lower the clock speed substantially, thus saving a significant amount of energy consumption. Consider the path $p_1 = (b_1, b_{call1}, b_8, b_9, b_{11}, b_{if}, b_{call4}, b_8, b_{10}, b_{11}, b_7)$ of Figure 3.1(b) whose execution takes $95 \times 10^6$ cycles. In the ideal case, we can start the execution with the clock speed of 47.5 MHz without violating

[1]For simple explanations, we assume that the number of execution cycles for an execution path is the sum of all basic blocks' execution cycles. This is true when the target machine is a single issue architecture without pipeline and cache. However, our algorithms can be used for any architecture if there is a timing analysis tool for the architecture.

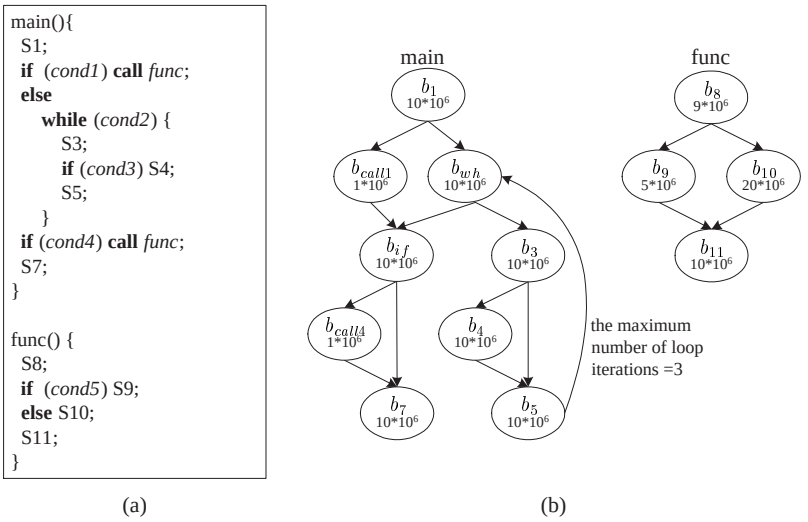



Figure 3.1: An example program $P$: (a) an example real-time program with the 2 sec deadline and (b) its CFG representation $G_P$

the 2 sec deadline, if we can perfectly predict that the actual execution path will be $p_1$ before the processor starts $b_1$. Unfortunately, we do not generally know in advance which execution path will be taken by the next program execution. Therefore, we cannot start with the 47.5 MHz clock speed, although this will improve the energy efficiency significantly.

The solution for this problem is to adjust the clock speed within the task depending on the workload-variations. For example, when the program control flow follows the execution path $p_1$ of Figure 3.1(b), we can reduce the clock speed at edge $(b_1, b_{call1})$ because we know this control flow dose not follow the WCEP. In the proposed IntraDVS algorithm, we identifies the appropriate program locations where the clock speed should be adjusted, and inserts clock and voltage scaling codes to the selected program locations at compile time. The branching edges of the CFG, i.e., branch or loop

statements, are the candidate locations for inserting voltage scaling codes because where the changes of remaining execution cycles are occurred.

## 3.2 Problem Modeling

We consider a real-time task $\tau$ with the deadline $D$. The task $\tau$ is represented by its CFG $G_\tau$. If the task $\tau$ has $N$ number of basic blocks, $b_1, b_2, \cdots, b_N$, $G_\tau$ consists of $N$ nodes. (We assume that $b_1$ is the entry basic block of the task $\tau$.) We associate each basic block $b_i$ with its basic block information (BBI) structure. The BBI structure BBI($b_i$) of the basic block $b_i$ consists of three entries: $C_{EC}(b_i)$, $S(b_i)$, and $E(b_i)$. $C_{EC}(b_i)$ denotes the number of clock cycles[2] needed to execute $b_i$. $S(b_i)$ represents the processor speed in clock frequency at which $b_i$ is executed. $E(b_i)$ is defined as $C_{EC}(b_i) \cdot S(b_i)^2$ by Equation (2.4). (We assume the supply voltage is proportional to the clock speed.) $E(b_i)$ is the relative energy consumption during the execution of $b_i$.

Similar notations are defined for execution paths. $p_i$ denotes an execution path of a task $\tau$. $p_i$ can be expressed as a sequence of basic blocks. $C_{EC}(p_i)$ represents the number of execution cycles when $p_i$ is executed. For paths, we use a notation $\Phi^{b_i}$ which means the set of all the partial execution paths starting from basic block $b_i$.

[2]Note that BBI definition above is represented in execution cycles, instead of execution time. This is because, as we adjust the clock speed on a variable-voltage processor, the execution time is changing for a given basic block, but the number of execution cycles remains constant. Given the number of execution cycles, the execution time can be computed by multiplying the clock cycle time.

The IntraDVS algorithm is to find how much we should change the clock speed at each edge $(b_i, b_j)$ in the CFG of a target program to minimize the total energy consumption of the program satisfying the timing constraint of the program. That is, we should find the value $r_{i,j} = S(b_j)/S(b_i)$ for each edge $(b_i, b_j)$. We call this ratio a *speed update ratio* (SUR). For an execution path $p_m = (b_1, \cdots, b_{n_m})$, $S(b_i)$ can be represented as

$$S(b_i) = S_0 \cdot \prod_{k=1}^{i} r_{k-1,k} \quad (S(b_i) \geq S_{min} \text{ and } S(b_i) \leq S_{max}), \tag{3.1}$$

where $S_0$ is the initial clock speed at the start of a program and $S_{min}$ ($S_{max}$) is the minimum (maximum) value of the clock speed provided by the target variable-voltage processor. We can also denote $E(b_i)$ as

$$E(b_i) = C_{EC}(b_i) \cdot S(b_i)^2 = C_{EC}(b_i) \cdot (S_0 \prod_{1}^{i} r_{k-1,k})^2. \tag{3.2}$$

Then, the energy consumption during the execution of the path $p_m$ is proportional to

$$E(p_m) = \sum_{i=1}^{n_m} E(b_i) = \sum_{i=1}^{n_m} \left( C_{EC}(b_i) \cdot (S_0 \prod_{1}^{i} r_{k-1,k})^2 \right). \tag{3.3}$$

From this formula, we can represent the target function to minimize as

$$\sum_{\forall p_m \in \Phi^{b_1}} E(p_m) \cdot prob(p_m) = \sum_{\forall p_m \in \Phi^{b_1}} \left[ \sum_{i=1}^{n_m} \left( C_{EC}(b_i) \cdot (S_0 \prod_{1}^{i} r_{k-1,k})^2 \right) \cdot prob(p_m) \right] \tag{3.4}$$

$prob(p_m)$ is the probability that the path $p_m$ is executed among all the paths in $\Phi^{b_1}$.

There is a timing constraint for this problem. Since the target program $\tau$ should be completed before the deadline $D$, we can denote the timing constraint as

$$\forall p_m, \sum_{i=1}^{n_m} \frac{C_{EC}(b_i)}{S(b_i)} = \sum_{i=1}^{n_m} \frac{C_{EC}(b_i)}{S_0 \prod_1^i r_{k-1,k}} \leq D. \tag{3.5}$$

Using a simple inference, we can conclude the optimal solution satisfies

$$\forall p_m, \sum_{i=1}^{n_m} \frac{C_{EC}(b_i)}{S_0 \prod_1^i r_{k-1,k}} = D, \quad \forall p_m, \sum_{i=1}^{n_m} \frac{C_{EC}(b_i)}{D \prod_1^i r_{k-1,k}} = S_0. \tag{3.6}$$

So, $S_0$ can be estimated when $r_{i,j}$ for each edge $(b_i, b_j)$ is determined.

Since the formula (3.4) is a non-linear equation for $r_{i,j}$, this problem is a Non-Linear Program (NLP) problem. Generally, there is no polynomial time algorithm for NLP problem. So, we propose an heuristic algorithm similar to the gradient descent method.

Figure 3.2 shows the heuristic search algorithm. For each $r_{i,j}$, we set the initial value of it to 1, which means there is no speed update at the corresponding edge, and constitute the set $R$ which has all edges in $G_\tau$ of the target program. We change $r_{i,j}$ as the amount of $\Delta r$, which is a very small number between 0 and 1, during the successive iterations in Figure 3.2. In each iteration, we first find the edge $(b_i, b_j)$ having the largest $\Delta E$. $\Delta E$ means the energy gain when the $r_{i,j}$ is changed to $r'_{i,j} = r_{i,j} \cdot \Delta r$. From the formula (3.1), we can represent $\Delta E$ as

$$\Delta E(r_{i,j}) = S_0 \cdot (1 - \Delta r^2) \cdot \sum_{p_m \in \Phi^{b_j}} prob(p_m) \sum_{k=j}^{n_m} C_{EC}(b_k) \cdot S(b_k)^2. \quad (3.7)$$

After we change the value of $r_{i,j}$, we should check whether the timing constraint of formula (3.5) is satisfied in spite of the increased execution time of $p_m$. If there is no deadline miss, we try again the same process.

When there is a deadline miss, we can choose two kinds of approaches. First approach is to admit no speed-up, that is, SURs are always smaller than 1. In this case, the clock speed only decreases as a target program executes. So, there is no chance to solve the deadline miss problem by increasing the clock speed at an edge after $(b_i, b_j)$. We should restore $r'_{i,j}$ to $r_{i,j}$ and eliminate the corresponding edge $(b_i, b_j)$ from the set of candidate edges, $R$. Second approach is to admit speed-up. In this case, since the clock speed can be increased by an edge after $(b_i, b_j)$, we can decrease $r_{i,j}$ despite of the deadline miss. To maintain the selected edge $(b_i, b_j)$ in $R$, we should find the other edge, say $(b_h, b_k)$, which has the smallest $\Delta\widehat{E}$. $\Delta\widehat{E}$ means the energy loss when the $r_{h,k}$ is changed to $r'_{h,k} = r_{h,k}\Delta\widehat{r}$ $(\Delta\widehat{r} > 1)$. After we change $r_{h,k}$, the potential energy gain should be checked whether $\Delta E$ is larger than $\Delta\widehat{E}$. If there is no edge satisfying such a condition, we restore $r'_{i,j}$ to $r_{i,j}$ and eliminate the corresponding edge $(b_i, b_j)$ from $R$.

This heuristic algorithm shows following features: (1) The nearest edge from the entry basic block is first selected from $R$ because it has the largest value of $\Delta E$. After the corresponding SUR of the selected edge is fixed, the following edges are processed. (2) The nearest edge from the exit basic block is first selected to increase its SUR. (3) In the constraint that SURs should be smaller than 1, $S(b_i)$ is determined to be close to the value under

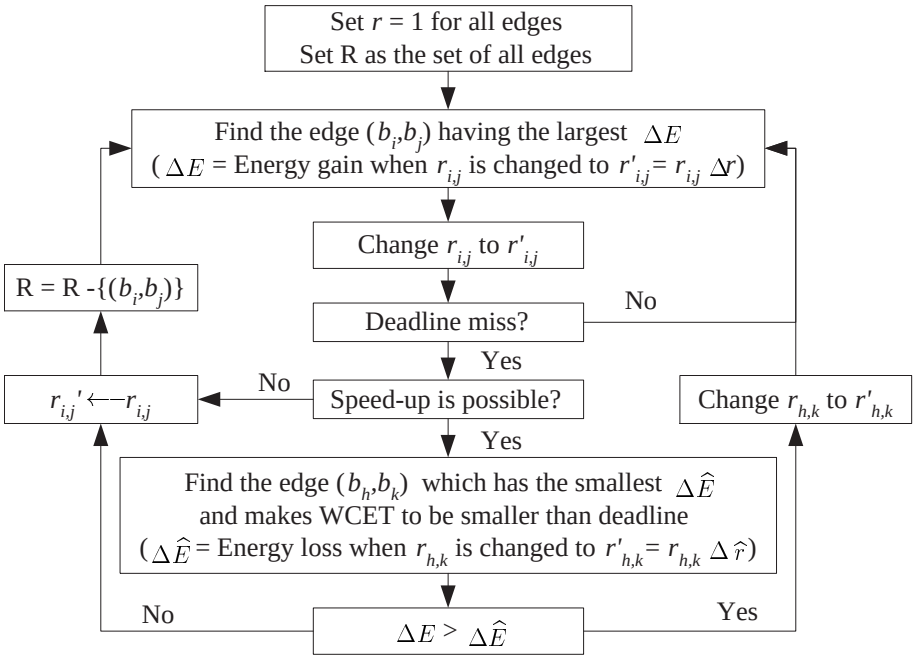



Figure 3.2: The heuristic search algorithm for IntraDVS problem.

which the target processor can execute $MAX_{p_j \in \Phi^{b_i}}(C_{EC}(p_j))$ cycles until the deadline. $MAX_{p_j \in \Phi^{b_i}}(C_{EC}(p_j))$ is the remaining worst-case execution cycle of $b_i$. Namely, the SURs are optimal to the remaining worst-case execution path. (4) Without such a constraint, the SURs are determined so that the average energy consumption is minimized satisfying the deadline constraint. In this case, the determined speed is near to the optimal speed value for the average-case execution path.

## 3.3 Voltage Scheduling Using Reference Path

The algorithm shown in Figure 3.2 has several problems. First, it requires a significant computation time. The computation time is proportional to the number of basic blocks and execution paths. Considering that general programs have at least hundreds of basic blocks, it is difficult to use the algorithm for real applications. Second, we should insert voltage scaling codes at most of edges in a program. This introduces a significant increase in code size. Therefore, we need more efficient algorithms.

From the features of the heuristic algorithm, we can modify the algorithm to the less complex algorithm. We predict an execution path which the control flow will follow. There can be several methods how to predict the execution path. A simple method is to use the WCEP. Once the execution path is predicted, we set the initial clock frequency and its corresponding voltage assuming that the task execution will follow the predicted execution path. We call the predicted execution path as the *reference path* because the clock speed is determined based on the execution path.

When the actual execution deviates from the (predicted) reference path (say, by a branch instruction), the clock speed can be adjusted depending on the difference between the remaining execution cycles of the reference path and that of the newly deviated execution path. If the new execution path takes significantly longer to complete its execution than the reference execution path, the clock speed should be *raised* to meet the deadline constraint. On the other hand, if the new execution path can finish its execution earlier than the reference execution path, the clock speed can be *lowered* to save the energy consumption. Once the actual execution takes a different path from the reference path, a new reference path is constructed starting from the deviated basic block.

In actual implementation of the IntraDVS, we do not need to maintain the reference path. To implement the IntraDVS algorithm efficiently, we identifies the appropriate program locations where the clock speed should be raised or lowered relative to the current clock speed using a static program-analysis technique. For run-time clock speed adjustment, it inserts voltage scaling codes to the selected program locations at compile time. The branching edges of the CFG, i.e., branch or loop statements, are the candidate locations for inserting voltage scaling codes because where the prediction miss for the reference path can be occurred. They are called as *Voltage Scaling Edges* (VSEs) because the clock speed and voltage are adjusted at these edges. At each VSE $(b_i, b_j)$, the clock speed is determined by the predicted remaining execution cycles (RPECs) of $b_i$, $C_{RPEC}(b_j)$. The value of $C_{RPEC}(b_j)$ depends on the prediction algorithm.

There are two issues in the IntraDVS. One is how to predict the reference path. Depending on the prediction method, the IntraDVS framework can

be implemented into different IntraDVS algorithms. In this dissertation, we adopt two kinds of reference paths, i.e. remaining worst-case execution path (RWEP) and remaining average-case execution path (RAEP). Based on the prediction method, there are two different algorithms, i.e. RWEP-based IntraDVS and RAEP-based IntraDVS. In the former, the clock speed is monotonically decreased at all the VSEs. This is correspond to the case speed-up is not admitted. On the contrary, in the latter, the clock speed may be increased as well at some VSEs. In this case, we classify VSEs into Up-VSEs and Down-VSEs. The clock speed is increased at an Up-VSE while it is decreased at a Down-VSE.

The second issue is how to select VSEs. This is to determine the voltage scaling points in the program code. The optimal points are the earliest points where we can detect the changes of the remaining predicted execution cycles. We use two kinds of selection algorithms, one is to use only the control flow information and the other is to use the data flow information in addition. Selecting VSEs, the timing overhead due to voltage transition and inserted codes should be considered. We provide the solutions for these issues in Chapters 4 and 5.

# Chapter 4

# IntraDVS Using Static Timing Analysis

## 4.1 RWEP-based IntraDVS Algorithm

In the RWEP-based IntraDVS, we use the remaining worst-case execution cycles (RWECs) $C_{RWEC}(b_i)$ for the predicted remaining execution cycles. This guarantees that all possible execution paths meet the deadline. Figure 4.1 shows an augmented CFG $G_P^{RWEP}$ with $C_{RWEC}(b_i)$ values for the RWEP-based IntraDVS. Using the static timing analysis tools, we can compute $C_{RWEC}(b_i)$ for each basic block $b_i$ and construct the graph $G_P^{RWEP}$ statically. To compute the worst-case execution cycles, we use follow equa-

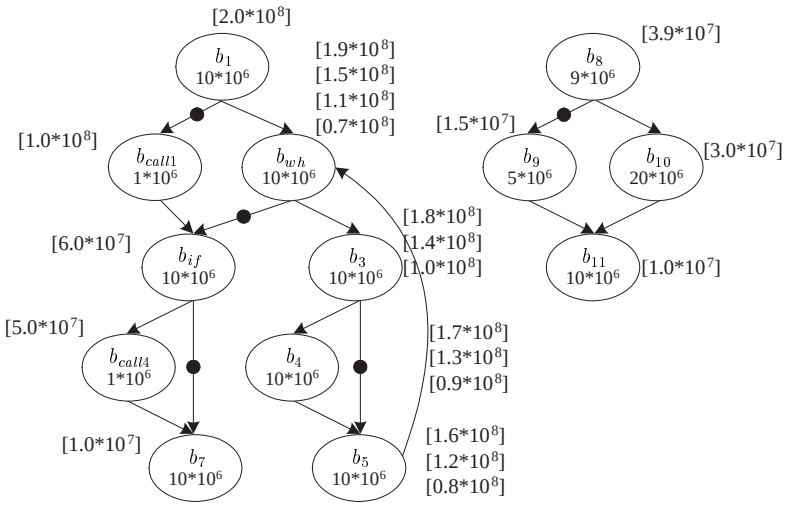



Figure 4.1: A RWEP-based CFG $G_P^{RWEP}$.

tions:

$$S : \textbf{if}(E) \ \ \textbf{then} \ \ S_1; \ \ \textbf{else} \ \ S_2.$$
$$WCEC(S) = WCEC(E) + max(WCEC(S_1), WCEC(S_2))$$
$$S : \textbf{while}(E) \ \ S_1.$$
$$WCEC(S) = (WCEC(E) + WCEC(S_1)) \cdot N_{max} + WCEC(E)$$

where $N_{max}$ means the maximum number of loop iterations.

For the basic blocks related to the **while** loop (i.e., $b_{wh}$, $b_3, b_4, b_5$), the corresponding nodes are associated with multiple $C_{RWEC}(b_i)$ values, reflecting the maximum three iterations of the **while** loop.

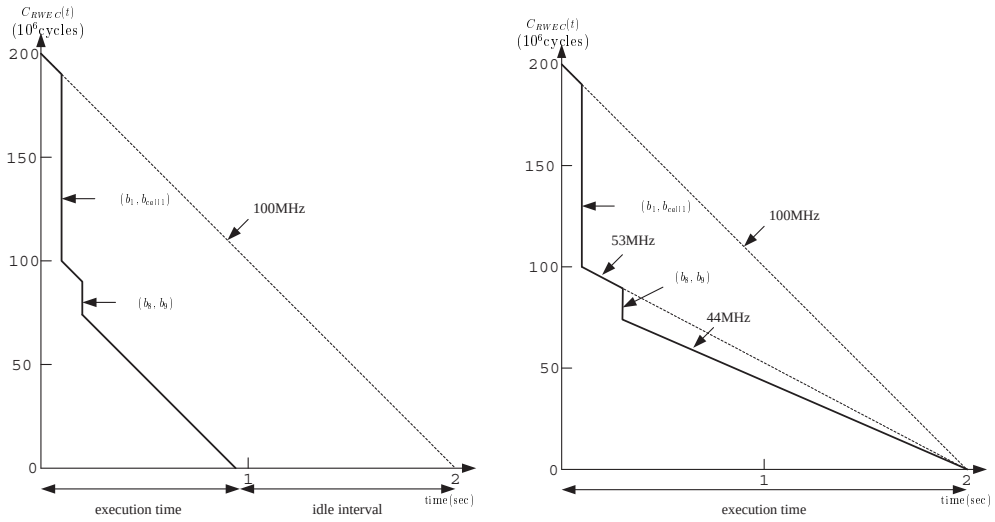
With the graph $G_P^{RWEP}$, we can identify VSEs that drops the remaining worst-case execution cycles *faster* than the current execution rate. For example, in Figure 4.1, five VSEs are identified, i.e., $(b_1, b_{call1})$, $(b_{wh}, b_{if})$, $(b_{if}, b_7)$, $(b_3, b_5)$ and $(b_8, b_9)$. In Figure 4.1, these edges are marked by the

symbol •. When the thread of execution control branches to the next basic block through one of VSEs, say $(b_1, b_{call1})$, the clock speed can be lowered because the remaining work is reduced by the difference between $C_{RWEC}(b_{wh})$ and $C_{RWEC}(b_{call1})$. By reducing the clock speed so that the $C_{RWEC}(b_{call1})$ cycles can be completed exactly at the deadline, the proposed technique always meets the required timing constraint. Since the voltage scaling decisions are made at compile time, there exists no run-time overhead directly related to the selection of voltage scaling edges. In addition, the compile-time static analysis procedure does not require special programmer's interventions other than ones typically required in developing normal hard real-time programs (e.g., the maximum number of loop iterations).

At the entry basic block $b_1$, the starting speed is set to $C_{WCEC}/D$, where $C_{WCEC}$ is the worst-case execution cycles of the whole program. When we denote $C_{RWEC}(t)$ as the remaining worst-case execution cycles at time $t$, $C_{RWEC}(t)$ is linearly decreased at the rate of clock speed along with the program execution, as far as the execution follows the worst-case execution path $p_{worst}$. However, if the execution deviates from the basic block $b_i$ in the worst-case execution path $p_{worst}$ to other basic block $b_j$ not included in $p_{worst}$, $C_{RWEC}(t)$ drops by the difference between $C_{RWEC}(b_i) - C_{EC}(b_i)$ and $C_{RWEC}(b_j)$ after the execution of $b_i$ is completed. Then, we adjust the clock speed at the time $t$, $S(t)$ as follows:

$$S(t) = \frac{C_{RWEC}(t)}{(D-t)} \tag{4.1}$$

Figure 4.2 shows how $C_{RWEC}(t)$ dynamically changes as the path $p_1$=($b_1$, $b_{call1}$, $b_8$, $b_9$, $b_{11}$,$b_{if}$, $b_{call4}$, $b_8$, $b_{10}$, $b_{11}$, $b_7$) of the example program $P$ shown in Figure 4.1 is executed. In Figure 4.2(a) where no speed scheduling is




(a) (b)

Figure 4.2: The changes of $C_{RWEC}(t)$ over different speed scaling algorithms: (a) no IntraDVS and (b) RWEP-based IntraDVS.

used, $C_{RWEC}(t)$ drops at two edges, $(b_1, b_{call1})$ and $(b_8, b_9)$. Since no speed scheduling is used, $C_{RWEC}(t)$ is decreased at the rate of 100 MHz, resulting in a slack time interval of 1.05 sec. Figure 4.2(b) shows the effect of speed scheduling for the same execution path assuming there is no voltage transition overhead. When the remaining worst-case execution cycles drops, the minimum processor speed that can complete the remaining program execution before the deadline also drops. Thus, processor speed is changed from 100 MHz to 53 MHz when $C_{RWEC}(t)$ drops right after the execution of $b_1$ is completed. When $C_{RWEC}(t)$ drops right after $b_8$, speed is also changed due to the same reason. $C_{RWEC}(t)$ is dropped vertically at VSEs in Figure 4.2. The number of the reduced cycles of $C_{RWEC}(t)$ at VSEs is denoted as $C_{saved}$.

**Theorem 1** *If we can use any real value of clock frequency $f$, $0 < f \leq f_{max}$, at a variable-voltage processor and there is no overhead time during the clock and voltage transition, the execution time of a task scheduled by the IntraDVS algorithm with a relative deadline $D$ is exactly $D$ when $WCEC/f_{max} \leq D$.*

*Proof.* The task starts with the initial speed $WCEC/D$. Assume the task meets a VSE at time $t$ but meets no VSE after the time $t$. The IntraDVS algorithm sets the clock speed $S(t)$ such that

$$S(t) = \frac{C_{RWEC}(t)}{(D-t)}$$

as shown in Equation (4.1). $C_{RWEC}(t)$ is smaller than or equal to the following value:

$$WCEC - t \cdot \frac{WCEC}{D}$$

So,

$$S(t) = \frac{C_{RWEC}(t)}{(D-t)} \leq \frac{WCEC}{D} \leq f_{max}.$$

Therefore, we can use the clock speed of $S(t)$ at time $t$. Then, the execution time of the task is

$$t + \frac{C_{RWEC}(t)}{S(t)} = t + (D-t) = D.$$

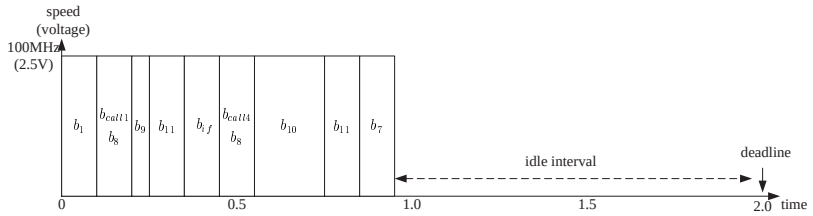
□

Theoretically, since the IntraDVS can fully exploit all workload-variation slack times, all of task executions are completed exactly at the deadline. However, some slack time can be generated in real variable-voltage processors even though we use the IntraDVS technique. This is because variable-voltage processors provide only finite numbers of clock/voltage levels and require voltage transition times to change the clock/voltage level. These two factors prevent IntraDVS from adjusting the clock/voltage level at all VSE candidates (i.e., branching edges).

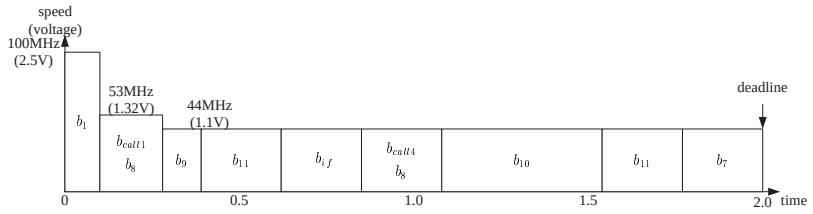
Figure 4.3 compares how the speed and voltage change depending on whether the IntraDVS is used or not. Assume that no energy is consumed in an idle state. When the execution follows the path $p_1$, the energy consumption ratio of Figure 4.3(b) to Figure 4.3(a) is 0.288. With the IntraDVS, the energy consumption is reduced by 71.2%.

## 4.2 Selection of Voltage Scaling Edges

To adjust the clock speed and voltage at run time, the voltage scaling edges should be selected at compile time considering the saved cycles and the




(a)




(b)

Figure 4.3: Speed and voltage changes: (a) without IntraDVS and (b) with the RWEP-based IntraDVS

overhead cycles. Observing its behavior, VSEs are classified into B-type VSEs and L-type VSEs.

### 4.2.1 B-type Voltage Scaling Edges

A B-type VSE corresponds to the CFG edge between two basic blocks that are part of conditional statements such as the `if` statement. For the `if` statement, the WCET is predicted to be the larger of two execution times, one for the `then` path and the other for the `else` path. Assume that the condition of the `if` statement is evaluated in $b_{cond}$, the `then` path starts from $b_{then}$ and the `else` path starts from $b_{else}$. If the condition of the `if` statement evaluates to true and the `then` path is shorter than the `else` path, $C_{RWEC}(t)$ is decreased by $(C_{RWEC}(b_{else}) - C_{RWEC}(b_{then}))$. In this case, the speed can

be decreased before the $b_{then}$ block is executed by a ratio of $\frac{C_{RWEC}(b_{then})}{C_{RWEC}(b_{else})}$. This value is a SUR and is represented by $r(b_{cond}, b_{then})$.

In adjusting clock/voltage at VSEs, several instructions are required other than voltage-changing instruction (`change_f_V`($f_{clk}$)). We denote the number of cycles needed for these extra instructions at a B-type VSE as $C_{VSO_B}$. The total number of overhead cycles $C_{overhead_B}$ for a B-type VSE, therefore, is given by $C_{VTO} + C_{VSO_B}$. The SUR $r(b_i, b_j)$ for a B-type VSE $(b_i, b_j)$ is calculated as follows:

$$r(b_i, b_j) = \frac{C_{RWEC}(b_j)}{C_{RWEC}(succ_{worst}(b_i)) - C_{overhead_B}} \tag{4.2}$$

where $succ_{worst}(b_i)$ is the basic block $b_k$ that is an immediate successor of $b_i$ and has the largest $C_{RWEC}(b_k)$ among all the successors of $b_i$. If $C_{RWEC}(b_j) \geq C_{RWEC}(succ_{worst}(b_i)) - C_{overhead_B}$, that is $r(b_i, b_j) \geq 1$, the edge $(b_i, b_j)$ is not selected as a VSE. For a VSE between $b_i$ and $b_j$, a SUR $r(b_i, b_j)$ is multiplied to the current speed before $b_j$ starts its execution. For example, assuming $C_{overhead_B}$ as 0, $S(b_{call1})$ in Figure 4.1 is changed from 100 MHz to 53 MHz (=100 MHz $\times \frac{100 \times 10^6}{190 \times 10^6}$).

**Theorem 2** *Using the IntraDVS algorithm with the speed update rule in Equation (4.2), the execution time of a task with the relative deadline $D$ is no more than $D$ at a variable-voltage processor which provides continuous clock frequency $f$ such that $0 < f \leq f_{max}$.*

*Proof.* Assume that a task has the worst-case execution cycles $W$. If the control flow meets a VSE $(b_i, b_j)$ at the time $t$ and $succ_{worst}(b_i)$ is $b_k$, the

speed is changed to

$$S(b_j) = S(b_i) \cdot \frac{C_{RWEC}(b_j)}{C_{RWEC}(b_k) - C_{overhead_B}}$$

where $S(b_i) \leq f_{max}$. Since there is no voltage transition at the edge $(b_i, b_k)$, $S(b_i)$ is same to $S(b_k)$. Assume that

$$S(b_i) = S(b_k) = C_{RWEC}(b_k)/(D - t - \delta) \tag{4.3}$$

where $(\delta \geq 0)$. If the first clock/voltage transition occurs at the VSE $(b_i, b_j)$, this assumption is true because $S(b_i) = W/D$ and $C_{RWEC}(b_k) = W \cdot (D - t)/D$. For the cases where $(b_i, b_j)$ is not the first VSE, the assumption will be shown to be true using $S(b_j)$.

Then, we can rewrite $S(b_j)$ as follows:

$$S(b_j) = \frac{S(b_i) \cdot C_{RWEC}(b_j)}{S(b_i) \cdot (D - t - \delta) - C_{overhead_B}} \tag{4.4}$$

During the clock and voltage transition, there is the overhead time $C_{overhead_B}/f_{max}$. If the VSE $(b_i, b_j)$ is the last VSE which the task meets during execution, the completion time is as follows:

$$\begin{aligned} & t + \frac{C_{overhead_B}}{f_{max}} + \frac{C_{RWEC}(b_j) \cdot (S(b_i) \cdot (D - t - \delta) - C_{overhead_B})}{S(b_i) \cdot C_{RWEC}(b_j)} \\ = \quad & t + \frac{C_{overhead_B}}{f_{max}} + (D - t - \delta) - \frac{C_{overhead_B}}{S(b_i)} \\ = \quad & D - (\delta + C_{overhead_B} \cdot (\frac{1}{S(b_i)} - \frac{1}{f_{max}})) \leq D \end{aligned} \tag{4.5}$$

If the task execution meets another VSE $(b_u, b_v)$ after $(b_i, b_j)$ at time $t'$ and $succ_{worst}(b_u)$ is $b_w$,

$$S(b_j) = \frac{C_{RWEC}(b_j)}{D - t - (\delta + \frac{C_{overhead_B}}{S(b_i)})} = \frac{C_{RWEC}(b_j)}{D - t - \delta'} \tag{4.6}$$

$$C_{RWEC}(b_w) = C_{RWEC}(b_w) - (t' - t) \cdot S(b_i) \tag{4.7}$$

From Equations (4.6) and (4.7),

$$S(b_j) = \frac{C_{RWEC}(b_w)}{(D - t' - \delta')} = S(b_w) \tag{4.8}$$

From Equation (4.8), we can know that the assumption of Equation (4.3) is true. Consequently, we can conclude that the execution time is smaller than $D$. □

### 4.2.2 L-type Voltage Scaling Edges

Although WCEC is predicted assuming that a loop will be iterated by the user-provided maximum number of loop iterations, the loop is generally iterated smaller times than the maximum loop bound. In this case, slack time occurs and clock speed can be scaled down. We call this type of scaling L-type scaling. L-type VSEs correspond to the loop-exit edges in a CFG. In the L-type scaling, the number of saved cycles $C_{saved}$ for a loop $l$ is given by

$$C_{saved}(l) = C_{WCEC}(l) \cdot (N_{worst}(l) - N_{exec}(l)) \tag{4.9}$$

where $C_{WCEC}(l)$ is the number of worst-case execution cycles to execute the loop $l$ once, $N_{worst}(l)$ is the number of user-provided maximum loop bound value for the loop $l$, and $N_{exec}(l)$ is the number of actual loop iterations measured at run time. Consider the edge $(b_{wh}, b_{if})$ in Figure 4.1. Assuming $N_{exec}(l) = 1$, and $C_{overhead_L} = 0$, $S(b_{if})$ is updated as follows:

$$
\begin{aligned}
S(b_{if}) &= S(b_{wh}) \cdot \frac{C_{RWEC}(b_{if})}{C_{RWEC}(b_{if}) + C_{saved}(l) - C_{overhead_L}} \qquad (4.10)\\
&= S(b_{wh}) \cdot \frac{60 \times 10^6}{60 \times 10^6 + 40 \times 10^6 \times (3-1)}\\
&= S(b_{wh}) \cdot r(b_{wh}, b_{if})
\end{aligned}
$$

When $S(b_{wh})$ is 100 MHz, $S(b_{if})$ is reduced to 43 MHz before executing $b_{if}$.

Unlike a B-type VSE, calculating the SUR for an L-type VSE requires the run-time information such as $N_{exec}(l)$[1]. The SUR may be larger than 1 depending on the value of $N_{exec}(l)$ and $C_{overhead_L}$. To avoid this problem, we select an L-type VSE in two phases. First, we select a loop-exit edge of a loop $l$ as an L-type *candidate* VSE if $C_{WCEC}(l) > C_{overhead_L}$, which means that if $N_{exec}(l) < N_{worst}(l)$, the SUR is always smaller than 1. When $N_{exec}(l) = N_{worst}(l)$, the speed is not changed but the timing behavior of an original program is changed due to the code inserted to check whether $N_{exec}(l) = N_{worst}(l)$ or not. Among the L-type candidates, we choose the final L-type VSEs by the algorithm explained in Section 4.4. Although L-type VSEs are more complicated than B-type VSEs, the contribution of L-type VSEs on the overall energy reduction is much bigger, since slack times from loop executions are generally much larger than those from conditional statements.

[1] Note that the selection of L-type VSEs are done in compile time. The run-time information such as $N_{exec}(l)$ is necessary when calculating the speed update ratio.

### 4.2.3 VSEs in Loops or Functions

The remaining worst-case execution cycles of nodes in loops or functions are changed depending on the iteration number of the loop or the location from which the function is called. So, we cannot represent the SUR of a VSE in loop or function as a fixed value. Instead, the SUR is represented by a formula using the run-time informations as variables. For example, in Figure 4.1, the SUR of $(b_3, b_5)$ is represented as

$$r(b_3, b_5) = \frac{C_{EC}(b_5) + C_{EC}(b_{wh}) + C_{WCEC}(l) \cdot (N_{worst}(l) - N_{exec}(l)) + C_{RWEC}(b_{if})}{C_{EC}(b_4) + C_{EC}(b_5) + C_{EC}(b_{wh}) + C_{WCEC}(l) \cdot (N_{worst}(l) - N_{exec}(l)) + C_{RWEC}(b_{if})}$$

where $l$ is the loop in Figure 4.1. $r(b_3, b_5)$ is 0.957 at the first iteration but 0.947 at the second iteration of the loop $l$.

For a basic block $b_i$ in the function $f$, we define $C_{RWEC}(b_i)$ by the remaining worst-case execution cycles of $b_i$ in the scope of the function $f$ because the remaining execution cycles of $b_i$ in the overall program are changed depending on where the function $f$ is called. Therefore, we need $C_{RWEC}(b_j)$ to get the speed update ratio of $b_i$, where $b_j$ is the return point of the function $f$. For example, the SUR of $(b_8, b_9)$ in Figure 4.1 can be calculated as

$$r(b_8, b_9) = \frac{C_{RWEC}(b_9) + C_{RWEC}(b_{if})}{C_{RWEC}(b_{10}) + C_{RWEC}(b_{if})}$$

when the function is called from $b_{call1}$.

However, denoting the SUR as the formula using the run-time informations is an obstacle to select VSEs *statically* and insert the voltage scaling

code. To avoid this problem, we propose a simple solution as follows. We denote all the possible SUR values of edge $(b_i, b_j)$ as $r^1_{i,j}, \cdots, r^n_{i,j}$. For example, the VSE candidate $(b_3, b_5)$ in Figure 4.1 has three possible SUR values, $r^1_{3,5}, r^2_{3,5}, r^3_{3,5}$ because it can be executed three times in the loop. If any $r^k_{i,j}$ is smaller than 1, all other $r^h_{i,j}$ are also smaller than 1. The is based on the following simple formula.

$$\frac{a+l}{b+l} < 1 \text{ and } \frac{a-l}{b-l} < 1 \quad \text{if} \quad \frac{a}{b} < 1, a > l, \text{ and } b > l.$$

If we represent any SUR $r^k_{i,j}$ of a VSE candidate in loop or function as $\frac{a}{b}$, the other SUR $r^h_{i,j}$ of the VSE can be denoted by $\frac{a+l}{b+l}$ or $\frac{a-l}{b-l}$, where $l$ is the execution cycles between two instances of the VSE candidate. Since $\frac{a+l}{b+l}$ or $\frac{a-l}{b-l}$ is also smaller than 1 if $\frac{a}{b} < 1$, we can say that $r^h_{i,j}$ is also smaller than 1 if $r^k_{i,j} < 1$. This means that it is sufficient to check only one instance of the VSE candidate to know whether an edge in loop or function can be selected to a VSE.

## 4.3 Code Transformation

Since the SUR value of a VSE may not be determined as a fixed value at compile time as mentioned, the target program should calculate the SUR using the VSE information at run time. The VSE information consists of 5 elements, i.e., `Type`, `preRWEC`, `postRWEC`, `loopWCEC`, and `MI`. The `preRWEC` and `postRWEC` are $C_{RWEC}(b_i) - C_{EC}(b_i)$ and $C_{RWEC}(b_j)$ for a VSE $(b_i, b_j)$, respectively. These values are used to calculate the speed update ratio. For

B-type VSE, the SUR is `postRWEC/prePWEC`. For L-type VSE, `loopWCEC` and `MI` are used additionally. The `loopWCEC` is the worst-case execution cycles of L-type VSE's corresponding loop ($C_{WCEC}(l)$ in Equation (4.2)). `MI` is the maximum number of loop iteration.

Figure 4.4 shows the code examples for VSEs. Voltage scaling codes for VSE includes a code segment that calculates the SUR and updates the current speed by multiplying with the SUR (code B and code L in Figure 4.4). These voltage scaling codes are different in B-type VSE and L-type VSE. In L-type VSE, the voltage scaling codes calculate the speed update ratio using the iteration number ($LoopIterNum(b_{wh})$) of the corresponding loop. Therefore, several codes are needed to maintain the iteration number of the loop (code 2 and 3 in Figure 4.4).

To calculate $C_{RWEC}(b_i)$, where $b_i$ is in the loop $l$, a code is needed to transfer the $C_{post}(l)$ (the remaining execution cycles after a loop $l$) to the loop (code 1 in Figure 4.4). The same code is inserted for functions (code 5 in Figure 4.4). As the loop or function can be nested, $C_{post}(l)$ is saved at a stack. When the loop or function is completed, the stack index (top) is decreased (code 4 and 6 in Figure 4.4).

## 4.4 Overall Selection Algorithm

While voltage scaling codes for B-type VSEs does not increase $C_{WCEC}$ of a given program, those for L-type VSEs can increase $C_{WCEC}$ depending on the number of loop iterations executed. If a loop iterates its maximum number of iterations (i.e., the maximum number of loop iterations given

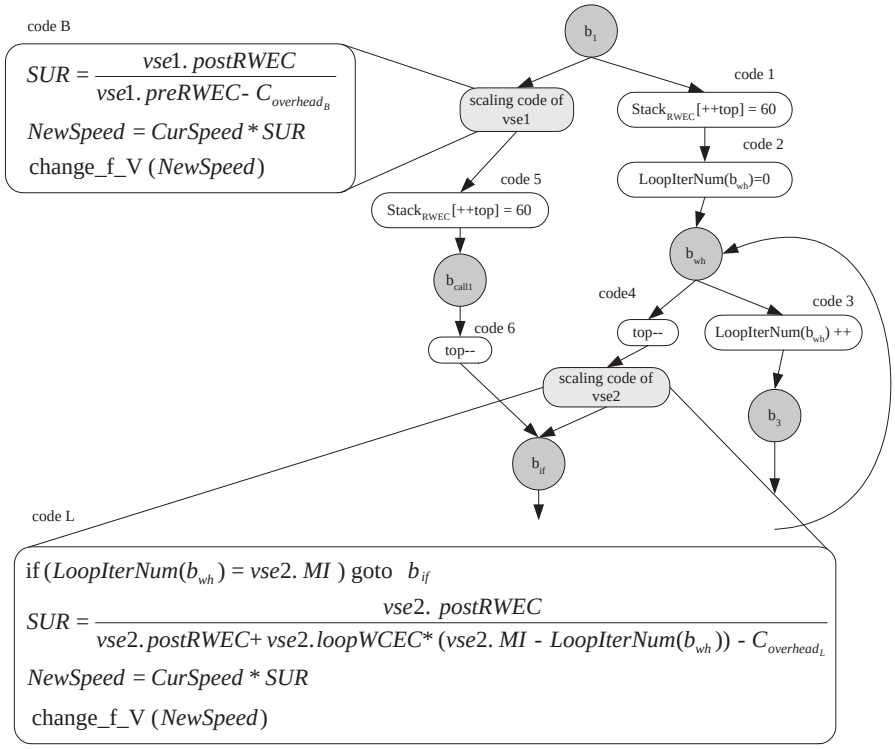



Figure 4.4: Code generation for VSEs.

by user) and the loop exit edge was selected as a candidate L-type VSE, $C_{WCEC}$ of the program will increase by the number of cycles to execute the code checking the number of loop iterations. This increase, if accumulated, may make the modified program violate the timing constraint of the original program.

To avoid this problem, we select the final L-type VSEs from the candidate L-type VSEs by the algorithm shown in Figure 4.5. Assuming that the target processor can execute $M$ cycles at its full speed within the given deadline interval, we first select the candidate VSEs using the selection algorithms in the previous section. Then we calculate the increase of worst-case execution cycles. To calculate the increase, we should know how much cycles are needed for voltage scaling codes. We denote these values as $C_{inc}(B)$[2] and $C_{inc}(L)$. If the total worst-case execution cycles $C_{WCEC}$ is larger than $M$, we exclude some candidate VSEs until the increase in $C_{WCEC}$ will be smaller than $M$. How many candidate VSEs should be excluded is easily known with the values of $C_{inc}(B)$ and $C_{inc}(L)$. The VSE with little effect on energy reduction is preferred to be excluded. $C_{RWEC}(b_i)$'s are recomputed after some candidate VSEs are excluded. When $C_{WCEC} < M$ is satisfied, the final VSEs are determined.

Since the inserted voltage scaling codes change the timing behaviors of the target program, we should analyze the timing information of the program again after the VSE selection. The change of the timing information affects the speed update ratios of the selected VSEs. So, we recompute the

[2]In the RAEP-based IntraDVS to be explained in Chapter 5, the B-type VSEs can increase the worst-case execution cycles. So, we consider the B-type VSEs as well in this algorithm.

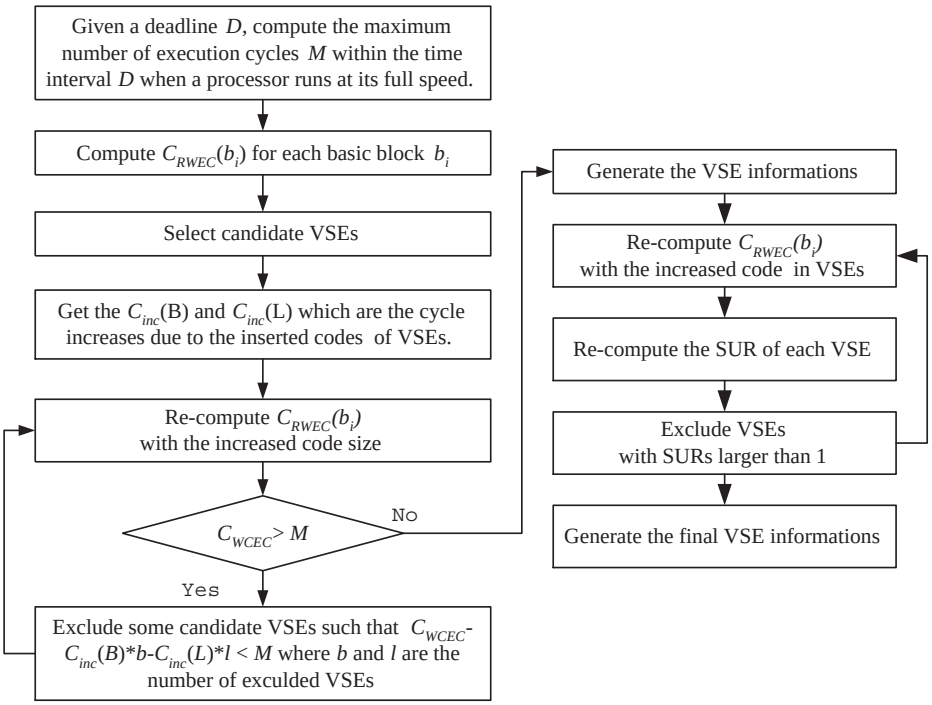



Figure 4.5: Overall VSE selection algorithm.

speed update ratio of each VSE. If the speed update ratio of a VSE is larger than 1, we should exclude the VSE. This examination continues until there is no VSE which has a SUR larger than 1.

Based on the proposed IntraDVS algorithm, we have developed the Automatic Voltage Scaler (AVS), a software tool that *automates* the development of hard real-time programs on a variable-voltage processor. AVS takes a *DVS-unaware* (thus regular) program $P$ and its timing requirements as inputs, and produces a *DVS-aware* low-energy program $P_{DVS}$ that satisfies the same timing requirements. The converted program $P_{DVS}$ contains voltage scaling code that handles all the idiosyncrasy of scaling clock/voltage on a variable-voltage processor. Using AVS, DVS-unaware hard real-time

programs can be converted to DVS-aware low-energy programs in a completely transparent fashion to software developers. The detail description of AVS is provided in Appendix B.

## 4.5 Experiments

To evaluate the energy reduction performance of the RAEP-based IntraDVS algorithm, we have experimented with both a simulation and a real DVS-enabled systems. The experimental results on a real DVS-enabled system are described at Section 4.5.3.

### 4.5.1 Experiments with Artificial Workloads

We first experimented with artificial workloads. An artificial workload of a task can be represented by the tuple $(W, C, \Psi)$, where $W$ is the WCEC (= deadline) and $C$ is the real execution cycles. $\Psi$ is the set of slack informations denoted by $\Psi = \{(t_1, \omega_1), \cdots, ((t_n, \omega_n))\}$ where $t_i$ and $\omega_i$ are the location and the size of the $i$-th intra-slack respectively. For example, the artificial workload $(10, 6, \{(1, 2), (3, 2)\})$ means that the task consumes only 6 cycles and the 4 cycles of slack times are identified after 1 cycle execution and 3 cycles execution. The sum of slack sizes in $\Psi$ $(\sum_1^n \omega_i)$ is same to $W - C$.

We assume that the execution cycles of a task is drawn from a random Gaussian distribution with mean, denoted by $m$, and standard deviation,

denoted by $\sigma$, given by

$$m = \frac{BCEC + WCEC}{2} \quad \text{and} \quad \sigma = \frac{WCEC - BCEC}{6}.$$

We varied the B/W ratio (=BCEC/WCEC) from 0.1 to 0.9. The slack locations ($t_i$) are assumed to be distributed uniformly. The slack sizes ($\omega_i$) are generated by an exponential distribution function with a mean $\lambda$. Figure 4.6 shows the normalized energy consumption of the RWEP-based IntraDVS varying the B/W ratio and the average slack size ($\lambda$). The graph also shows the energy performance of the optimal DVS, which uses the single clock speed of $S = C/W$. The energy consumption decreases as the slack size increases. This is because the small slack times are excluded from VSEs by the IntraDVS. The change of energy consumption is obvious when the B/W ratio is small. But, the difference with the energy performance of optimal DVS is also large when the B/W ratio is small.

Figure 4.7 shows the energy performance of RWEP-based IntraDVS under various distribution functions for slack information. We used four kinds of distributions for the slack locations, i.e., L, U, N and H. N and U mean the normal distribution and the uniform distribution respectively. H and L mean the skewed distributions. While most of slack times are identified at the front part of a program execution in L distribution, they are identified at the rear part of a program execution in H distribution. The energy performances are superior in the L distribution. This is because we can know the workload at the front part of a program. Extremely, if we can know all the slack information at the start of a program, the IntraDVS shows the same energy performance of the optimal DVS. However, the slack locations in general applications follow the uniform distribution.

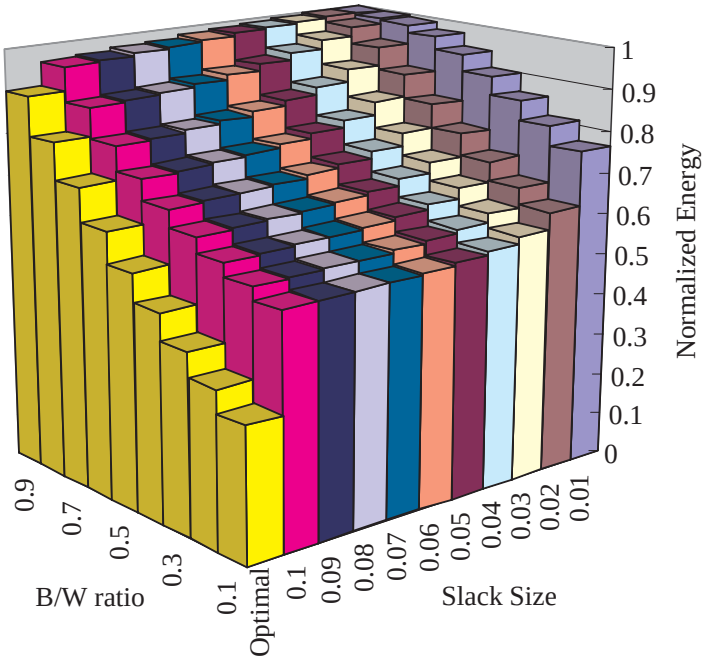



Figure 4.6: Normalized energy consumptions of the RWEP-based IntraDVS (varying the B/W ratio and the slack size).

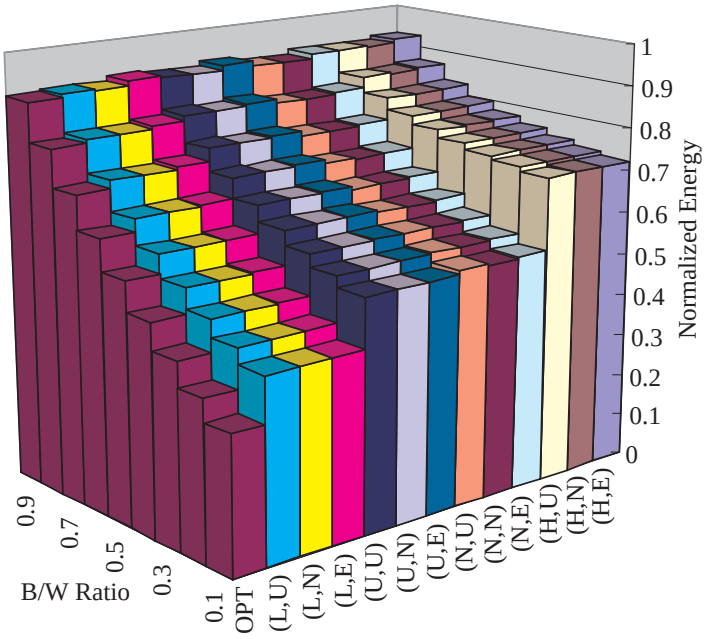



Figure 4.7: Normalized energy consumptions of the RWEP-based IntraDVS (varying the B/W ratio and the slack distribution).

For the slack size, we used three kinds of distributions, i.e., U (Uniform), N (Normal) and E (Exponential). The energy performances under the exponential distribution are worse than the results under the normal distribution or the uniform distribution. This is because the number of the large slack times is small under the exponential distribution. The slack times in general applications follow the exponential distribution.

### 4.5.2 Experiments with Real Applications

We have experimented with software MPEG-4 video encoder and decoder. For a simulation, we developed an energy simulator for the simulation exper-

iments. The energy simulator takes an assembly program and its execution trace as inputs, and calculates total energy consumption by emulating the program execution on the target variable-voltage processor. We assume that both DVS-aware and DVS-unaware systems enter into a power-down mode when the system is idle. Supply voltage for a given clock frequency is obtained from $f_{clk} = 1/T_D \propto (V_{dd} - V_T)^\gamma / V_{dd}$ [24] where $V_{dd}$, $V_T$, and $\gamma$ are assumed to be 2.5 V, 0.5 V, and 1.3, respectively. Since recent frequency synthesizers and DC-DC converters achieve clock/voltage transition time of less than 200 $\mu$sec, clock/voltage transition overhead $C_{VTO}$ is assumed to be 0$\sim$20,000 cycles, corresponding to 0$\sim$200 $\mu$sec of transition time with 100 MHz clock frequency. For non-zero $C_{VTO}$ values, the processor stops its execution and enters into a power-down mode during clock/voltage transition.

Figures 4.8(a) and 4.8(b) show the energy consumption of the AVS-converted MPEG-4 encoder and decoder programs, respectively. Simulated results were normalized over the energy consumption of the original program running on a DVS-unaware system with the power-down mode. It was assumed that the power-down mode consumes no energy. The AVS-converted MPEG-4 encoder and decoder programs consume less than 38% and 29% of the original program, respectively. Figures 4.8(a) and 4.8(b) also show the number of voltage transitions which represents how many times voltage scaling code were executed during the program execution. When $C_{VTO} <$ 3,000 cycles (=30 $\mu$sec) in the MPEG-4 encoder, the number of voltage transitions decreases sharply, and the energy consumption increase rapidly. When $C_{VTO} >$ 5,000 cycles (=50 $\mu$sec) in the MPEG-4 encoder, the energy consumption does not increase rapidly. This is because the number of dis-

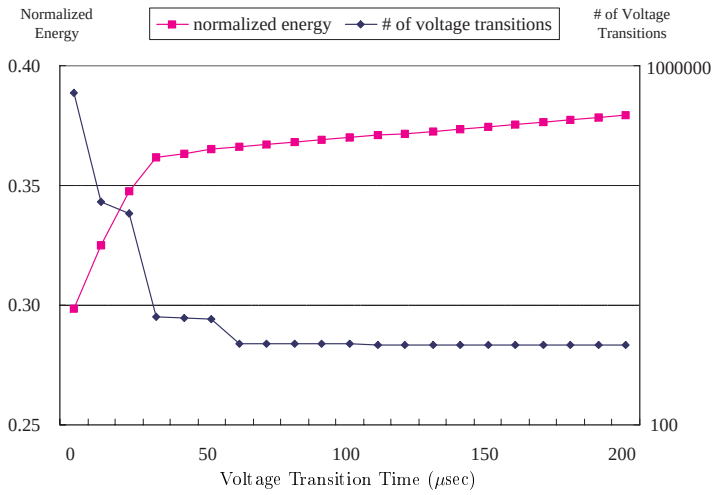
carded voltage scaling edges (due to clock/voltage transition overhead) is small.

The number of VSEs, which represents how many copies of voltage scaling codes were inserted into the AVS-converted program, indicates the degree of code size increment by inserting voltage scaling codes using in-line expansions. For the AVS-converted MPEG-4 encoder and decoder programs, about 20 VSEs are inserted when $C_{VTO} > 5{,}000$ cycles, meaning that insertion of voltage scaling code hardly increases the total code size. This is because a small number of voltage scaling edges occupies quite a large portion of the total power reduction.

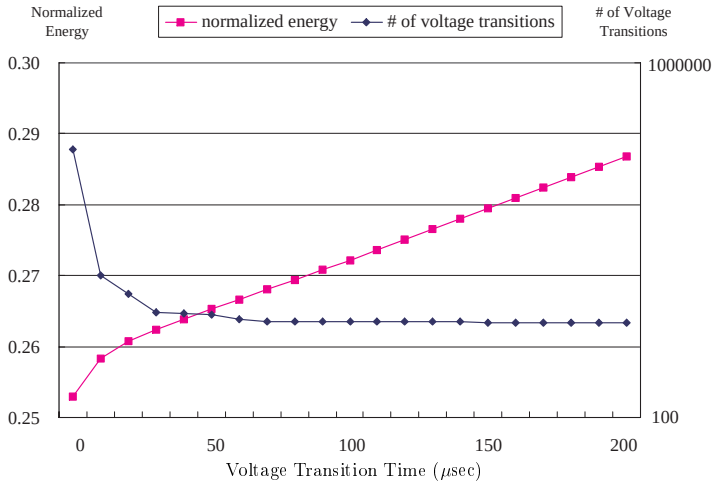
In the proposed IntraDVS algorithm, there are some unselected VSEs because VSE is selected or neglected by considering the transition time overhead of speed change. Unfortunately, the slack times generated by these unselected VSEs cannot be exploited by the IntraDVS algorithm. This limitation comes from the fact that the proposed algorithm does not use the run-time timing information. We call such a speed assignment an *off-line* speed assignment method.

The described IntraDVS can be improved if we can get an elapsed time at run time. In order to reclaim these slack times, the new clock speed should be set to the remaining cycles divided by the remaining time ( = deadline-elapsed time). This method is called an *on-line* speed assignment method. For the on-line speed assignment method, a target system should support efficient real-time counter accesses to get the elapsed time at run time[3].

[3]Although many embedded processors for real-time applications have an on-board RTC (real time clock), the resolution of RTC can be too low to use for IntraDVS.




(a) MPEG-4 encoder




(b) MPEG-4 decoder

Figure 4.8: Normalized energy consumption and the number of voltage transitions of the AVS-converted MPEG-4 encoder and decoder programs.

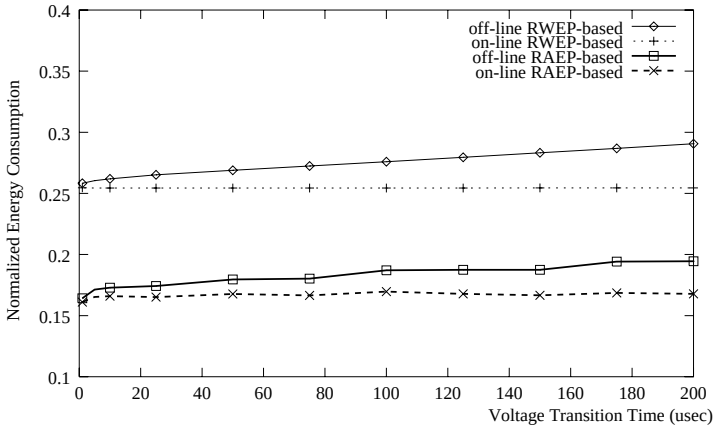



Figure 4.9: Normalized energy consumptions of the on-line and off-line speed assignment methods (varying the threshold value).

When the voltage transition overhead is small and the number of unselected VSEs is small, the off-line assignment method work relatively well compared to the on-line assignment method. On the other hand, when the voltage transition overhead is large, the on-line speed assignment becomes more effective than the off-line speed assignment, because the number of unselected VSEs increases.

Figure 4.9 shows the energy consumption of the off-line speed assignment and the on-line speed assignment. We assumed that on-line speed assignment does incur additional 40 overhead cycles. When the voltage transition time is small, there is little differences in energy consumption between the on-line and off-line speed assignment methods. However, the difference increases up to 10% as the voltage transition time increases because a large voltage transition time deselects more VSE candidates.

Moreover, the system call to access a RTC may incur a large overhead. So, we specify this assumption.

### 4.5.3 Experiments on a Real DVS-enabled System

For experiments on a real DVS-enabled System, we used *Itsy pocket computer* v2.6 from Compaq [28] as our experimental platform. The platform is equipped with a StrongARM SA-1100 processor as a main processor. The SA-1100 processor uses the phase-locked loop (PLL), allowing to change the CPU core frequency to one of 11 levels between 59.0 MHz and 226.4 MHz. Furthermore, Itsy v2.6 has a programmable core voltage regulator; supply voltage can scale to one of 30 levels between 1.00 V and 2.00 V. The overhead time to change the clock and voltage level is different depending on the current and target value of clock level, and is 189 $\mu$sec at maximum. Itsy runs the Linux operating system (ver. 2.0.30) with a kernel support for dynamic voltage scaling. Applications can change the clock frequency and supply voltage by the ioctl system call to the "/dev/clkspeed" device file.

Since it is difficult to establish the execution time model of a StrongARM SA-1100, we analyzed the timing behavior of a target application by the hybrid method which uses both the static analysis technique and the execution time profiling on the Itsy platform. Figure 4.10 shows the flow of experiment for Itsy platform. The target application (e.g., mpeg4dec.c) is compiled into an assembly code (mpeg4dec.s) by the arm-linux-gcc compiler. The *Static Analyzer* selects the execution time profiling points by the static timing analysis technique. It enables the efficient profiling for the execution time by choosing only small number of candidate VSEs from the target application. The *Static Analyzer* uses the assumption of one cycle for each instruction for the timing analysis. The *Profiler* generates the annotated assembly code (mpeg4dec-profile.s) which has the execution

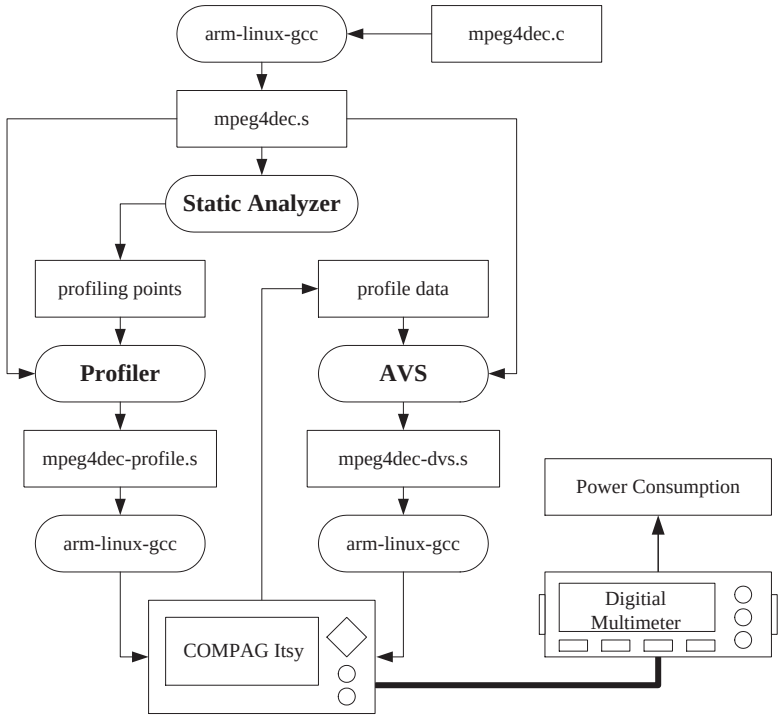



Figure 4.10: The experiment environments for Itsy platform.

time profiling code at the location specified by the *Static Analyzer*. The profile-enabled assembly code is compiled and executed at the Itsy platform. During the execution, the target application outputs the profile data which has the timing informations of the candidate VSEs. Using this data, AVS analyzes the timing informations of candidate VSEs, selects the VSEs and generates the DVS-aware assembly code (mpeg4dec-dvs.s). At VSEs in the DVS-aware assembly code, the DVS function is called to adjusted the clock speed and voltage. The digital multimeter measures the power consumption of Itsy system while the DVS-aware code is executed at the Itsy platform.

We have experimented with the same MPEG-4 programs used at the

simulation experiments. We assumed that each task processes 10 frames (IPPPPIPPPP) before deadline[4]. Table 4.1 shows the comparison between DVS-aware programs and DVS-unaware programs. The energy reductions are 49% and 65% for the decoder and the encoder, respectively. The execution times of DVS-aware programs are shorter than WCETs of them because there are unselected VSEs and the Itsy platform provides discrete clock and voltage levels. Especially, when the adjusted clock speed reaches at 54 MHz which is the lowest clock speed, the speed adjustments at VSEs do not occur.

For DVS-aware programs, Table 4.1 also shows the number of selected VSEs and the number of functions and loops where management codes (such as code 1-6 in Figure 4.4) are inserted. Since these numbers are small, we can conclude the additional overhead for IntraDVS is little. The AVS does not significantly increase the WCEC and the code size of target programs as shown in Table 4.1. We can also know each VSE and management code requires very small instructions in Itsy platform as shown in Table 4.2[5].

Figure 4.11(a) and (b) show the graphs of power consumption measured during the executions of the DVS-aware MPEG-4 program and the DVS-unaware MPEG-4 program. The digital multimeter sampled the power consumption at the period of 20 msec. The DVS-aware programs set the clock and voltage to the maximum level at the start of the execution, and reduce the clock and voltage at VSEs. The DVS-unaware programs execute

[4]Due to the low resolution of multimeter, we increased the size of a task artificially for a better observation.

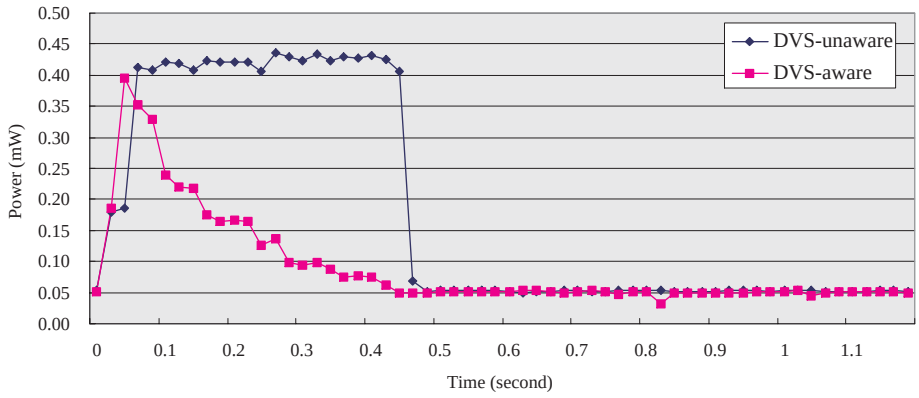
[5]Since the number of instructions for VSEs are different depending on the location, we show the average values.

Table 4.1: DVS experiments on Itsy.

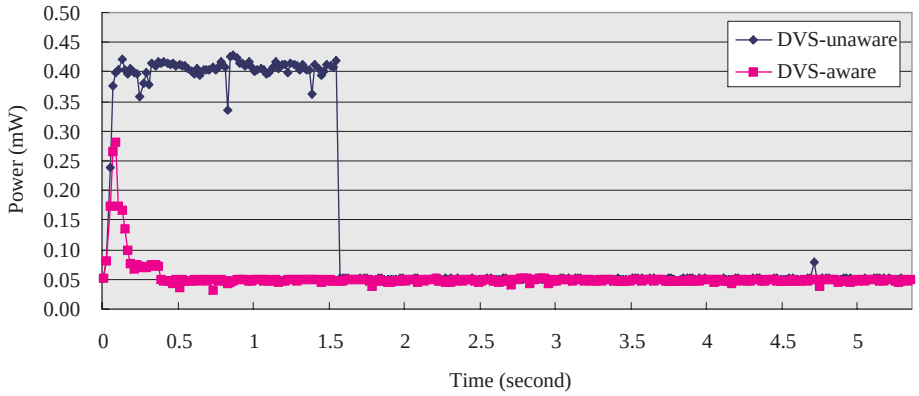
| Factors | | MPEG-4 Decoder | | MPEG-4 encoder | |
|---|---|---|---|---|---|
| | | DVS-aware | DVS-unaware | DVS-aware | DVS-unaware |
| Energy (mJ) | | 0.11 | 0.22 | 0.28 | 0.81 |
| Normalized Energy | | 0.51 | 1 | 0.35 | 1 |
| Execution Time (sec) | | 1.18 | 0.46 | 5.34 | 1.54 |
| Normalized WCEC | | 1.0003 | 1 | 1.003 | 1 |
| Selected VSEs | B-VSE | 2 | - | 1 | - |
| | L-VSE | 1 | - | 2 | - |
| Management Code | Function | 1 | - | 3 | - |
| | Loop | 2 | - | 5 | - |
| Normalized Code Size | | 1.026 | 1 | 1.027 | 1 |

Table 4.2: Management code overhead.

| Management Code | Code Number in Figure 4.4 | Number of Assembly Instructions |
|---|---|---|
| Loop Enter | code 1 and 2 | 23 |
| Loop Header | code 3 | 12 |
| Loop Exit | code 4 | 12 |
| Function Enter | code 5 | 10 |
| Function Return | code 6 | 8 |
| B-type VSE | code B | $\sim$ 200 |
| L-type VSE | code L | $\sim$ 200 |




(a) MPEG-4 decoder




(b) MPEG-4 encoder

Figure 4.11: Power estimation of MPEG-4 program.

at the full speed (and the maximum power) until completion and has the idle interval. This experiments verify the effectiveness of the IntraDVS technique on a real variable-voltage system.

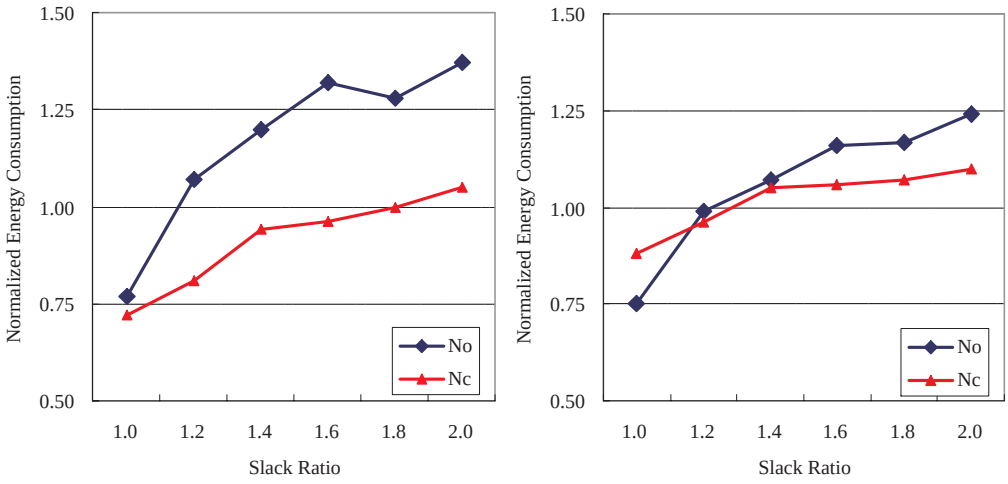
### 4.5.4 Comparisons of IntraDVS Algorithms

We have evaluated the energy efficiency of our reference path-based IntraDVS algorithm (`IntraDVS-P`) and the stochastic IntraDVS algorithm (`IntraDVS-S`) [30, 38] using an MPEG-4 video decoder and an MPEG-4 video encoder. The execution times of both the MPEG-4 decoder and encoder were assumed to follow a normal distribution $N_o = N(m_1, (\frac{m_2}{6})^2)$ where $m_1 = \frac{1}{2} \times$WCET and $m_2 = \frac{9}{10} \times$WCET.

Since the energy efficiency of `IntraDVS-S` largely depends on the slack ratio[6] given in the on-line phase and the accuracy of the execution time distribution used in the off-line profiling, we performed experiments varying these two factors. Figure 4.12 shows the relative energy consumption ratio of `IntraDVS-S` over `IntraDVS-P`. If the ratio is larger (smaller) than 1, `IntraDVS-S` performs better (worse) than `IntraDVS-P`. In Figure 4.12, the $N_o$ line represents the case when the actual execution times follow the assumed $N_o$ distribution. The $N_c$ line indicates the case where the actual execution times follow different normal distributions from the assumed $N_o$, where $N_c = N(1.5 \cdot m_1, (\frac{m_2}{7})^2)$.

When the slack ratio is less than 1.2, `IntraDVS-P` outperforms `IntraDVS-S` because `IntraDVS-P` spends more time in the lower speed region than `IntraDVS-S`. When the slack ratio is increased, `IntraDVS-S` spends more time in the lower speed region than `IntraDVS-P`. Figure 4.12 also shows that `IntraDVS-P` works better than `IntraDVS-S` when the distribution of actual execution times is significantly different from the assumed distribution, as shown in the $N_c$ line.

[6]The slack ratio is defined as the ratio of the assigned execution time to WCET.




(a) MPEG-4 decoder (b) MPEG-4 encoder

Figure 4.12: Energy consumption ratio of `IntraDVS-P` and `IntraDVS-S`.

Another comparisons between two IntraDVS algorithms have been presented by Gruian [41]. Since the energy performance of our IntraDVS algorithm is dependent on the internal task structure, two IntraDVS algorithms were compared varying the unawareness factor, which represents how early we can know the exact value of the execution cycles of a task during the task execution. When the unawareness factor is small (large) (i.e., VSEs are located close to the entry (exit) block), the performance of our IntraDVS is better (worse) than that of the stochastic IntraDVS.

### 4.5.5 Comparisons of InterDVS and IntraDVS

Although the IntraDVS algorithm is mainly proposed for single-task environments, it is also useful for multi-task environments. When a task scheduler assigns a time slot and its speed for each task at run time, our proposed

algorithm can additionally adjust the execution speed at the intra-task level. The proposed IntraDVS algorithm forces each task to use only the time slot assigned by the OS scheduler in multi-task environments. When an InterDVS algorithm determines a time slot for a task at run time, the IntraDVS has only to adjust the initial start speed of the task based on the assigned time slot.

To evaluate the feasibility of the IntraDVS algorithm in the multi-task environments, we compared the energy performance of the IntraDVS algorithm with several InterDVS algorithms. Figure 4.13 shows the normalized energy consumptions of two typical InterDVS algorithms, i.e. `lppsEDF` [31] and `DRA` [10], over the RWEP-based IntraDVS algorithm. The videophone task set shown in Table 1.1 and randomly generated task sets with 2, 3, 4, and 5 tasks were used in the experiment. Note that the randomly generated task sets do not represent normal program execution. Also note that they are not favorable to the IntraDVS, because they don't have a dominant task as in the videophone task set.

As shown in Figure 4.13, IntraDVS outperforms InterDVS in the videophone task set. In the randomly generated task sets, IntraDVS defeats `lppsEDF`, but it shows similar or worse performance with `DRA` when it has 3 or more tasks. This is because `DRA` uses a more sophisticated algorithm to utilize the slack time. Especially, when the number of task is 5, `DRA` algorithm shows a better performance than IntraDVS. From above results, we can know that IntraDVS outperforms InterDVS even in multi-task environments regardless of the existence of dominant tasks when the number of tasks is small. However, it could be better to use InterDVS when the real-time system has many tasks.

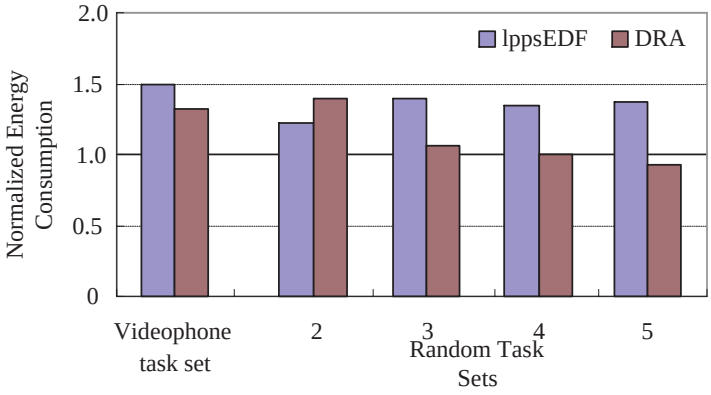



Figure 4.13: Comparison of InterDVS and IntraDVS in multi-task environments.

To observe the effect of the task set characteristics, we simulated the conventional InterDVS algorithms and the proposed IntraDVS algorithm with two different task sets. Two tasks sets, task set `A` and `B`, have 6 tasks but the characteristics are quite different. The task set `A` is homogeneous (i.e. the tasks in `A` have similar periods and WCETs) while the task set `B` is heterogeneous (i.e. the tasks in `B` have large variations in their periods and WCETs). Figure 4.14 shows the normalized energy consumption of several InterDVS algorithms such as `lppsEDF` [31], `ccEDF` [11], `laEDF` [11], and `DRA` [10] over IntraDVS. Except for `DRA`, IntraDVS algorithm outperforms most InterDVS algorithms tested.

It is also observed that InterDVS algorithms show poor energy performance when the worst-case processor utilization[7] (WCPU) is small. This is because following tasks cannot fully exploit all slack times generated by past

[7]The worst-case processor utilization means the total WCETs of all task instances divided by the hyper period.

and current tasks, when WCPU is too small. Some slack times are disappeared before following tasks are released. In this case, IntraDVS performs much better than InterDVS, because it utilizes all slack times.

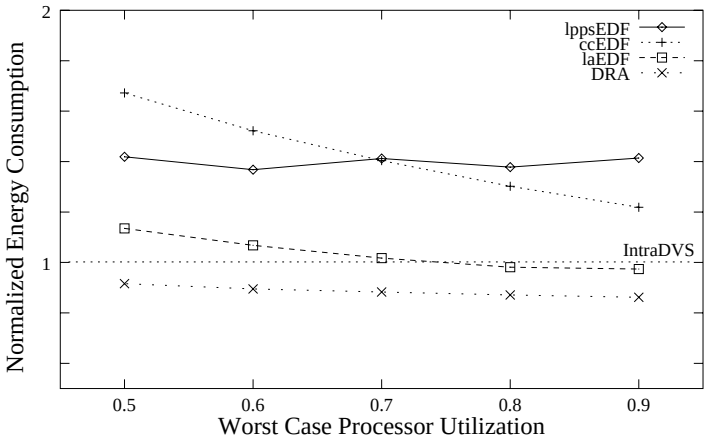
The performance of `DRA` algorithm is significantly different in two task sets. As shown in Figure 4.14(a), `DRA` outperforms both `laEDF` and IntraDVS when the task set `A` is used. However, when the task set `B` is used, `DRA` is inferior to IntraDVS, as shown in Figure 4.14(b). This is because the slack estimation method in `DRA` does not work well with non-uniform task utilizations. From this results, we can know that IntraDVS is better than InterDVS especially when the WCPU is small and the task set is heterogeneous.

## 4.6 IntraDVS for Soft Real-Time Tasks

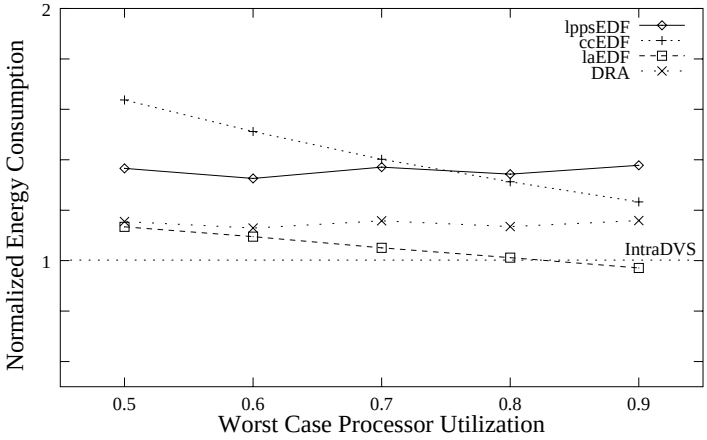
### 4.6.1 QoS-driven IntraDVS

Battery-powered mobile devices are becoming important platforms for processing multimedia data such as audio, video, and images. Such mobile systems need to support quality of service (QoS) requirements. Unlike hard real-time tasks, they require only statistical performance guarantees (e.g., meeting 95% of deadlines)[8] and are called soft real-time tasks. In this section, we present how we can use the IntraDVS algorithm for soft real-time tasks. If a task has 95% QoS constraint, deadline misses are permitted within 5% bound. For example, consider an MPEG video player, which

[8]In a general framework, we can consider that hard real-time applications require 100% QoS constraint.




(a) Task set A




(b) Task set B

Figure 4.14: Comparison of InterDVS and IntraDVS in different task sets.

should decode one frame within 30 ms, with 95% QoS constraint. The decoding times for 5% of all video frames might be over 30 ms. If the jobs (task instances) demanding more than $10^6$ cycles occupy only 5% among all jobs, it is best to give up to meet the deadlines for the jobs demanding more than $10^6$ cycles to minimize the energy consumption. This is because we can use a lower clock speed if we care only the jobs which demand no more than $10^6$ cycles. We call this techniques as the **QoS-driven IntraDVS**.

For example, consider a task whose WCEC and deadline is $W$ and $D$ respectively. The cumulative distribution function (CDF) of the task's cycle demands is

$$f(x) = \mathcal{P}[X \leq x] \tag{4.11}$$

where $X$ is the random variable of the task's demands. Let $\rho$ be the statistical performance requirement of the task. The task needs to meet $\rho$ percentage of deadlines. In other words, each job of the task should meet its deadline with a probability $\rho$. To support this requirement, the QoS-driven IntraDVS allocates $V$ cycles to each job of the task, so that the probability that each job requires no more than the allocated $V$ cycles is $\rho$, i.e.,

$$f(V) = \mathcal{P}[X \leq V] = \rho \tag{4.12}$$

It also uses the value $V$ as a virtual RWEC $\widehat{C}_{RWEC}$. The start speed $S_0$ of the task is determined with the virtual RWEC, i.e., $S_0 = V/D$. $\widehat{C}_{RWEC}$ is reduced at the speed of $S_0$. The VSEs generated by the QoS-driven IntraDVS accumulate the saved cycles at the VSEs, and examine whether the accumulated saved cycles is larger than $(W - V)$. Voltage scalings are performed only when the total saved cycles are larger than the value i.e., when

$C_{RWEC}(t)$ is smaller than $\widehat{C}_{RWEC}(t)$. This is to reflect the underestimated worst-case execution cycles. If a job execution requires $V$ cycles, the job completes exactly at its deadline when the voltage transition overhead is 0. However, the jobs demanding more than $V$ will miss deadlines.

Figure 4.15 shows the changes of $C_{RWEC}$ by the original IntraDVS and the QoS-driven IntraDVS of a task. The WCEC of the task is $200 \times 10^6$ cycles. The CDF of the task's cycle demands, $f$, satisfies a following equation:

$$f(150 \times 10^6) = \mathcal{P}[X \leq 150 \times 10^6] = 0.95 \tag{4.13}$$

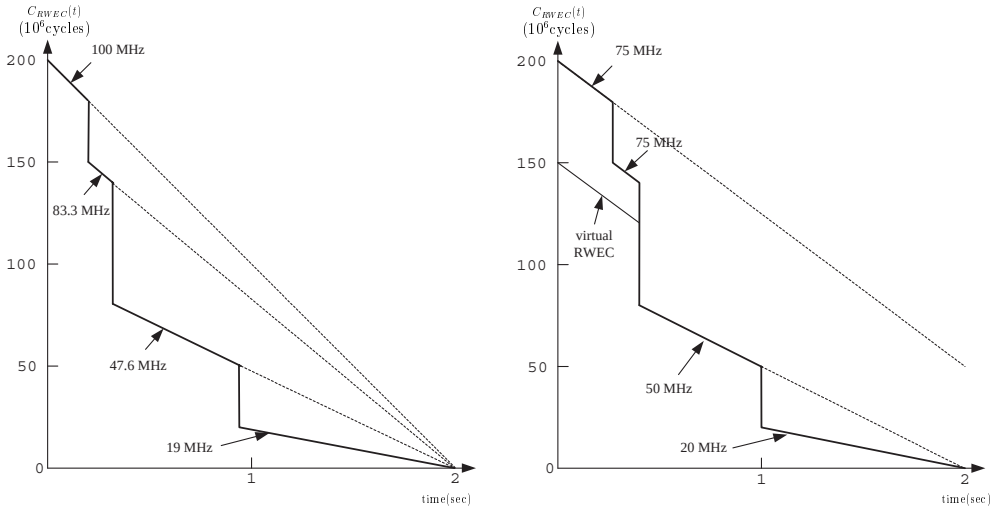
In the QoS-driven IntraDVS, if the QoS requirement is 95%, the program starts with the speed of 75 MHz because the virtual RWEC is $150 \times 10^6$ cycles. At the time 0.37 sec in Figure 4.15(b), the clock speed sustains 75 MHz although the remaining worst-case execution cycles are reduced. When the total saved cycles pass over $50 \times 10^6$ cycles at the time 0.4 sec, the clock speed is adjusted to 50 MHz. The new speed is calculated using the virtual RWEC as follows:

$$75 \text{ MHz} \times \frac{C_{RWEC}(t)}{\widehat{C}_{RWEC}(t)} = 75 \text{ MHz} \times \frac{80 \times 10^6 \text{ cycles}}{120 \times 10^6 \text{ cycles}}$$

After the first voltage transition, the speed update ratio is calculated by the same algorithm of the original IntraDVS.

### 4.6.2 Experiments

We evaluated the energy performance of the QoS-driven IntraDVS using a simulation. Figure 4.16(a) shows the energy consumptions under the QoS-driven IntraDVS. The energy consumptions are normalized by the results




(a) (b)

Figure 4.15: The changes of $C_{RWEC}(t)$ over different speed scaling algorithms: (a) Original IntraDVS and (b) QoS-driven IntraDVS.

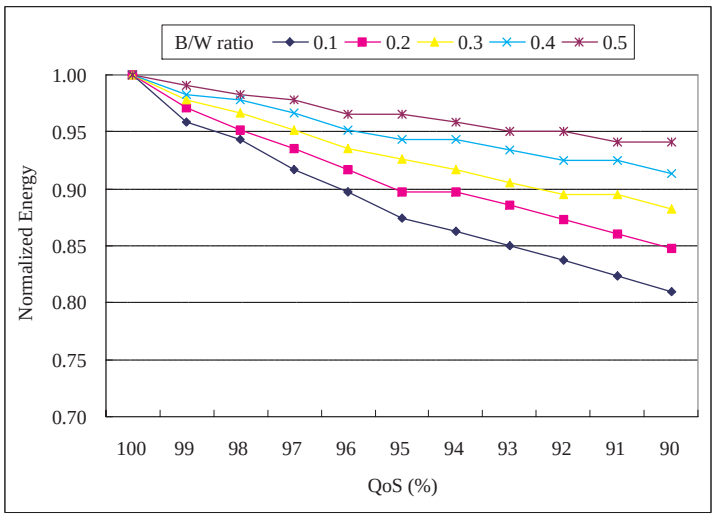
of IntraDVS under 100% QoS. Since the energy consumption is dependent on the QoS constraint, we experimented varying the QoS constraint from 100∼90%. We assume that the execution cycles of each instance of a program is drawn from a random Gaussian distribution with mean, denoted by $m$, and standard deviation, denoted by $\sigma$, given by

$$m = \frac{BCEC + WCEC}{2} \quad \text{and} \quad \sigma = \frac{WCEC - BCEC}{6}.$$

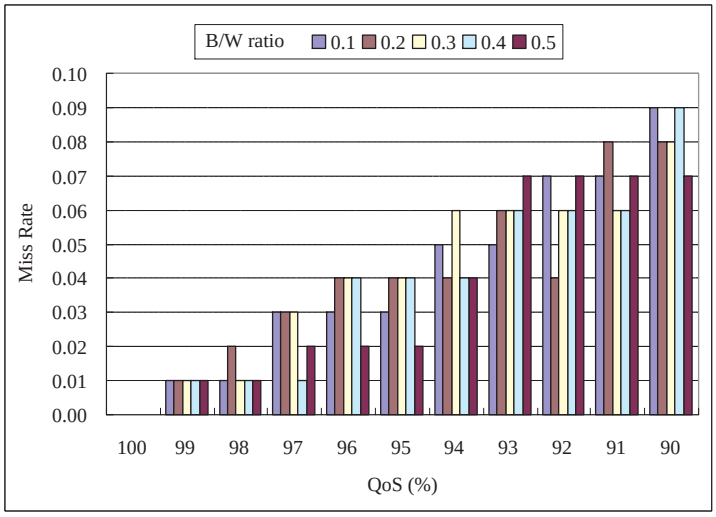
We varied the B/W ratio (=BCEC/WCEC) from 0.1 to 0.5.

When the QoS requirement is small and there is a big difference between BCEC and WCEC, QoS-driven IntraDVS uses a small virtual RWEC thus assigns a low start speed. So, the energy consumption is reduced significantly when the application has a weak QoS constraint and a small B/W ratio.

Figure 4.16(b) shows the deadline miss rate. We can know that if the QoS requirement is $\rho$ then the miss rate is always smaller than $1-\rho$. This result shows the QoS-driven IntraDVS satisfies the QoS constraints of applications.




(a)




(b)

Figure 4.16: The experimental results: (a) Energy Consumption and (b) Deadline Miss.

# Chapter 5

# Energy-Efficiency Improvement Techniques for IntraDVS

## 5.1 IntraDVS Using Profile Information

### 5.1.1 Motivation

Although the RWEP-based IntraDVS reduces the energy consumption significantly while guaranteeing the deadline, this is a pessimistic approach because it always predicts that the longest path will be executed. A more optimistic approach is to use the average-case execution path (ACEP) as a reference path. The ACEP is defined to be an execution path with the largest possibility to be executed. The ACEP can be decided by observing

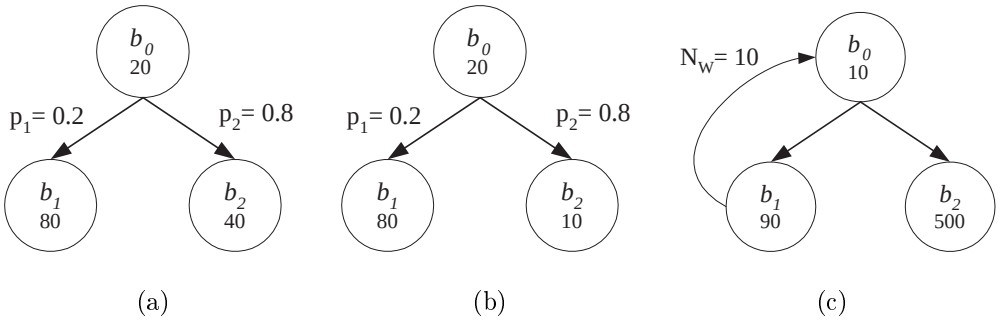



Figure 5.1: Example task graphs for RAEP-based IntraDVS.

the execution profile information.

It is easily understood that using ACEP instead of WCEP is more energy-efficient. For a typical program, about 80 percent of the program execution occurs in only 20 percent of its code, which is called the hot paths [42]. To achieve high energy efficiency, an IntraDVS algorithm should be optimized so that these hot paths are energy-efficient. If we use one of hot paths as a reference path, the speed change graph for the hot paths will be a near flat curve with little changes in the clock speed, which gives the best energy efficiency under a given amount of work [43]. In this case, even other paths (that are not the hot paths) become more energy-efficient because they can start with a lower clock speed than when the WCEP is used as a reference path.

For example, consider the task graph shown in Figure 5.1(a). Each edge has a probability to be taken at run time. In the RWEP-based IntraDVS, the execution path $(b_0, b_1)$ is taken as a reference path. However, if we use the RAEP-based IntraDVS, the reference path is $(b_0, b_2)$ because it has a higher probability. If we assume that the deadline of the task graph is 100

and the effective load capacitance $C_{eff}$ is 1, the average energy consumptions under two scheduling schemes are as follows:

$$\begin{aligned} E_{RWEP} &= p_1 \cdot (C_{EC}(b_0) \cdot S(b_0)^2 + C_{EC}(b_1) \cdot S(b_1)^2) \\ &\quad + p_2 \cdot (C_{EC}(b_0) \cdot S(b_0)^2 + C_{EC}(b_2) \cdot S(b_2)^2) \\ &= 0.2 \cdot (20 \cdot 1.0^2 + 80 \cdot 1.0^2) + 0.8 \cdot (20 \cdot 1.0^2 + 40 \cdot 0.5^2) = 44 \\ E_{RAEP} &= 0.2 \cdot (20 \cdot (\frac{3}{5})^2 + 80 \cdot (\frac{6}{5})^2) + 0.8 \cdot (20 \cdot (\frac{3}{5})^2 + 40 \cdot (\frac{3}{5})^2) = 41.76 \end{aligned}$$

This example shows that it is more energy-efficient to use the ACEP as a reference path.

However, consider the task graph shown in Figure 5.1(b). The average energy consumptions of the task graph are as follows:

$$\begin{aligned} E_{RWEP} &= 0.2 \cdot (20 \cdot 1.0^2 + 80 \cdot 1.0^2) + 0.8 \cdot (20 \cdot 1.0^2 + 10 \cdot 0.25^2) = 36.125 \\ E_{RAEP} &= 0.2 \cdot (20 \cdot (\frac{3}{10})^2 + 80 \cdot (\frac{12}{5})^2) + 0.8 \cdot (20 \cdot (\frac{3}{10})^2 + 10 \cdot (\frac{3}{10})^2) = 41.76 \end{aligned}$$

This example means that it is not always energy-efficient to use the ACEP as a reference path. This is because we consider only the probability of the execution path but the remaining execution cycles to determine the reference path.

Therefore, we need to modify the definition of ACEP. On determining a reference path, we have two choices in selecting one edge among two out edges of a branch node. The case which generates a lower energy consumption is optimal. For the task graphs in Figure 5.1, when we denote $C_{EC}(b_0), C_{EC}(b_1)$ and $C_{EC}(b_2)$ as $c_0, c_1$ and $c_2$ respectively, the average energy consumption, if the execution path $(b_0, b_1)$ is a reference path, is as follows:

$$E_1 = p_1 \cdot (c_0 + c_1) \cdot (\frac{c_0 + c_1}{D})^2 + p_2 \cdot (c_0 \cdot (\frac{c_0 + c_1}{D})^2 + c_2 \cdot (\frac{c_2(c_0 + c_1)}{c_1 D})^2)$$

But, the average energy consumption, when the execution path $(b_0, b_2)$ is a reference path, is as follows:

$$E_2 = p_1 \cdot (c_0 \cdot (\frac{c_0 + c_2}{D})^2 + c_1 \cdot (\frac{c_1(c_0 + c_2)}{c_2 D})^2) + p_2 \cdot (c_0 + c_2) \cdot (\frac{c_0 + c_2}{D})^2$$

Since the condition for $E_1 < E_2$ is

$$(\frac{c_0}{c_1} + 1)^2(p_1 c_1^3 + p_2 c_2^3 + c_0 c_1^2) < (\frac{c_0}{c_2} + 1)^2(p_1 c_1^3 + p_2 c_2^3 + c_0 c_2^2), \qquad (5.1)$$

we can select the execution path $(b_0, b_1)$ as a reference path if Equation (5.1) is true.

For the L-type VSE, we should predict the number of loop iterations. Consider the task graph in Figure 5.1(c), which has a loop with the maximum iteration number $N_w$. If we insert a voltage scaling code at the edge $(b_0, b_2)$, the average energy consumption under the RWEP-based IntraDVS scheme is as follows when we denote $C_{EC}(b_0), C_{EC}(b_1)$ and $C_{EC}(b_2)$ as $c_0, c_1$ and $c_2$ respectively:

$$\begin{aligned} E_{RWEP} &= \sum_{0}^{N_w} p_i \cdot [((i+1) \cdot c_0 + i \cdot c_1) \cdot S_0^2 + c_2 \cdot S_{1,i}^2] \\ S_0 &= \frac{N_w \cdot (c_0 + c_1) + c_0 + c_2}{D} \\ S_{1,i} &= S_0 \cdot \frac{c_2}{(N_w - i) \cdot (c_0 + c_1) + c_2} \end{aligned}$$

where $i$ and $p_i$ are the number of loop iterations and the probability of the $i$-number of loop iterations, respectively.

In the RAEP-based IntraDVS, we can use the average number of loop iterations to calculate the remaining execution cycles. If the actual number of loop iterations is larger than the average number of loop iterations, we should increase the clock speed. If we adjust the clock speed at only the

exit point of the loop $(b_0, b_2)$, the average energy consumption under the RAEP-based IntraDVS scheme is as follows:

$$\begin{aligned} E_{RAEP} &= \sum_{i=0}^{N_w} p_i \cdot [((i+1) \cdot c_0 + i \cdot c_1) \cdot S_0^2 + c_2 \cdot S_{1,i}^2] \\ S_0 &= \frac{N_{avg} \cdot (c_0 + c_1) + c_0 + c_2}{D} \\ S_{1,i} &= S_0 \cdot \frac{Max(0, i - N_{avg}) \cdot (c_0 + c_1) + c_2}{Max(0, N_{avg} - i) \cdot (c_0 + c_1) + c_2} \end{aligned} \tag{5.2}$$

where $N_{avg}$ means the average number of loop iterations.

To adjust the clock speed when the loop continues its execution even though it has completed the average number of iterations, we should insert the voltage scaling codes at both $(b_0, b_1)$ and $(b_0, b_2)$. At the VSE $(b_0, b_1)$, if the actual iteration number is larger than the average number, the clock speed is increased by the factor of $(c_0 + c_1 + c_2)/c_2$ because we can know at the edge that one loop iteration at least is required additionally. For this technique, the average energy consumption is as follows:

$$\begin{aligned} E_{RAEP} &= \sum_{i=0}^{N_{avg}} p_i \cdot [((i+1) \cdot c_0 + i \cdot c_1) \cdot S_0^2 + c_2 \cdot S_{1,i}^2] \\ &\quad + \sum_{i=N_{avg}+1}^{N_w} p_i \cdot [((N_{avg}+1) \cdot c_0 + N_{avg} \cdot c_1) \cdot S_0^2 \\ &\quad + \sum_{j=1}^{i-N_{avg}} (c_0 + c_1) \cdot S_{j,i}^2 + c_2 \cdot S_{i-N_{avg},i}^2] \\ S_0 &= \frac{N_{avg} \cdot (c_0 + c_1) + c_0 + c_2}{D} \\ S_{1,i} &= S_0 \cdot \frac{c_0 + c_1 + c_2}{c_2}, \quad S_{j,i} = S_{j-1,i} \cdot \frac{c_0 + c_1 + c_2}{c_2} \end{aligned} \tag{5.3}$$

Therefore, we should use the value of $N_{avg}$ which minimizes Equation

(5.2) or (5.3) (depending on the used technique). But, for a simple RAEP-based IntraDVS model, we define the ACEP as follows:

$$S : \textbf{if}(E) \ \ \textbf{then} \ \ S_1; \ \ \textbf{else} \ \ S_2.$$

$$ACEC(S) = ACEC(E) + \begin{cases} ACEC(S_1) & \text{if } \ ACEC(S_1) \cdot prob(S_1) \geq ACEC(S_2) \cdot prob(S_2) \\ ACEC(S_2) & \text{otherwise} \end{cases}$$

$$S : \textbf{while}(E) \ \ S_1.$$

$$ACEC(S) = (ACEC(E) + ACEC(S_1)) \cdot N_{avg} + ACEC(E)$$

where $prob(S_1)$ and $prob(S_2)$ are the execution probabilities of the statements $S_1$ and $S_2$ respectively.

Figure 5.2 shows an RAEP-based CFG $G_P^{RAEP}$ with $C_{RAEC}(b_i)$ values that represent the remaining average-case execution cycles (RAECs) among all the paths that start from $b_i$. The bold edges in $G_P^{RAEP}$ means that it composes the average-case execution path between either branching edges.

In Figure 5.2, the initial reference path is ($b_1$, $b_{call1}$, $b_8$, $b_9$, $b_{11}$,$b_{if}$, $b_{call4}$, $b_8$, $b_9$, $b_{11}$, $b_7$). With the reference path, $C_{RAEC}(b_i)$ is computed. For example, $C_{RAEC}(b_{if}) = C_{EC}(b_{if}) + C_{RAEC}(b_{call4})$. At the RAEP-based IntraDVS, there are Up-VSEs (marked by $\circ$ in Figure 5.2) as well as Down-VSEs (marked by $\bullet$ in Figure 5.2). For the Up-VSEs, we should be careful. Though we can exclude a Down-VSE for voltage scaling points if the saved cycles are small at the Down-VSE, no Up-VSE can be excluded. If we do not increase the clock speed and supply voltage at an Up-VSE, there is a deadline miss. Therefore, the voltage scaling codes should be inserted at all Up-VSEs and there are voltage transition overheads at all Up-VSEs. To prevent this problem, we should select the worst-case execution path as a

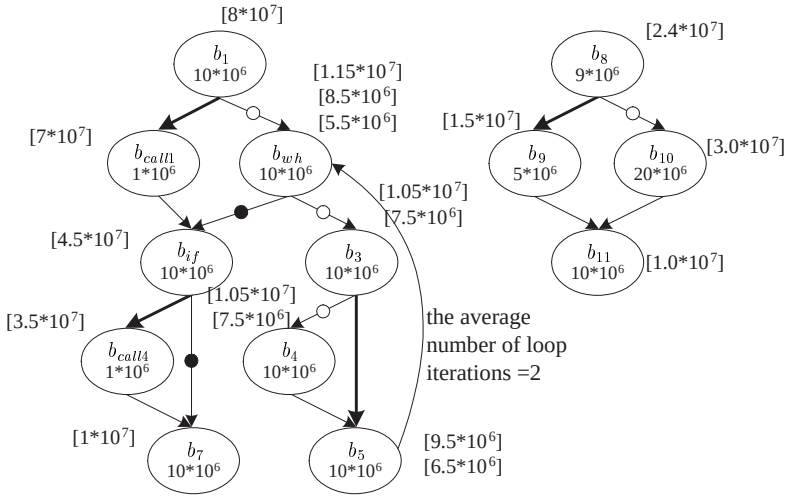



Figure 5.2: A RAEP-based CFG $G_P^{RAEP}$.

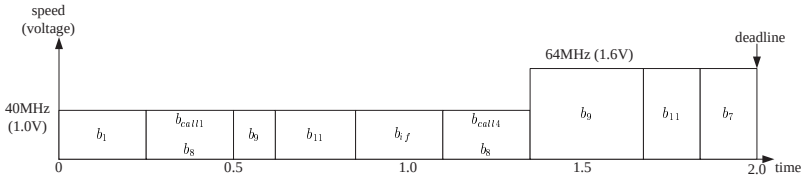



Figure 5.3: Speed and voltage changes by the RAEP-based IntraDVS.

reference path even though the execution path has a low probability if the difference between RWEC and RAEC is small.

Figure 5.3 shows how the speed and voltage change in the RAEP-based scheduling. The speed changed from 40 MHz to 64 MHz at the edge $(b_8, b_{10})$ because this is an Up-VSE with the SUR value of 1.6 $(= \frac{40\times10^6}{25\times10^6})$. Compared to the RWEP-based IntraDVS algorithm, the RAEP-based IntraDVS algorithm can achieve more energy reduction if the execution path follows the reference path.

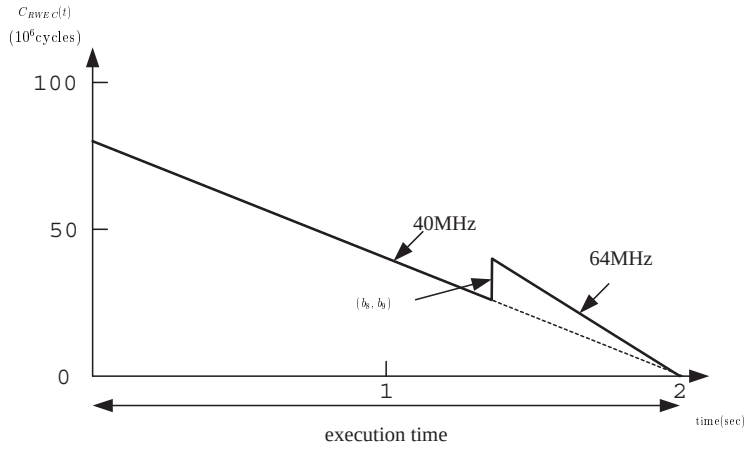



Figure 5.4: The changes of $C_{RAEC}(t)$ over RAEP-based IntraDVS.

### 5.1.2 Guaranteeing Safeness

Although the RAEP-based scheduling is more energy-effective than the RWEP-based scheduling, the pure RAEP-based approach cannot meet the timing requirements of hard real-time applications. This is because it does not satisfy the timing constraints for all the execution paths if ACEP is used as reference path. For example, consider the case when the WCEP and ACEP take significantly different number of execution cycles. When the execution takes the WCEP at the middle of program execution, it is possible that the program fails to meet its deadline even if the processor runs at its maximum speed during the remaining paths. Therefore, we need a safe approach which can guarantee the timing constraint.

To overcome the deadline miss problem of the pure RAEP-based IntraDVS algorithm, we use the remaining *safe* execution cycles (RSEC) as well as the RAEC. When we determine the clock speed based on the RSEC,

the safeness is guaranteed. To know the RSEC of a basic block, we should first calculate the lower bound of a basic block's clock speed, $S_{LB}(b_i)$.

$$S_{LB}(b_i) = \frac{C_{EC}(b_i)}{d(b_i) - \sigma(b_i)}$$

where $d(b_i)$ and $\sigma(b_i)$ are the deadline and the start time of $b_i$ respectively. The deadline of a basic block $b_i$, $d(b_i)$, is defined as follows:

$$d(b_i) = D - \frac{C_{RWEC}(b_i) - C_{EC}(b_i)}{f_{max}}$$

where $D$ is the deadline of the overall program and $f_{max}$ is the maximum clock speed. The deadline of $b_i$ is estimated assuming that all basic blocks after the basic block $b_i$ are executed with the full speed $f_{max}$.

The clock speed of $b_i$, $S(b_i)$, should be larger than $S_{LB}(b_i)$. Then, the remaining safe execution cycles can be represented as follows:

$$S(b_i) = \frac{C_{RSEC}(b_i)}{D - \sigma(b_i)} \geq S_{LB}(b_i) = \frac{C_{EC}(b_i)}{d(b_i) - \sigma(b_i)}$$

$$C_{RSEC}(b_i) \geq \frac{D - \sigma(b_i)}{d(b_i) - \sigma(b_i)} C_{EC}(b_i) \tag{5.4}$$

The right part of Equation (5.4) has the maximum value when $\sigma(b_i)$ is the largest value. The largest value of $\sigma(b_i)$ is the latest start time of a basic block $b_i$, $lst(b_i)$, which is defined as follows:

$$lst(b_i) = D - \frac{C_{RAEC}(b_i)}{S_{max}(b_i)}$$

where $S_{max}(b_i)$ is the maximum clock speed which $b_i$ can have.

Consequently, the RSEC of $b_i$ is

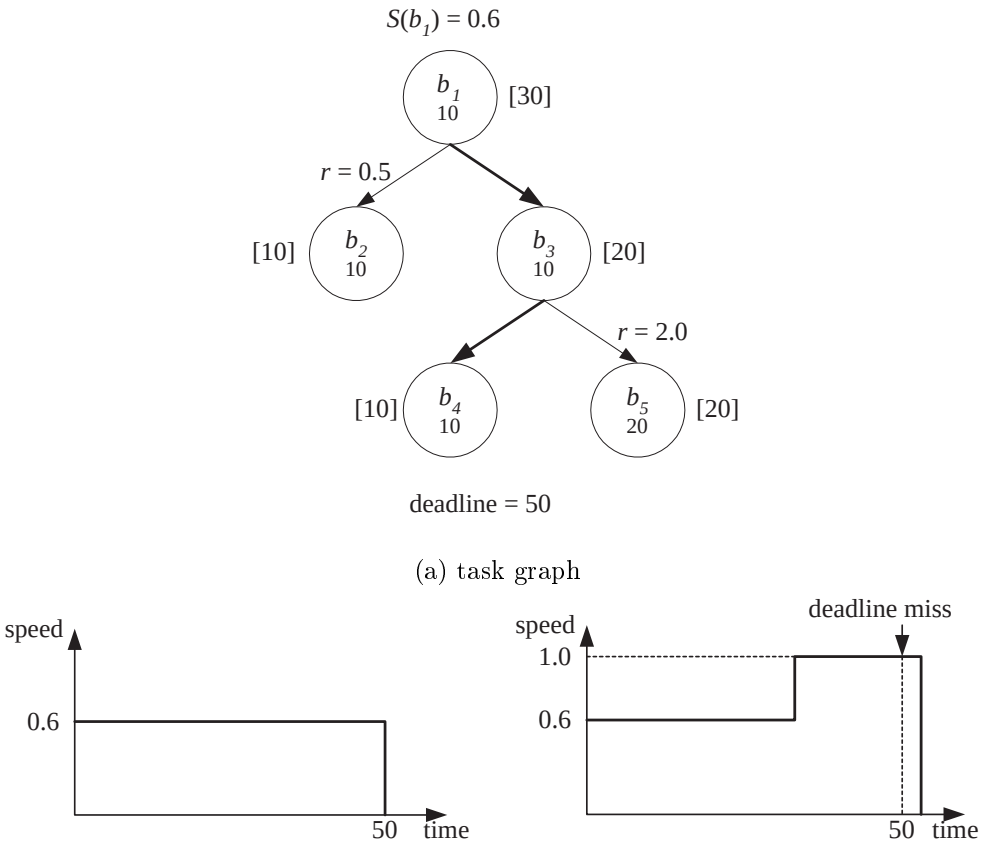
$$C_{RSEC}(b_i) = \frac{D - lst(b_i)}{d(b_i) - lst(b_i)} C_{EC}(b_i) \tag{5.5}$$

A basic block $b_i$ should have both $C_{RAEC}(b_i)$ and $C_{RSEC}(b_i)$. If $C_{RSEC}(b_i)$ is larger than $C_{RAEC}(b_i)$, the clock speed should be determined based on $C_{RSEC}(b_i)$. Consequently, the clock speed is adjusted as follows at an edge $(b_i, b_j)$:

$$S(b_j) = S(b_i)\frac{Max(C_{RAEC}(b_j), C_{RSEC}(b_j))}{Max(C_{RAEC}(b_i), C_{RSEC}(b_i)) - C_{EC}(b_i)}$$

Figure 5.5(a) shows a control flow graph with a deadline 50 and the remaining average-case execution cycles of basic blocks. The ACEP, $(b_1, b_3, b_4)$, is used as the reference path. The bold edges indicate the average-case execution path. As we can see at Figure 5.5(c), the deadline miss can be occurred. Figure 5.6(a) shows the remaining safe execution cycles (RSEC) as well as the remaining average-case execution cycles (RAEC) of basic blocks. For example, $C_{RSEC}(b_3)$ is $25.04(= \frac{50-16.7}{30-16.7} \cdot 10)$ by Equation (5.5). A basic block $b_i$ uses the maximum between $C_{RSEC}(b_i)$ and $C_{RAEC}(b_i)$. So, the remaining execution cycles of the basic block $b_3$ is 25.04. At the edge $(b_1, b_3)$, the speed update ratio is $1.25$ $(= C_{RSEC}(b_3)/(C_{RAEC}(b_1) - C_{EC}(b_1)) = \frac{25.04}{30-10})$.

Using this safe RAEP-based IntraDVS, we can get an energy efficient speed schedule satisfying the deadline constraint. But, there is a more energy-efficient speed schedule under the deadline constraint. Though the speeds of basic blocks $b_1$ and $b_3$ are different at Figures 5.6(b) and (c), it is more energy-efficient to use the same speed for both $b_1$ and $b_3$. In other words, it is better for the basic blocks on a reference path to have the same clock speed because there is a large probability to be taken the path at run time. If the basic blocks $b_i, \cdots, b_j$ compose a reference path, we estimated the deadline and the latest start time of the group of basic

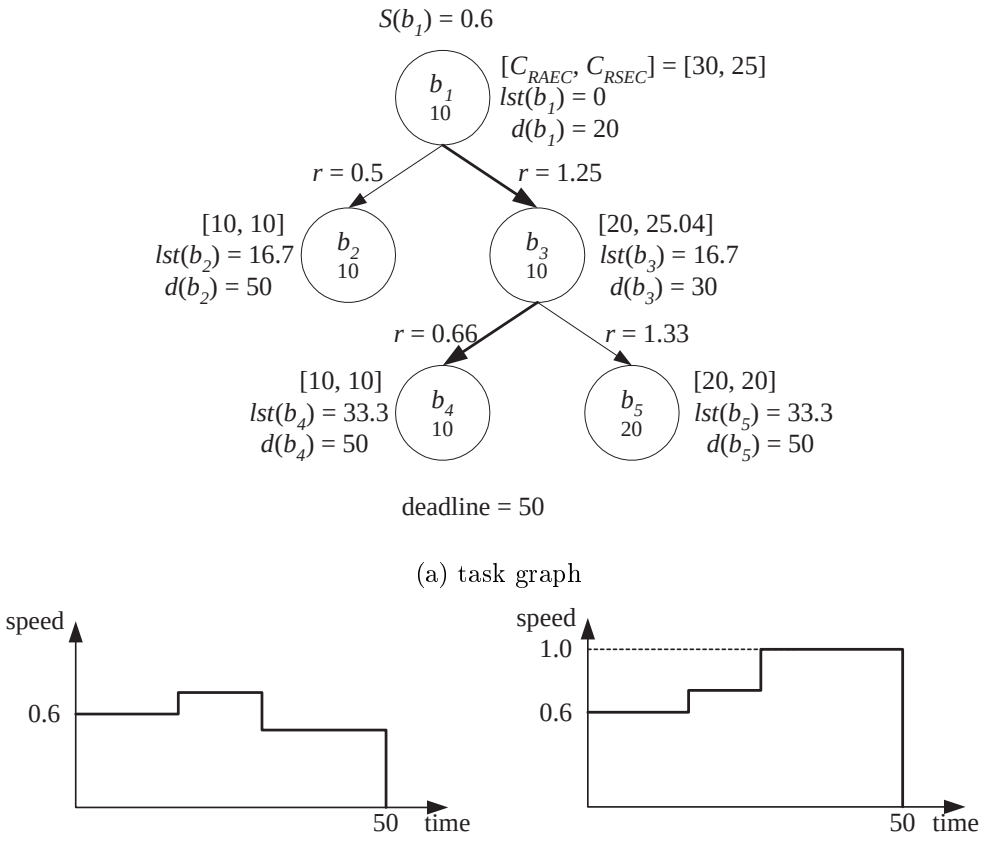



(a) task graph

(b) speed change at $(b_1, b_3, b_4)$

(c) speed change at $(b_1, b_3, b_5)$

Figure 5.5: Pure RAEP-based IntraDVS.




(a) task graph

(b) speed change at $(b_1, b_3, b_4)$

(c) speed change at $(b_1, b_3, b_5)$

Figure 5.6: Safe RAEP-based IntraDVS.

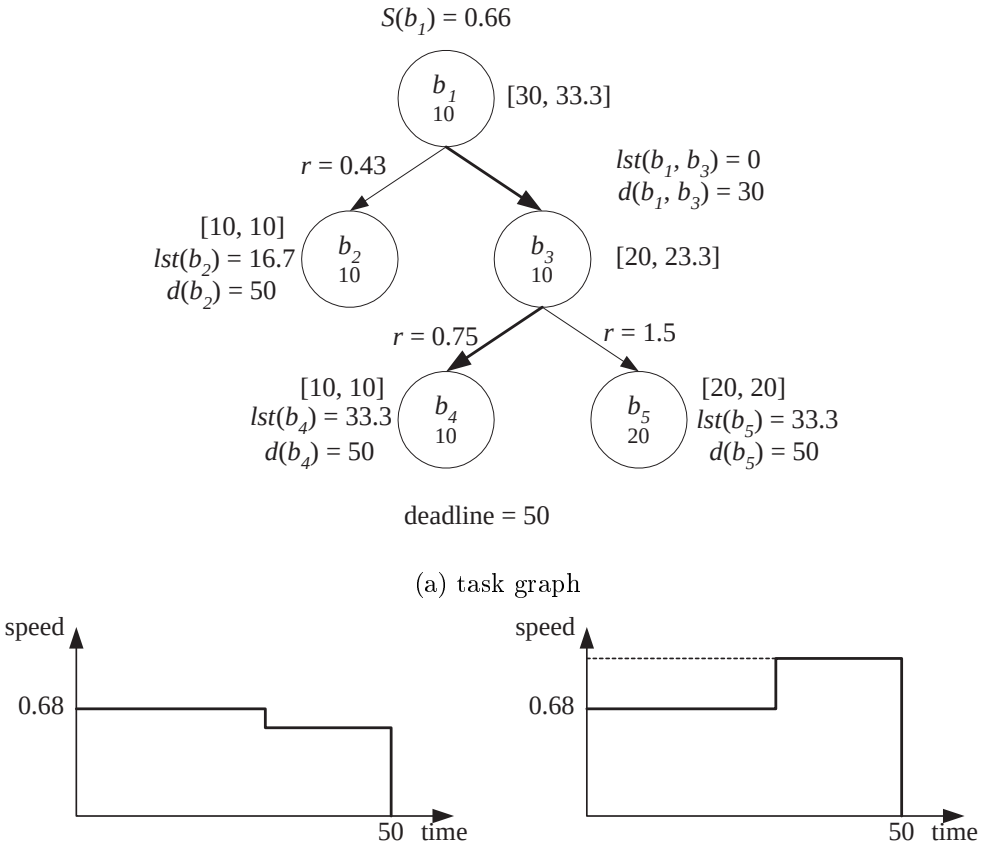
blocks. The deadline and the latest start time of basic blocks $b_i, \cdots, b_j$, $d(b_i, \cdots, b_j)$ and $lst(b_i, \cdots, b_j)$, are same to $d(b_j)$ and $lst(b_i)$, respectively. For example, in Figure 5.7(a), the basic block $(b_1, b_3)$ have the deadline $d(b_1, b_3) = d(b_3) = 30$ and the latest start time $lst(b_1, b_3) = lst(b_1) = 0$. Using these values, we estimate $C_{RSEC}(b_i), \cdots, C_{RSEC}(b_j)$ as follows:

$$C_{RSEC}(b_k) = \begin{cases} \frac{D - lst(b_i, \cdots, b_j)}{d(b_i, \cdots, b_j) - lst(b_i, \cdots, b_j)} \cdot (C_{EC}(b_i) + \cdots + C_{EC}(b_j)) & \text{if } k = i \\ C_{RSEC}(b_p) - C_{EC}(b_p) & \text{otherwise.} \end{cases}$$

where $b_p$ is the predecessor basic block of $b_k$. As we can see at Figures 5.7(b) and (c), the basic blocks $b_1$ and $b_3$ has a flat speed schedule, thus the schedules consume less energy than the schedules in Figures 5.6(b) and (c). Moreover, the task graph in Figure 5.7(a) has less voltage scaling edges than the task graph in Figure 5.6(a). This technique is called the *profile-aware* safe RAEP-based IntraDVS to separate from the original safe RAEP-based IntraDVS. For a brevity, we use the terminology of the safe RAEP-based IntraDVS to denote the profile-aware safe RAEP-based IntraDVS.

### 5.1.3 Comparisons of RWEP-based IntraDVS and RAEP-based IntraDVS algorithms

The RAEP-based IntraDVS outperforms the RWEP-based IntraDVS in energy efficiency because the scheduled speed does not fluctuate much. However, the RAEP-based IntraDVS algorithm needs the reference path modification to guarantee the timing constraint as shown in the previous subsection, which is very complicated and time-consuming. It also inevitably needs the profiling information, while the RWEP-based IntraDVS only requires the WCET analysis.




(a) task graph

(b) speed change at $(b_1, b_3, b_4)$

(c) speed change at $(b_1, b_3, b_5)$

Figure 5.7: Profile-aware Safe RAEP-based IntraDVS.

Another problem of the RAEP-based IntraDVS is that the time slot between the release time and the deadline of a task should be determined statically. This does not matter in the single task environments. However, it is a serious problem in the multi-task environments, since the time slot for a task is changed depending on the release time. Unfortunately, it prohibits from processing reference path modification at compile time. One solution for this problem is to prepare multiple configurations at each VSEs for different values of task's time slots. For example, we prepare the configurations when the assigned time slots are 1, 2, and 3 times of WCET. At run time, if the assigned time slots are 2.5 times of WCET, we use the second configuration.

### 5.1.4 Experiments

To compare the power reduction performance of the RAEP-based IntraDVS algorithm with the RWEP-based IntraDVS algorithm, we have experimented with an MPEG-4 video encoder. In the RAEP-based IntraDVS, the probability of branch edges and the average number of loop iterations in a CFG of the MPEG-4 video encoder are estimated using the profiled information. A probability of 0.5 is assigned to the branch edges for which we cannot collect the execution profiles with sample test bitstreams.

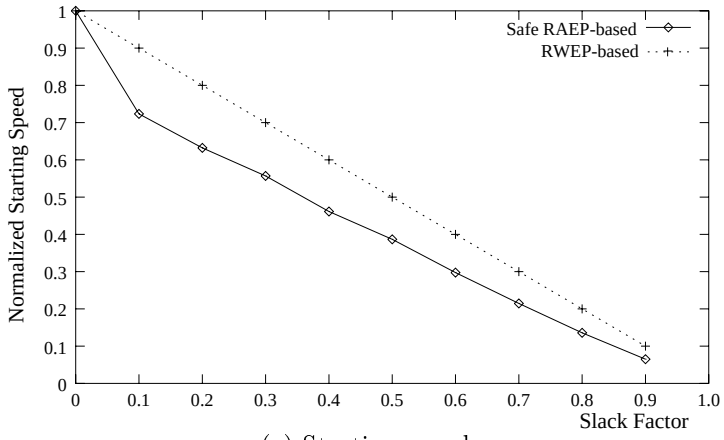
Figure 5.8(a) shows how the normalized starting speed changes over various slack factor values. The slack factor, defined by $\frac{deadline-WCET}{deadline}$, represents the fraction of time that a processor becomes idle after WCET. The execution times of safe ACEPs (by the procedure described in Section 5.1) for the MPEG-4 encoder is up to 35% smaller than the WCET. This

means that the processor can start initially 35% more slowly than the speed required by the RWEP-based IntraDVS algorithm.

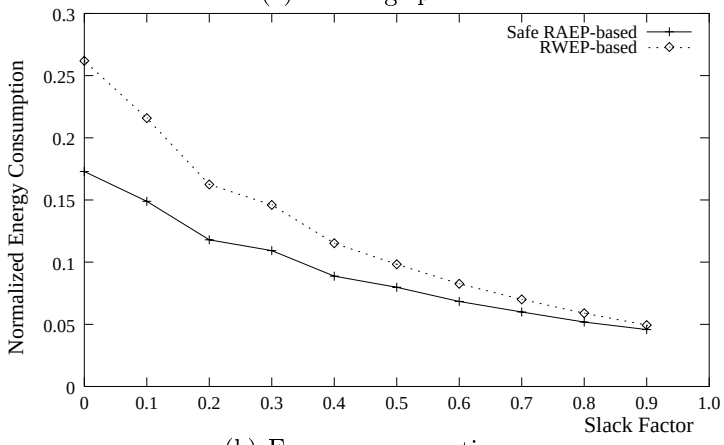
Figure 5.8(b) compares the energy consumption of two IntraDVS scheduling algorithms, varying the slack factor. All the results were normalized over the energy consumption of the original program running on a DVS-unaware system. For the MPEG-4 encoder, the safe RAEP-based IntraDVS algorithm reduces the energy consumption up to 34% over the RWEP-based IntraDVS algorithm.

Note that there is a large gap between energy consumption of RWEP-based and RAEP-based IntraDVS algorithms, even when the slack factor is 0 (i.e. deadline = WCET). This is because, although the starting speed is set to the same speed as in the RWEP-based IntraDVS, there are many execution paths that still can take advantage of the RAEP-based speed settings. That is, in order to meet the timing constraint, virtual blocks are added so that the initial speed is set to the same speed as in the RWEP-based IntraDVS algorithm. However, the (partial) paths following the virtual blocks can take advantage of the RAEP-based speed settings. As the slack factor increases, the energy consumption gap decreases because supply voltages of both IntraDVS algorithms get lower. Since the energy consumption is proportional to ${V_{dd}}^2$, the lower voltage values result in a smaller difference in the energy consumption.

We also experimented with artificial workloads which are drawn from a random Gaussian distribution. Figure 5.9 shows the normalized energy consumptions under the RWEP-based IntraDVS, the RAEP-based IntraDVS and the optimal DVS. The energy gain of the RAEP-based IntraDVS over




(a) Starting speed




(b) Energy consumption

Figure 5.8: Normalized starting speed and energy consumption of the RWEP-based IntraDVS and the RAEP-based IntraDVS versus the slack factor.

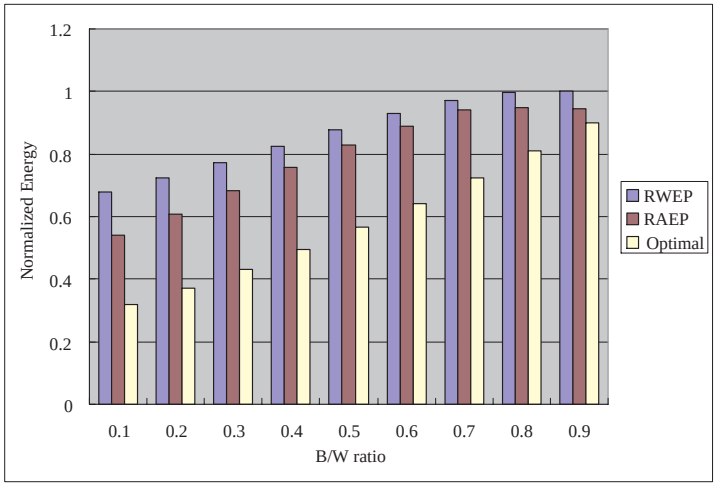


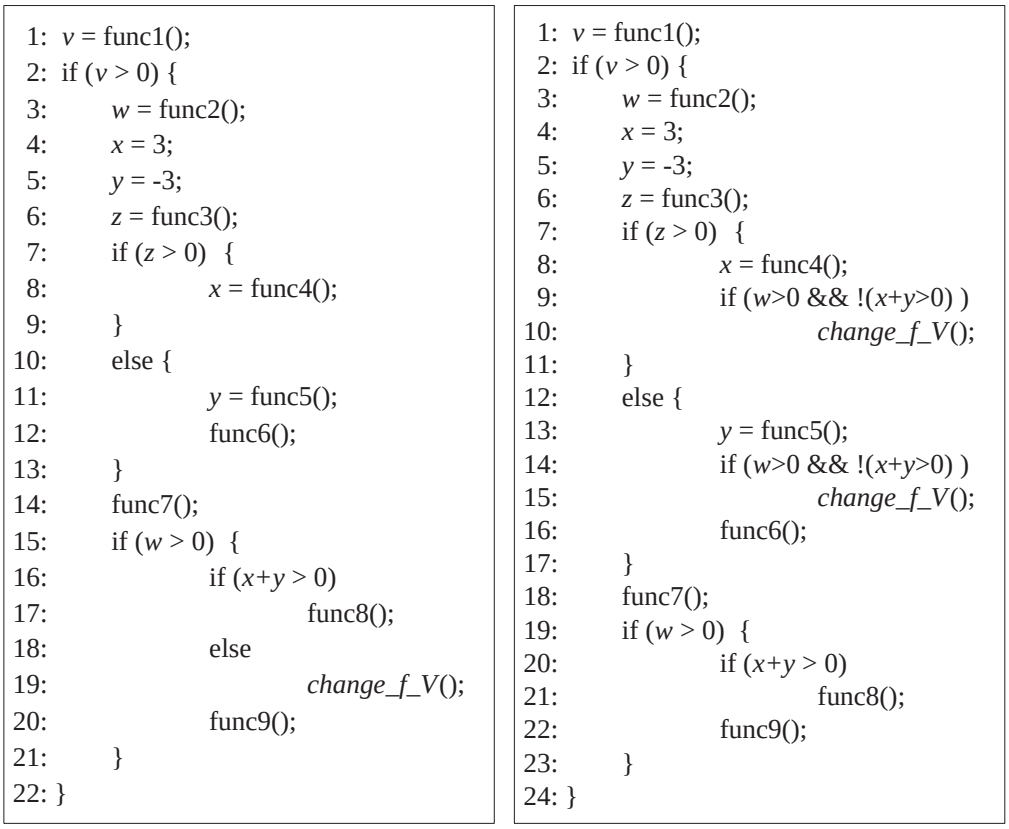

Figure 5.9: Experimental results of the RWEP-based IntraDVS, the RAEP-based IntraDVS and the optimal DVS.

the RWEP-based IntraDVS increases as the B/W ratio (=BCET/WCET) decreases. When the B/W ratio is 0.1, the RAEP-based IntraDVS reduces the energy consumption by 21% over the RWEP-based IntraDVS.

## 5.2 IntraDVS Using Data Flow Analysis

### 5.2.1 Motivation

The original IntraDVS techniques select the voltage scaling points (VSPs) using the control flow information (i.e., branch and loop) of a target program. For example, in Figure 5.10(a), the IntraDVS algorithm inserts the voltage scaling code, $change\_f\_V()$, at the line 19. At the line 19, we can know that the remaining worst-case execution cycles are reduced because the function func8 is not executed. However, we can decide the direction




(a) Original IntraDVS

(b) Look-ahead IntraDVS

Figure 5.10: An example program for look-ahead IntraDVS.

of the branch at the line 16 earlier because the values of $x$ and $y$ are not changed after the line 8 or the line 11. Figure 5.10(b) shows the modified program which adjusts the clock speed and the supply voltage at the line 10 or the line 15. The program in Figure 5.10(b) consumes less energy than the one in Figure 5.10(a) because the functions func6 and func7 is executed with a lower speed if $w > 0$ and $x + y \leq 0$.

This example shows that we can improve the energy performance of IntraDVS further if we can move voltage scaling points to the earlier instruc-

tions. To change the voltage scaling points, we should identify the data dependency using a data flow analysis technique. The data flow analysis provides the information about how a program manipulates its data [44]. Using data flow analysis, we can decide program locations where each variable is defined and used. We call the proposed IntraDVS technique based on data flow information as the **look-ahead IntraDVS** (LaIntraDVS) technique.

### 5.2.2 Single-Step Look-ahead IntraDVS

For LaIntraDVS, we need several post-processing steps after the voltage scaling points are selected by the original IntraDVS algorithm. To explain the post-processing steps, we define following terms and notations.

**Definition 1** *An instruction* `I` *is called a* **definition** $d_x$ *of a variable* $x$ *if the instruction* `I` *assigns, or may assign, a value to* $x$.

**Definition 2** *Given a program location* `t`*, a definition* $d_x$ *of a variable* $x$ *is called a* **data predecessor** $P_x^t$ *of the variable* $x$ *at* `t` *if there exists a path from* $d_x$ *to* `t` *such that the value of* $x$ *is not changed along the path. A data predecessor set* $\mathbb{P}(t, x)$ *of the variable* $x$ *at* `t` *is a set of all data predecessors of the variable* $x$ *at* `t`.

**Definition 3** *Given a program location* `t` *and a variable* $x$*, a program location* `p` *is called a* **look-ahead point** $L_x^t$ *of the variable* $x$ *at* `t` *if the following two conditions are satisfied:*

- *There exists one or more paths from* $p$ *to* $t$ *but there is no path from* $p$ *to* $t$ *such that the value of* $x$ *is changed along the path.*
- *There is no other program location* $p\prime$ *between* $P_x^t$ *and* $p$*, which satisfies the first condition.*

*A look-ahead point set* $\mathbb{L}(t, x)$ *is a set of all look-ahead points of the variable* $x$ *at* $t$*.*

**Definition 4** *Given a voltage scaling point* $s$*, a variable* $v$ *is a* **condition variable** *of* $s$ *if the value of the variable* $v$ *determines whether* $s$ *is executed or not at run time.*

**Definition 5** *Given a voltage scaling point* $s$ *and the set of condition variables* $V(s) = \{v_1, \cdots, v_n\}$ *of* $s$*, a look-ahead point* $p \in \mathbb{L}(s, v_1) \cup \cdots \cup \mathbb{L}(s, v_n)$ *is a* **look-ahead voltage scaling point** *(LaVSP) of* $s$ *if there is no other look-ahead point* $p\prime \in \mathbb{L}(s, v_1) \cup \cdots \cup \mathbb{L}(s, v_n)$ *along the path from* $p$ *to* $s$*. The set of all look-ahead voltage scaling points is denoted by* $LaVSP(s)$*.*

Given an original voltage scaling point $s$, we first identify the branch condition $C(s)$ which is the necessary condition for $s$ to be executed at run time. Second, using the variables in the expression of $C(s)$, we compose a set of condition variables $V(s)$. Third, the data predecessor set $\mathbb{P}(s, v_i)$ and the look-ahead point set $\mathbb{L}(s, v_i)$ are identified for each variable $v_i$ in $V(s)$ using a data flow analysis technique. Fourth, we identify the look-ahead voltage scaling points $LaVSP(s)$. Lastly, we insert the voltage scaling codes at the look-ahead voltage scaling points.

For example, in Figure 5.10(a), the branch condition for the voltage scaling point at line 19 is $C(s) = (v > 0) \wedge (w > 0) \wedge \neg(x + y > 0)$. The variables in $C(s)$ are $v, w, x$, and $y$ (i.e., $V(s) = \{v, w, x, y\}$). If we represent a program point with its line number, $\mathbb{P}(s, v) = \{1\}$, $\mathbb{L}(s, v) = \{2\}$, $\mathbb{P}(s, w) = \{3\}$, $\mathbb{L}(s, w) = \{4\}$, $\mathbb{P}(s, x) = \{4, 8\}$, $\mathbb{L}(s, x) = \{9, 11\}$, $\mathbb{P}(s, y) = \{5, 11\}$, and $\mathbb{L}(s, y) = \{8, 12\}$. From this information, we can know that $LaVSP(s) = \{9, 12\}$. Figure 5.10(b) shows the modified program with LaVSPs. At the lines 9 and 14, control expressions are inserted to reflect the condition $C(s) = (v > 0) \wedge (w > 0) \wedge \neg(x + y > 0)$. Since the condition $(v > 0)$ is always true at the lines 9 and 14, it is unnecessary to insert a control expression for the condition.

With the LaVSPs, the next step is to determine the speed update ratio. For example, if a original VSP $(b_i, b_j)$ has the LaVSP $p$, the speed update ratio at $p$ is

$$r(p) = \frac{C_{RWEC}(p) - (C_{RWEC}(b_i) - C_{EC}(b_i) - C_{RWEC}(b_j))}{C_{RWEC}(p)}$$

because the reduced cycles at the VSP $(b_i, b_j)$ is $C_{RWEC}(b_i) - C_{EC}(b_i) - C_{RWEC}(b_j)$.

In Figure 5.10(a), if the clock speed is $f_{15}$ at the line 15, the clock speed at the line 19, $f_{19}$, will be $f_{19} = f_{15} \times \frac{C_{func9}}{C_{func8} + C_{func9}}$ (when we consider only the execution cycles for functions), where $C_{func8}$ and $C_{func9}$ are the worst-case execution cycles for the functions func8 and func9 respectively. However, in Figure 5.10(b), the clock speed at the line 10 and the line 15 are

$$f_{10} = f_9 \times \frac{C_{func7} + C_{func9}}{C_{func7} + C_{func8} + C_{func9}}$$

and

$$f_{15} = f_{14} \times \frac{C_{func6} + C_{func7} + C_{func9}}{C_{func6} + C_{func7} + C_{func8} + C_{func9}}$$

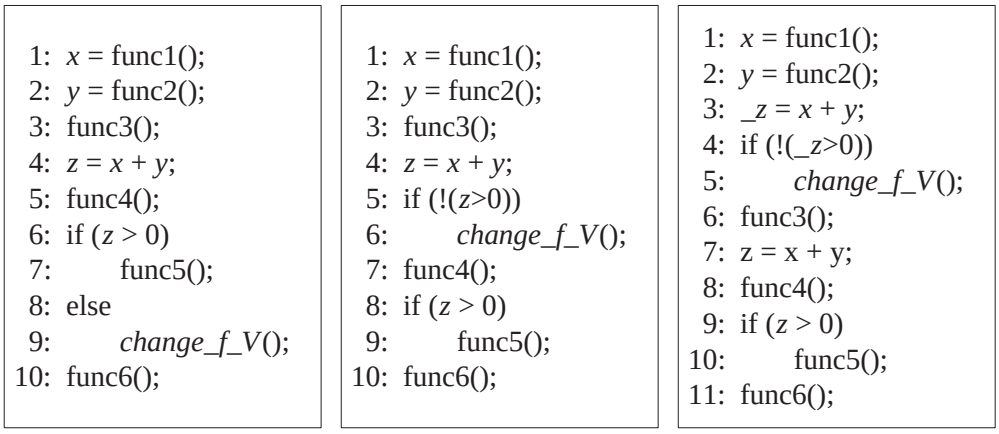
respectively.

### 5.2.3 Multi-Step Look-ahead IntraDVS

Although the look-ahead approach in LaIntraDVS can improve the energy performance of the IntraDVS technique, there are many cases where the cycle distance between the original VSP and the newly identified LaVSP is relatively short, achieving a small energy gain only[1]. This is the limitation of the single-step LaIntraDVS approach, where an look-ahead point is directly used as a voltage scaling point. To solve this problem, we propose the multi-step look-ahead IntraDVS technique, where the look-ahead point is recursively processed to find earlier scaling points.

Figure 5.11 shows an example of the multi-step look-ahead IntraDVS algorithm. For the program generated by the original IntraDVS algorithm (shown in Figure 5.11(a)), the single-step LaIntraDVS algorithm moves the scaling location to the line 6 as shown in Figure 5.11(b). Since the variable $z$ is defined at the line 4, LaIntraDVS inserted the voltage scaling code at the lines 5 and 6. However, the variable $z$ is the sum of $x$ and $y$, and the values of both $x$ and $y$ are known before the function func3. If the number of execution cycles for func3 is large and the addition operation requires small execution cycles, it is better to insert the addition code and the voltage scaling code just after the line 2. Figure 5.11(c) shows the

[1]Since a variable is generally defined just before the variable is used, the look-ahead IntraDVS approach would show little enhancement in the energy performance.




(a) Original IntraDVS (b) Single-Step LaIntraDVS (c) Multi-Step LaIntraDVS

Figure 5.11: An example program for multi-step look-ahead IntraDVS.

program modified using this idea. Since the variable $z$ could be used before the definition point at the line 7, we use the variable $\_z$ at the lines 3 and 4. (If the variable $z$ is not used before the line 7, we do not need to use the variable $\_z$.) If $x + y \leq 0$, the function func3 is executed with a lower speed in Figure 5.11(c) compared with in Figure 5.11(b).

Figure 5.12 summarizes the detailed steps of the multi-step LaIntraDVS algorithm. The algorithm has two functions. The function MS_LaVSP_Search does the same operations with the single-step LaIntraDVS algorithm except that it calls Find_MDP. The function Find_MDP finds the multi-step data predecessors. It first finds the predecessor set, $P$, for an input variable. Each predecessor $p$ in $P$ is examined whether there is an energy gain when the cycle distance between $s$ and $p$ is $Distance(p, s)$ and the overhead value is $C_{overhead}$. This is to consider the overhead instructions required for the multi-step LaVSP technique such as the line 3 in Figure 5.11(c).

If there is an energy gain in spite of the overhead cycles $C_{overhead}$, we further examine the data predecessor $p$. In this case, we call $p$ as the intermediate data predecessor. Then, the variables in the data predecessor $p$ are identified. For the data predecessor at the line 4 in Figure 5.11(a), it has the variables $x$ and $y$. We call the function Find_MDP with the variables recursively. The function also has the number of overhead cycles for the intermediate data predecessor $p$, $Overhead(p)$, as an input. If there is no energy gain due to a large $C_{overhead}$, the recursive function call is terminated. With this algorithm, we can find LaVSPs which can reduce the energy consumption despite of overhead instructions.

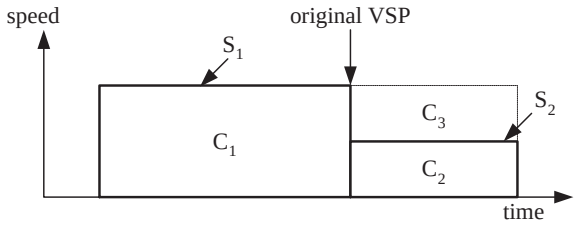
In transforming a program, the intermediate data predecessors are used as well as the conditions of the original voltage scaling point. For the variable which is defined in the intermediate data predecessors, we should use a copy of the variable (e.g., $\_z$ in Figure 5.11(c)) to preserve the program behavior.

Figure 5.13 shows how to estimate whether there is an energy gain when a LaVSP is used. In Figure 5.13(a), the clock speed is changed from $S_1$ to $S_2 = S_1 \cdot \frac{C_2}{C_3}$ at the original voltage scaling point because the remaining workload is changed from $C_3$ to $C_2$. In this case, the energy consumption can be computed by $E_{org} = C_1 S_1^2 + C_2 S_2^2$ assuming that supply voltage is proportional to clock speed.

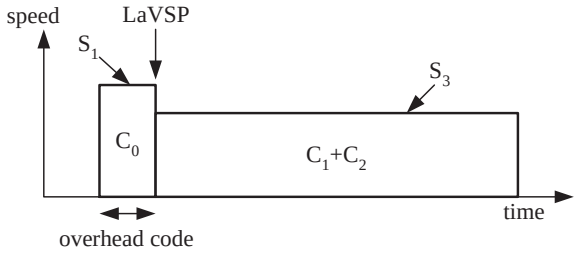
In Figure 5.13(b), LaIntraDVS found the look-ahead VSP which is executed $C_1$ cycles earlier than the original VSP. Assuming that we need $C_0$ overhead cycles to adjust the clock speed at the LaVSP, the energy consumption is given by $E_{La} = C_0 S_1^2 + (C_1 + C_2) S_3^2$ where $S_3$ is $S_1 \frac{C_1 + C_2}{C_1 + C_3 - C_0}$.

```
1: MS_LaVSP_Search(s) {
2:     C(s) := Find_Conditions(s);
3:     V(s) := ∅;
4:     for c_i ∈ C(s)
5:         V(s) := V(s) ∪ Find_Variables(c_i);
6:     for v_j ∈ V(s) {
7:         ℙ(s, v_j) := Find_MDP(s, v_j, 0);
8:         𝕃(s, v_j) := Look-ahead(ℙ(s, v_j));
9:     }
10:    LaVSP(s) := Merge(𝕃(s, v_1), ··· , 𝕃(s, v_n));
11:    Transform(LaVSP(s),C(s));
12: }
13:
14: Find_MDP(s, v_j, C_overhead) {
15:    P := Find_Data_Predecessor(s, v_j);
16:    for p ∈ P {
17:        if (EnergyGain(Distance(p, s), C_overhead)) return {s};
18:        V'(p) := Find_Variables(p);
19:        P := P − {p};
20:        for v_k ∈ V'(p)
21:            P : = P ∪ Find_MDP(p, v_k, Overhead(p));
22:    }
23:    return P;
24: }
```

Figure 5.12: Multi-step LaVSP search algorithm.




(a) Original IntraDVS




(b) LaIntraDVS

Figure 5.13: Overhead in LaIntraDVS.

The condition for LaIntraDVS to be more energy-efficient than the original IntraDVS technique is $E_{org} > E_{La}$.

$$\begin{aligned} E_{org} - E_{La} &= C_1 S_1^2 + C_2 S_2^2 - C_0 S_1^2 - (C_1 + C_2) S_3^2 > 0 \\ C_0 &< C_1 + C_2 (S_2/S_1)^2 - (C_1 + C_2)(S_3/S_1)^2 \\ C_0 &< C_1 + \frac{C_2^3}{C_3^2} - \frac{(C_1 + C_2)^3}{(C_1 + C_3 - C_0)^2} \end{aligned} \tag{5.6}$$

The function EnergyGain in Figure 5.12 checks this condition to decide whether there is an energy gain.

For L-type VSPs, it is not trivial to make the condition for the VSPs. In Figure 5.14(a), the while loop executes $\lceil (N-i)/k \rceil$ times. In the original IntraDVS technique, the variable $LoopIterNum$ is used to know the number of loop iteration. The voltage scaling code at the line 12 reduces the clock speed if $\lceil (N-i)/k \rceil$ is smaller than the maximum number of loop iterations,

$M$. Therefore, the condition for voltage scaling is $C(s) = \lceil (N-i)/k \rceil < M$. If we know the values of $i, k$, and $N$ in advance, we can reduce the clock speed before the while loop. However, it is not trivial to derive the number of loop iterations $\lceil (N-i)/k \rceil$ from a program. Using a parametric worst-case execution time analysis technique such as [45], we can know the number of loop iterations. But, we can use a simpler technique. For the L-type VSP, the loop termination condition and the multi-step LaIntraDVS technique are used.

For example, in Figure 5.14(a), the loop termination condition is $C(s) = \neg(i < N)$. (Note that the expression does not have the variable $k$.) By analyzing data predecessors for the variables in $C(s)$, we can get $\mathbb{P}(s, i) = \{3, 9\}$ and $\mathbb{P}(s, N) = \{1\}$. If we handle the data predecessor at the line 9 as an intermediate data predecessor, $\mathbb{P}(s, i)$ is changed into $\{2, 3\}$. Using $\mathbb{P}(s, i)$ and $\mathbb{P}(s, N)$, we can get the look-ahead voltage scaling point $LaVSP(s) = \{3\}$. Therefore, we can insert the voltage scaling codes after the line 3. Figure 5.14(b) shows the modified program by the multi-step LaIntraDVS. To reflect the condition $C(s) = \neg(i < N)$, the while statement is inserted at the line 6. An assignment statement is also inserted at the line 8 because the statement is related to an intermediate data predecessor.

### 5.2.4 Further Enhancements

The look-ahead IntraDVS is to move voltage scaling points to the LaVSPs where we can predict the direction of a control flow. The energy reduction by LaIntraDVS is significant when the distance between the original VSP and the LaVSP. Therefore, it is better to schedule the look-ahead voltage scaling

```
 1: N = func1();
 2: k = func2();
 3: i = func3();
 4: func4();
 5: LoopIterNum=0;
 6: while ( i < N ) {
 7:         LoopIterNum++;
 8:         func5();
 9:         i = i + k;
10: }
11: if (LoopIterNum < M)
12:         change_f_V();
13: func6();
```

(a) Original IntraDVS

```
 1: N = func1();
 2: k = func2();
 3: i = func3();
 4: _i = i;
 5: LoopIterNum = 0;
 6: while( _i < N ) {
 7:         LoopIterNum++;
 8:         _i = _i + k;
 9: }
10: if (LoopIterNum < M)
11:         change_f_V();
12: func4();
13: while ( i < N ) {
14:         func5();
15:         i = i + k;
16: }
17: func6();
```

(b) Look-ahead IntraDVS

Figure 5.14: An example program for L-type VSP.

points as early as possible at the compiler level. We call this instruction scheduling as an *LaIntraDVS-aware* instruction scheduling.

In the algorithm level, the *loop splitting* technique can be useful for LaIntraDVS. When a loop body has both the original VSP and the corresponding LaVSP, we split the loop into two separated loops which have the VSP and the LaVSP respectively. By the loop splitting, we can change the distance between the original VSP and the LaVSP.

Figure 5.15 shows the code transformation by loop splitting. In Figure 5.15(a), we assume that the execution cycles of functions funcA, funcB and funcC are 10, 10 and 20, respectively. When $N$ is 10, the worst-case execution cycles of this loop is 300 (when we consider only the execution cycles for functions). Whenever the function funcA returns 1, the voltage scaling point at the line 6 reduces the clock speed. The clock speed $f_5$ at the line 5 is changed to

$$f_7 = f_5 \cdot \frac{10 + 30 \cdot (9 - i)}{20 + 30 \cdot (9 - i)}$$

at the line 7 (assuming the voltage transition overhead is 0). Since the look-ahead voltage scaling point (line 5) is same to the original voltage scaling point, we cannot use the LaIntraDVS technique.

However, if we transform the program using loop splitting as shown in Figure 5.15(b), we can take full advantage of LaIntraDVS. While the original VSP is located in the second loop, the LaVSP is in the the first loop. Whenever the value of each $a[i]$ is determined at the first loop, we can reduce the clock speed at the LaVSP at the line 6. If the clock speed at the line 4 is $f_4$, the clock speed is changed to

$$f_4 \cdot \frac{10 \cdot (i + 1) + 20 \cdot (9 - i)}{10 \cdot i + 20 \cdot (10 - i)}$$

by the LaVSP. Figure 5.15(c) shows the speed change graphs of two programs. The clock speed of the program transformed by the loop splitting is reduced more quickly and doe not change during the execution of the second loop. If we assume that the energy consumption is proportional to the square of the clock speed, the LaIntraDVS technique with loop splitting reduces the energy consumption by 15% in this example.

Another enhancement technique for LaIntraDVS is the *function inlining*. For the program in Figure 5.16(a), a voltage scaling point is the line 12 because the function funcC is not executed when $i > 0$. The data predecessor of the variable $i$ is the line 2 of the function funcA. But, the line 2 is not a look-ahead point of $i$ because the function funcA is called at the line 8 with the input variable $j$. Therefore, we cannot move the voltage scaling point to the line 3. If we inline the function funcA to the line 6 as shown in Figure 5.16(b), the line 6 becomes the look-ahead point of the variable $i$. LaIntraDVS inserted the voltage scaling point to the line 7-8.

### 5.2.5 Experiments

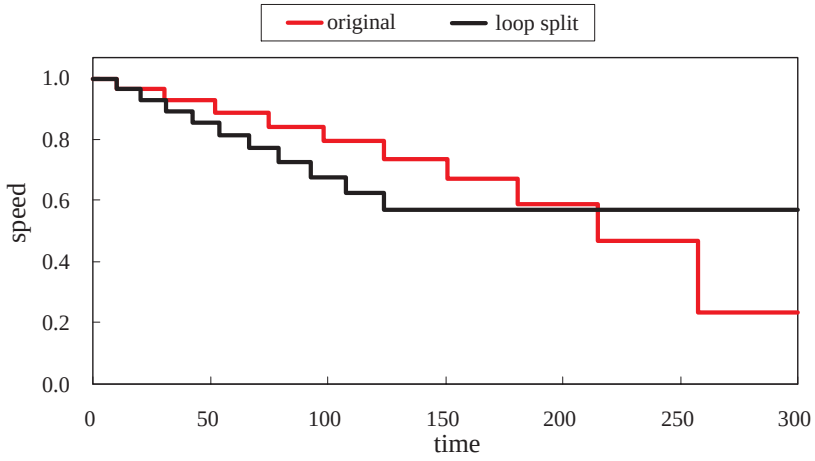
In order to evaluate the energy efficiency of LaIntraDVS techniques, we have experimented with an MPEG-4 video encoder and an MPEG-4 decoder. We first made a framework for LaIntraDVS as shown in Figure 5.17. We used the same automatic voltage scaler (AVS) introduced at the previous chapter. AVS takes a target program as an input and generates the VSP information using the original IntraDVS algorithm. The Look-ahead VSP Analyzer generates the look-ahead points for each VSP using the algorithm in Figure 5.12. The Data Flow Analyzer finds the data predecessors for

```
 1: int a[N];
 2:
 3: for (i=0; i<N; i++) {
 4:     a[i] = funcA();
 5:     if (a[i]) {
 6:         change_f_V();
 7:         funcB();
 8:     }
 9:     else
10:         funcC();
11: }
```

(a) Original Program

```
 1: int a[N];
 2:
 3: for (i=0; i<N; i++) {
 4:     a[i] = funcA();
 5:     if(a[i])
 6:         change_f_V();
 7: }
 8: for (i=0; i<N; i++) {
 9:     if (a[i])  funcB();
10:     else       funcC();
11: }
```

(b) Transformed Program




(c) Speed change graph

Figure 5.15: Code transformation: loop splitting.

```
 1: void funcA(int *a) {
 2:         *a =  funcF();
 3: }
 4:
 5: void main() {
 6:     funcA(&i);
 7:     funcB();
 8:     funcA(&j);
 9:     if (i > 0)
10:         funcC();
11:     else
12:         change_f_V();
13:     funcD();
14: }
```

(a) Original program

```
 1: void funcA(int *a) {
 2:         *a =  funcF();
 3: }
 4:
 5: void main() {
 6:     i =  funcF();
 7:     if (!(i >0))
 8:         change_f_V();
 9:     funcB();
10:     funcA(&j);
11:     if (i > 0)
12:         funcC();
13:     funcD();
14: }
```

(b) Transformed program

Figure 5.16: Code transformation: function inlining.

each VSP using the data flow analysis technique. The Data Flow Analyzer corresponds to the function Find_Data_Predecessor in Figure 5.12. Using the look-ahead VSP information, AVS generates the DVS-aware program.

Figure 5.18 shows the energy consumption of three kinds of MPEG-4 encoder programs, which employ the original IntraDVS, the single-step LaIntraDVS and the multi-step LaIntraDVS, respectively. The figure also compares the results for the RWEP-based techniques and the RAEP-based techniques. The energy consumption is normalized by the result of the RWEP-based IntraDVS technique. As shown in Figure 5.18, the single-step LaIntraDVS reduced the energy consumption only by 4∼6%. This is because most of look-ahead points are located closely to the original VSPs. However, the multi-step LaIntraDVS shows significant energy reductions of

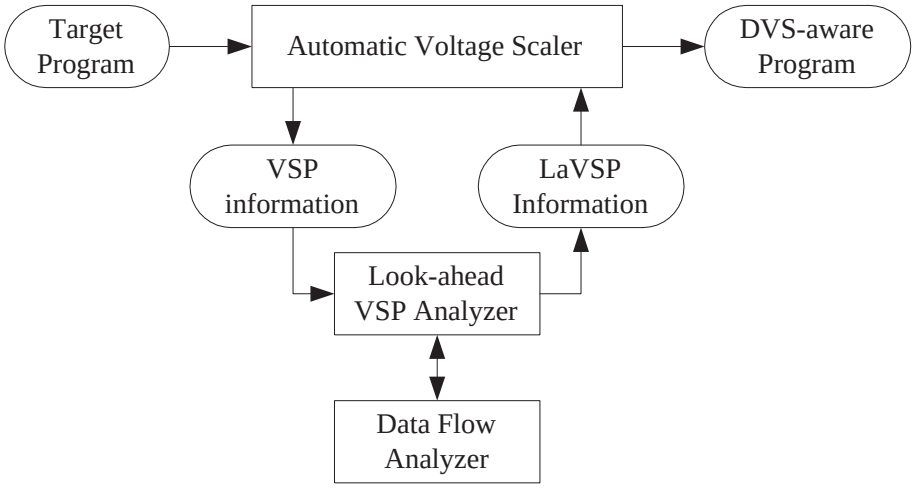



Figure 5.17: The framework for look-ahead IntraDVS.

40∼45%.

The energy performance of LaIntraDVS is dependent on the application characteristic. For an MPEG-4 decoder program, even the multi-step LaIntraDVS shows little energy reductions. Generally, the applications, which consist of several steps and the execution cycles of the $i^{th}$ step is dependent on the result of the $(i-1)^{th}$ step, is not improved by the LaIntraDVS. But, the LaIntraDVS can enhance the energy performances of the applications whose execution cycles are determined by input variables. For example, GUI applications can predict the execution time when the menu selection event occurs.

Another issue for the application characteristic is its slice size. Weisr [46] introduced the concept of program slice, which allows the user to focus on the portion of the program responsible for a particular phenomenon. There are two kinds of slices, i.e., backward slice and forward slice. While a

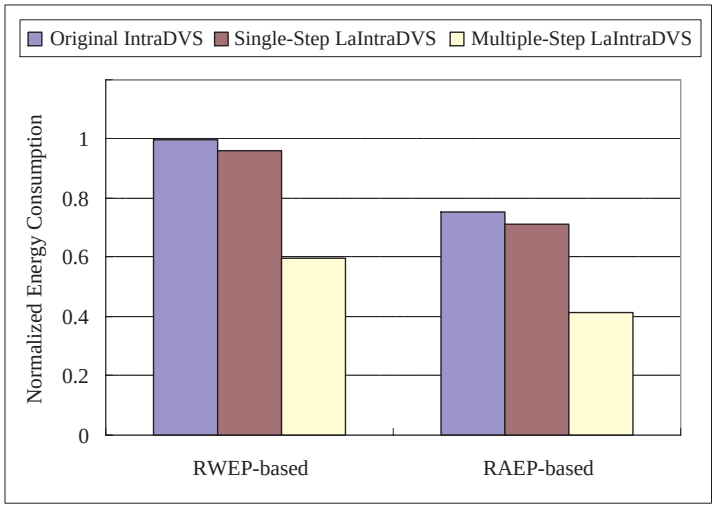



Figure 5.18: Experimental results of look-ahead IntraDVS.

backward slice consists of all program points that affect a given point in a program, a forward slice consists of all program points that are affected by a given point in a program. When we use the multi-step LaIntraDVS technique, a portion of the backward slice of a VSP should be cloned before the LaVSP point. Therefore, if the size of backward slice is large, $C_{overhead}$ becomes large. We can say that the size of backward slice limits the energy performance of LaIntraDVS.

The slice size is dependent on the target program point. However, average slice size is considerably smaller than the original code [46, 47]. The multi-step LaIntraDVS applied to MPEG-4 encoder program inserted only four C-statements.

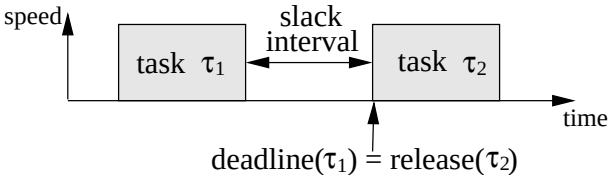
# Chapter 6

# Cooperative IntraDVS under OS-Level Voltage Scheduler

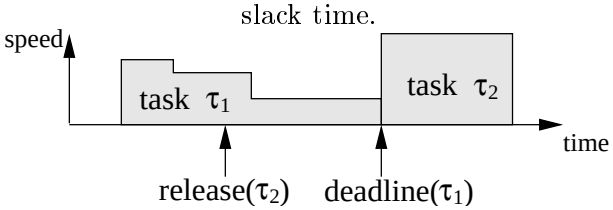
## 6.1 Motivation

We have compared the energy efficiency of the InterDVS algorithms and the IntraDVS algorithm in Chapter 4. However, there are cases where pure IntraDVS or pure InterDVS dose not work well. Figure 6.1 illustrates such cases.

In Figure 6.1(a), when an InterDVS algorithm is used, the slack time generated by the task $\tau_1$ cannot be used by the task $\tau_2$ because the release time of the task $\tau_2$ is same to the deadline of the task $\tau_1$. This slack time could be used if the task $\tau_1$ were scheduled using an IntraDVS algorithm. On the other hand, in Figure 6.1(b), when an IntraDVS algorithm is used, all the slack times generated by the task $\tau_1$ are used by the task $\tau_1$. However,




(a) The case where InterDVS can not utilize the slack time.




(b) The case where the slack distribution is not balanced due to IntraDVS.

Figure 6.1: Cases where pure InterDVS or pure IntraDVS performs poor.

this slack distribution is unbalanced. If we used InterDVS, we could get a more efficient schedule by distributing the slack time of $\tau_1$ for the task $\tau_2$.

From this example, we can know that the cooperation between IntraDVS and InterDVS is necessary for energy efficient systems. As shown in Figure 6.1(b), when the speed of an IntraDVS-enabled task is too low but there are released other tasks, it is better to stop voltage scaling in the task and transfer slack times to the released tasks. However, if there is no released task, the IntraDVS-enabled task should continue to adjust the clock speed.

It is impossible to find the optimal solution for the slack distribution because we have no knowledge of the future workload. Therefore, some heuristics are required, which can balance the slack time usage.

## 6.2 Hybrid DVS algorithms

In this section, we investigate whether hybrid DVS algorithms (HybridDVS algorithms) with both IntraDVS and InterDVS features perform better than pure IntraDVS algorithms or pure InterDVS algorithms.

HybridDVS algorithms select either the *intra mode* or the *inter mode* when slack times are available during the execution of the current task. At the inter mode, the slack time is used not for the current task but for the following tasks. Therefore, the speed of the current task is not changed by the slack time produced by the current task. At the intra mode, all the slack time is used for the current task, reducing its own execution speed.

Table 6.1 summarizes six heuristics for HybridDVS algorithms we consider in this section. The heuristics are different in that how close they are to the pure IntraDVS approach or pure InterDVS approach. H0 is identical to the pure InterDVS approach. H1 and H2 are closer to the pure InterDVS approach while H4 and H5 are closer to the pure IntraDVS approach.

The heuristic H1 is motivated to solve the case shown in Figure 6.1(a). H1 uses the intra mode only when there is no following task which can utilize the slack time from the current task. When the current task $\tau$ meets a VSE $(b_i, b_j)$ at the time $t$, H1 compares the expected completion time $ect(\tau)$ of $\tau$ and the next task arrival time $NTA$. When the remaining worst-case execution cycles and the clock speed of $\tau$ are $C_{RWEC}(t)$ and $S(t)$ at the time $t$ respectively, the $ect(\tau)$ is $t + C_{RWEC}(t)/S(t)$. If $ect(\tau) \geq NTA$, the HybridDVS H1 does nothing at the VSE. However, H1 scales down the clock speed when $ect(\tau) < NTA$. The speed change algorithm is same to

Table 6.1: Heuristics for HybridDVS algorithms.

| Heuristic | Description |
|---|---|
| H0 | always uses the inter mode (i.e., the pure InterDVS approach). |
| H1 | uses the inter mode as a default but uses the intra mode if no activated task exists. |
| H2 | uses the inter mode at first, but changes into the intra mode when the unused slack time is more than a predefined amount of slack time. |
| H3 | alternates the intra mode and the inter mode keeping the balance of slack consumption in each mode. |
| H4 | uses the intra mode at first, but changes into the inter mode when the current task has used a predefined amount of slack time. |
| H5 | always uses the intra mode. |

the IntraDVS's algorithm. The current speed $S(t)$ is changed to

$$S(t) \times \frac{C_{RWEC}(b_i) - C_{EC}(b_i)}{C_{RWEC}(b_j)}.$$

Under this algorithm, there is no task which completes before the next task arrival time (when we assume that there is no overhead time on voltage scaling) thus the system has no idle time.

The heuristic H4 is devised to solve the case shown in Figure 6.1(b). H4 uses the intra mode temporarily for the current task. So, it scales down the clock speed at VSEs. However, when the current task has used too much slack times thus the clock speed is too low, it changes the operating mode to the inter mode. We added the idea of the heuristic H1 to this heuristic H4. Even though the operating mode is transferred to the inter mode, it changes into the intra mode if the expected completion time is earlier than the next task arrival time. The issue of H4 is to determine when we change the operating mode.

H4 assigns the maximum slack time (MST) to each task before the task's execution. To balance the total slack time among tasks, the MST should be assigned so that the task, whose average execution cycles are large, has a large MST. Generally the execution cycles of a task has a temporal locality. so, we use the information about past task executions. The MST can be estimated as follows:

$$MST(\tau) = c(\tau)(\frac{1}{U_{tot}} - 1) \tag{6.1}$$

$c(\tau)$ is the average execution cycles based on the past executions of $\tau$. This is updated at every completion of the task instance. $U_{tot}$ is the total processor utilization until the current time.

During the task execution, H4 manages the total used slack time (UST). The UST is initially the slack time which is transferred from the previously executed tasks. When the task execution meets a VSE at the intra mode, H4 adds the saved cycles at the VSE to UST. If UST is smaller than MST, it sustains the intra mode. But, when UST becomes to be larger than MST, it changes into the inter mode.

The heuristic H2 is exactly the opposite of H4. H2 also assigns the maximum slack time to each task. Each task starts from the inter mode. At each VSE, it does not scale down the clock speed but adds the saved cycles at the VSE to the transferred slack time (TST). When the TST is larger than ($w(\tau) - c(\tau)$−MST), the task enters into the intra mode, where $w(\tau)$ is the worst-case execution cycles. The value of $w(\tau) - c(\tau)$ means the expected slack time from the task $\tau$.

While the heuristics H2 and H4 manage the absolute value of slack consumption, the heuristic H3 keeps the balance of slack consumption. It al-

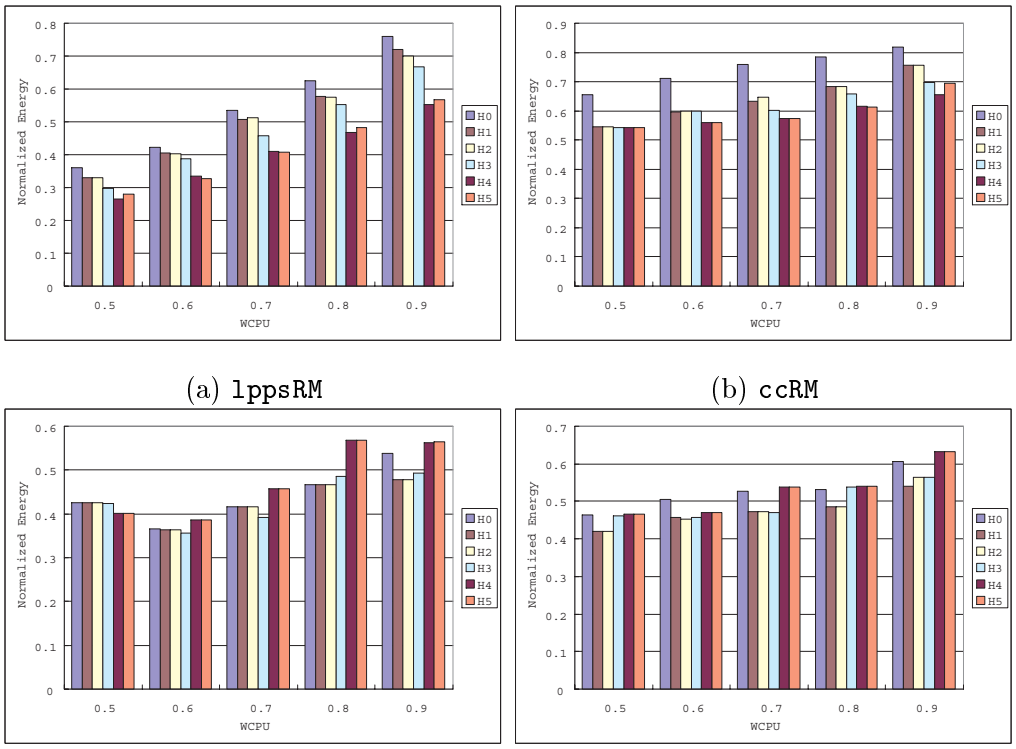
ternates the intra mode and the inter mode keeping the ratio between UST and TST at the value of $MST/(w(\tau) - c(\tau) - MST)$.

## 6.3 Experiments

We have evaluated six heuristics in Table 6.1 with four InterDVS algorithms. Figure 6.2 shows the energy efficiency comparison results of the HybridDVS algorithms over the power-down method varying WCPUs. In the power-down method, active tasks execute with the full speed. When there is no active task, the system enters into the power-down mode. The HybridDVS algorithms, H1, H2, H3 and H4, generally reduce the energy consumption by 5∼27% over that of the pure DVS algorithms, H0 and H5.

Figure 6.2 shows that the energy efficiencies of HybridDVS algorithms are strongly affected by the efficiency of the on-line slack estimation method used by each InterDVS algorithm. In `laEDF` [11], `DRA` [10] and `AGR` [10] where slack times are aggressively identified, it is a good idea that some (or all) of slack time produced by the current task is passed to the following tasks (as in Figure 6.1(b)). Especially, `laEDF` and `AGR` identify the slack time for a task assuming all following tasks will be executed with the full speed. With the slack estimation method, the high-priority or early-released tasks has a long slack time. In this scheme, both HybridDVS and IntraDVS shows poor energy performance.

However, in `lppsEDF/RM` [8] and `ccEDF/RM` [11] where slack times are less aggressively identified, there are many cases where the current slacks are wasted unless used by the current task (as in Figure 6.1(a)). In this case,




(a) `lppsRM` (b) `ccRM`

(c) `laEDF` (d) `AGR`

Figure 6.2: Energy efficiency comparison results of the HybridDVS algorithms.

it is better for the current task to utilize most of the slack time generated. Therefore, if a HybridDVS algorithm is based on `laEDF`, `DRA` or `AGR`, H1 and H2 are better choices. On the other hand, for `lppsEDF/RM` and `ccEDF/RM`, H4 and H5 are better choices.

Figure 6.3 shows the spectrum of HybridDVS heuristics, and summarizes well-matching hybrid heuristics for each InterDVS algorithm. For example, if `laEDF` is extended to a HybridDVS algorithm, H1 is a good candidate for a matching hybrid heuristic. However, if `lppsRM` is modified for a hybrid DVS algorithm, H4 is a better hybrid heuristic.

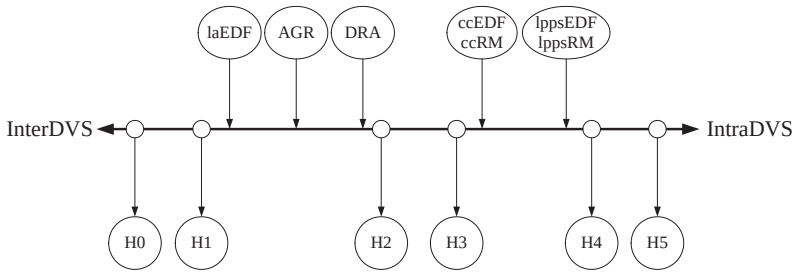



Figure 6.3: Spectrum of HybridDVS heuristics.

# Chapter 7

# Conclusions

## 7.1 Summary and Contributions

Embedded and ubiquitous computing are emerging rapidly as exciting new paradigms to provide computing and communication services all the time, everywhere. This emergence is a natural outcome of research and technological advances in embedded systems, pervasive computing and communications, wireless networks and mobile computing, etc. In these systems, energy efficient system design is a critical issue considering the limited battery capacity.

Most of these embedded systems are time critical real-time systems. Although classic real-time design techniques address the scheduling problem for the various system resources such as CPU cycle and I/O bandwidth, they do not take into account the energy efficiency. However, with the advent of the variable-voltage processors, we are facing the scheduling problem to

select the right speeds for each task.

The current dissertation presented several intra-task clock and voltage scheduling algorithms for systems containing variable-voltage processors. They make energy-efficient real-time systems, satisfying the hard real-time constraints. The dissertation started with a presentation of the relevant background, followed by a review of related research.

Next, we have proposed an intra-task voltage scheduling framework for low-energy hard real-time applications. By statically analyzing the timing behavior of a DVS-unaware real-time program, the proposed technique automates two time-consuming and complicated steps of applying intra-task voltage scheduling to DVS-unaware programs. First, the proposed technique automatically selects appropriate program locations where the supply voltage can be changed to minimize the energy consumption satisfying the timing constraint. Second, the proposed technique inserts to the selected program locations voltage scaling code in a completely transparent fashion to programmers.

By automating these two steps, the proposed algorithm makes it possible for programmers *without any knowledge on DVS* to develop DVS-aware programs on a variable-voltage processor. The converted program by the proposed scheduling algorithm has a unique characteristic that it always completes its execution near the deadline, thus resulting in no slack time. By lowering the execution speed and corresponding voltage to the maximum allowable extent, the proposed algorithm achieves a significant energy reduction ratio.

Based on the proposed intra-task voltage scaling framework, we have

built an automatic voltage scaling tool, AVS. It automatically transforms a DVS-unaware program to a DVS-aware low-energy program with the same functional behavior and timing requirement.

Next, we have described two kinds of improvement techniques for the IntraDVS algorithm, i.e. using profile information and using data flow analysis. While the worst-case timing information is used in the basic IntraDVS (RWEP-based IntraDVS), the average-case timing information is used for a better energy performance in the IntraDVS using profile information (RAEP-based IntraDVS). The IntraDVS using data flow analysis finds the voltage scaling points based on the data flow information as well as control flow information (Look-ahead IntraDVS).

Next, we have proposed hybrid voltage scheduling algorithms to cooperate with OS-level InterDVS algorithms. Though IntraDVS exploits all slack times within a task boundary, there are cases where it is better to transfer slack times to following tasks. The hybrid DVS algorithms determines the slack distribution observing the current system status.

The experimental results using simulations with an MPEG-4 video encoder and decoder showed that AVS using RWEP-based IntraDVS improves the energy efficiency of the programs by a factor of 2.6$\sim$3.4 over the programs running on a non-DVS system with a power-down mode. The RAEP-based IntraDVS improved the energy efficiency up to 34% over the RWEP-based IntraDVS. The look-ahead IntraDVS reduced the energy consumption by $40 \sim 45\%$.

In the experiment using a real DVS-enabled system providing a finite number of clock/voltage levels, the low-energy version of an MPEG-4 en-

coder/decoder consumed only 35∼51% of the energy consumption from the original program running on a fixed-voltage system with a power-down mode. We also compared the energy efficiencies of the IntraDVS algorithm and InterDVS algorithms. Although the IntraDVS algorithm generally outperformed the InterDVS algorithms, the relative energy efficiency was dependent on the task set characteristics.

## 7.2 Future Works

### 7.2.1 IntraDVS Using Frequency-Aware Timing Analysis

In this dissertation, we assumed that reducing a processor's clock frequency still results in the same number of execution cycles for a task. However, this simplistic view generally does not hold for any realistic architectures. Consider the impact of memory references. Any instruction or data reference that is resolved through a main memory access operates at external bus frequency. But bus frequencies generally diverge from internal processor frequencies, and they do not scale at the same rate as DVS scaling does. E.g., the first generation Compaq IPAQ has a StrongArm microprocessor (SA-1110) that scales at 8 frequencies but only supports two different external bus frequencies. By assuming that the WCEC remains constant, one ignores the fact that the WCEC reduces with frequency, which results in WCET overestimations. Therefore, an exact WCEC model should be frequency-aware. For example, Seth *et al.* [48] proposed the frequency-aware WCEC model assuming a constant access latency for memory references regard-

less of changing processor frequencies. They represented the WCEC of a program as follows:

$$WCEC = i + m \cdot L \cdot f \tag{7.1}$$

where $i$ is the total number of cycles used for non-memory operations and $m$ is the total number of memory references respectively. $L$ is the latency of the memory and $f$ is the frequency of the processor.

There are some obstacles to use the frequency-aware WCEC model in our IntraDVS techniques. First, RWEPs can be changed depending on the frequency. For example, in Equation (7.1), assume that $L$ is 5 and the pairs of $(i, m)$ of two remaining paths $p_1$ and $p_2$ are (50,10) and (30,16) respectively. If $f = 1$, $WCEC(p_2)$ is longer than $WCEC(p_1)$ ($WCEC(p_1) = 50 + 10 \cdot 5 \cdot 1 = 100$ and $WCEC(p_2) = 30 + 16 \cdot 5 \cdot 1 = 110$). But, if $f = 0.5$, $WCEC(p_1)$ is longer than $WCEC(p_2)$ ($WCEC(p_1) = 50 + 10 \cdot 5 \cdot 0.5 = 75$ and $WCEC(p_2) = 30 + 16 \cdot 5 \cdot 0.5 = 70$). This phenomenon prohibits from selecting voltage scaling edges statically. In addition, we cannot select VSEs from VSE candidates statically because the saved cycles at VSE candidates are different depending on clock frequency. ($C_{saved} = 10$ at $f = 1$ and $C_{saved} = 5$ at $f = 0.5$) Therefore, we should insert voltage scaling edges all possible points. At run time, it is should be checked whether the point is a VSE or not.

Second problem occurs when the memory has a complex model for access time. For example, Intel's PXA250 provides different memory bus frequency depending on the CPU clock frequency. In this case, we cannot use a simple formula such as Equation (7.1) as the WCEC model. Then, DVS algorithm should provide all solutions for all configurations and use one of

them at run time. This increases the overhead time and code size for voltage scaling. We are to investigate the efficient implementation techniques such that IntraDVS can use the frequency-aware timing analysis.

### 7.2.2 IntraDVS Using Run-Time Monitoring

The execution time of an application depends on the run-time hardware events (e.g., cache miss and TLB miss) as well as the control flow. We plan to integrate such run-time information into the proposed IntraDVS framework so that the energy efficiency could be further improved. To use the hardware events, the hardware monitor is required. Fortunately, recent processors provide hardware event counters. For example, for Intel's XScale processor, we can know the number of pipeline stalls, cache misses, TLB misses, and branch mispredictions using the *performance monitor count register* [4].

Recently, choi *et al.* [49] proposed a dynamic voltage scaling technique for MPEG decoding which using the performance-monitoring unit in XScale processor. They partitioned the computational workload in decoding a frame as on-chip and off-chip workload by using a dynamic event from the performance-monitoring unit. The on-chip workload for an incoming frame is predicted using a frame-based history so that the processor voltage and frequency can be scaled to provide the exact amount of computing power needed to decode the frame.

In using the hardware events for IntraDVS, unlike the technique based on control flow, we cannot adjust the clock/voltage at every hardware events. It is better to piggyback the saved cycles by hardware events on VSEs. At

each VSE, the voltage scaling code should check the saved cycles from both the control flow and the hardware events.

### 7.2.3 IntraDVS Considering Static Power

Although we have focused on the dynamic power consumption assuming the static part can be ignored, the static power will become a significant portion of the total power. [50] reported that the leakage power currently accounts for about $15 \sim 20\%$ of the total power for high speed processors. Recently, several techniques have been proposed, which minimize the leakage power. For example, drowsy cache [50] reduces the leakage power by lowering the supply voltage to the least upper bound that preserves state (drowsy state). Drowsy cache periodically clear all active lines to the drowsy state. Word lines in the drowsy state return to the active state when accessed. There are overhead time to awake from the drowsy state. With the drowsy cache, it is advantageous to have long idle times. This means that we may increase the leakage energy consumption using DVS techniques because idle times are reduced by stretching the execution times of tasks by DVS techniques. Therefore, we should consider the leakage power as well as the dynamic power at future systems.

For example, assume a task has an execution cycle $C$ and a deadline $D$. If the task consumes the dynamic energy $E$ at the maximum clock speed $f_m$, we can reduce the energy consumption to $E \cdot (f/f_m)^2$ by reducing the clock speed to $f \leq f_m$. When we use a drowsy cache, which consumes the leakage power $L$ if a task is activated but consumes no leakage power if the system is idle, the leakage energy at the clock frequency $f$ is $C \cdot L/f$. Then,

total energy consumption is $E \cdot (f/f_m)^2 + C \cdot L/f$. The clock frequency $f$, which minimizes the energy consumption, is $\sqrt[3]{\frac{C \cdot L \cdot f_m^2}{2E}}$. We should use this clock speed to minimize the total energy consumption.

We will further research into the IntraDVS scheduling considering the leakage energy. A more detail model for leakage energy is required.

### 7.2.4 Inter-Task DVS Using Intra-Task Slack Detection

Though we used the slack information detected within a task for IntraDVS, the information can also be useful for the InterDVS. When a task is resumed after preemption, general inter-task scheduling algorithms determine the clock speed based on the RWEC of the task. For example, assume that there is a preempted task, which has a WCEC $W$ and already executed $C$ cycles. InterDVS algorithms determine the clock speed of the task based on the RWEC $W - C$. Namely, the RWEC is estimated by subtracting the consumed cycles from the WCEC. This is not an exact value because the RWEC is changed by the slack time within the task. If the inter-task voltage scheduler can know the exact RWEC, it can further reduce the clock speed with the information. Furthermore, the inter-task voltage scheduler can utilize both the current task's RWEC information and the global slack information to make a better decision about the clock speed.

At each VSE, if a task only informs the change of RWEC to the operating system instead of voltage scaling, the operating system records the RWEC to the data structure for the task and can use when the task is resumed after

preemption. The preliminary result using the lppsEDF algorithm suggests that the energy can be reduced by about 5%. We have a plan to devise detail algorithms for various InterDVS algorithms.

# Bibliography

[1] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A Dynamic Voltage Scaled Microprocessor System. In *Proc. of IEEE International Solid-State Circuits Conference*, pages 294–295, 2000.

[2] M. Fleischmann. Crusoe Power Management: Reducing the Operating Power with LongRun. In *Proc. of HotChips 12 Symposium*, 2000.

[3] AMD, Inc. AMD PowerNow Technology, 2000.

[4] Intel, Inc. The Intel(R) XScale(TM) Microarchitecture Technical Summary, 2000.

[5] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc. of the 36th Annual Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[6] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processor. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 178–187, 1998.

[7] T. Okuma, T. Ishihara, and H. Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. In *Proc. of International Symposium On System Synthesis*, pages 24–29, 1999.

[8] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proc. of Design Automation Conference*, pages 134–139, 1999.

[9] Y. Lee and C. M. Krishna. Voltage-Clock Scaling for Low Energy Consumption in Real-Time Embedded Systems. In *Proc. of the 16th International Conference on Real-Time Computing Systems and Applications*, pages 272–279, 1999.

[10] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *Proc. of IEEE Real-Time Systems Symposium*, 2001.

[11] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[12] W. Kim, J. Kim, and S. L. Min. A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis. In *Proc. of Design Automation and Test in Europe*, pages 788–794, 2002.

[13] S. Lee and T. Sakurai. Run-Time Voltage Hopping for Low-Power Real-Time Systems. In *Proc. of Design Automation Conference*, pages 806–809, 2000.

[14] D. Shin, J. Kim, and S. Lee. Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications. *IEEE Design and Test of Computers*, 18(2):20–30, 2001.

[15] D. Shin and J. Kim. A Profile-Based Energy-Efficient Intra-Task Voltage Scheduling Algorithm for Hard Real-Time Applications. In *Proc. of International Symposium on Low Power Electronics and Design*, pages 271–274, 2001.

[16] D. Shin, W. Kim, J. Jeon, and J. Kim. SimDVS: An Integrated Simulation Environment for Performance Evaluation of Dynamic Voltage Scaling Algorithms. *Lecture Notes in Computer Science*, 2325:141–156, 2003.

[17] C. Y. Park and A. C. Shaw. Experiments with A Program Timing Tool Based on Source-Level Timing Schema. In *Proc. of the 11th Real-Time Systems Symposium*, pages 72–81, 1990.

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

[19] Y. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, pages 254–263, 1996.

[20] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. In *Proc. of the 3rd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, 1997.

[21] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[22] J. M. Rabaey and M. Pedram. *Low Power Design Methodologies*. Kluwer Academic Publishers, 1996.

[23] S. Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999.

[24] T. Sakurai and A. Newton. Alpha-Power Law MOSFET Model and Its Application to CMOS Inverter Delay and Other Formulas. *IEEE Journal of Solid State Circuits*, 25(2):584–594, 1990.

[25] Texas Instruments, Inc. Using the Power Scaling Library on the TMS320C5510, 2002.

[26] M. Fleischmann. LongRun Power Management - Dynamic Power Management for Crusoe Processors. Technical report, Transmeta Corporation, 2001.

[27] K. J. Nowka, Gary D. Carpenter, Eric W. MacDonald, Hung C. Ngo, Bishop C. Brock, Koji I. Ishii, Tuyet Y. Nguyen, and Jeffrey L. Burns. A 32-bit PowerPC System-on-a-Chip With Support for Dynamic Voltage Scaling and Dynamic Frequency Scaling. *IEEE Journal of Solid-State Circuits*, 37(11):1441–1447, 2002.

[28] R. Hamburgen, D. Wallach, M. Viredaz, L. Brakmo, C. Waldspurger, J. Bartlett, T. Mann, and K. Farkas. Itsy: Stretching the Bounds of Mobile Computing. *IEEE Computer*, 34(4):28–36, 2001.

[29] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *Proc. of Mobile Computing Conference(MOBICOM)*, pages 251–259, 2001.

[30] F. Gruian. Hard Real-Time Scheduling Using Stochastic Data and DVS Processors. In *Proc. of International Symposium on Low Power Electronics and Design*, pages 46–51, 2001.

[31] Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. In *Proc. of International Conference on Computer-Aided Design*, pages 365–368, 2000.

[32] G. Quan and Xiaobo Sharon Hu. Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors. In *Proc. of Design Automation Conference*, pages 828–833, 2001.

[33] W. Kim, D. Shin, H.-S. Yun, J. Kim, and S. L. Min. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 219–228, 2002.

[34] C. Im, H. Kim, and S. Ha. Dynamic Voltage Scheduling Technique for Low-Power Multimedia Applications Using Buffers. In *Proc. of International Symposium On Low Power Electronics and Design*, pages 34–39, 2001.

[35] I. Hong, M. Potkonjak, and M. B. Srivastava. On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *Proc. of International Conference on Computer Aided Design*, pages 653–656, 1998.

[36] D. Shin and J. Kim. Dynamic Voltage Scaling of Periodic and Aperiodic Tasks in Priority-Driven Systems. In *Proc. of Asia South Pacific Design Automation Conference*, pages 653–658, 2004.

[37] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[38] J. R. Lorch and A. J. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proc. of ACM SIGMETRICS Conference*, pages 50–61, 2001.

[39] C-H. Hsu and U. Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Proc. of ACM SIGPLAN Conference on Programming Languages, Design, and Implementation*, pages 38–48, 2003.

[40] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, 1995.

[41] F. Gruian. On Energy Reduction in Hard Real-Time Systems Containing Tasks with Stochastic Execution Times. In *Proc. of IEEE Workshop on Power Management for Real-Time and Embedded Systems*, pages 11–16, 2001.

[42] T. Ball and J. R. Larus. Using Paths to Measure, Explain, and Enhance Program Behavior. *IEEE Computer*, 33(7):57–65, 2000.

[43] T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proc. of International Symposium On Low Power Electronics and Design*, pages 197–202, 1998.

[44] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[45] Bjorn. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proc. of International Workshop on Worst-Case Execution Time Analysis*, pages 85–88, 2003.

[46] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[47] L. Bent, D. C. Atkinson, and W. G. Griswold. A Comparative Study of Two Whole Program Slicers for C. Technical Report CS2001-0668, Dept. of Computer Science and Engineering, University of California at San Diego, 2001.

[48] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. FAST: Frequency-Aware Static Timing Analysis. In *Proc. of the 24th IEEE International Real-Time Systems Symposium*, pages 40–51, 2003.

[49] K. Choi, R. Soma, and M. Pedram. Off-chip Latency-Driven Dynamic Voltage and Frequency Scaling for an MPEG Decoding. In *Proc. of Design Automation Conference*, 2004.

[50] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proc. of the 29th Internal Symposium on Computer Architecture*, 2001.

# Appendix

## A. DVS Hardware Platforms

Nowadays, it is easy to get a DVS platform. Here we briefly describe eight solutions, implemented by various embedded processors. The first two solutions are the result of academic research projects at UC Berkeley and Delft University respectively. The rest are industry developments by Transmeta, AMD, Intel, IBM, Texas Instruments and Compaq.

**UC Berkeley's lpARM** The lpARM processor, developed at UC Berkeley, is a low power, ARM core based architecture, capable of run-time voltage and clock frequency changes. The prototype described in [1] (0.6 technology) is, reportedly, able to run at clock frequencies in the 5-80 MHz range, with 5 MHz increments. The supply voltage is adjustable in the 1.2-3.8 V range. It includes an ARM8 core running at a clock frequency produced by an on-chip voltage-controlled oscillator (VCO). On lpARM, a speed switch from 5MHz@1.2V to 80MHz@3.8V takes a 70 $\mu$s. Due to the particular design of the lpARM, the processor can continue executing instructions while switching speeds. Although the transition between speeds

is not instantaneous, the property that the processor can continue operating while switching, makes the actual latency much smaller than the speed switch.

**TU Delft's LART** The LART is built by Delft University using Intel SA-1100 StrongARM processor. It can operate at clock speeds ranging from 59 to 221 MHz. The supply voltage is adjustable in the range of 0.8 V (at 59 MHz) to 1.5 V (at 221 MHz). They extended the Linux kernel with a module that changes the clock frequency and core voltage. The kernel module can be accessed from user-space by writing the desired frequency in the '/proc/scale' pseudo file; the kernel uses a lookup table to select the minimum core voltage at which the processor still functions for the selected frequency. The speed switch from 59 MHz to 221 MHz takes 140 $\mu$s. The voltage transition times are different depending on the directions. The voltage increase is rapidly handled (40 $\mu$s), but the decrease takes a long time (5.5 ms). This is caused by the high capacitance of the regulator and the low power deman of the processor at 59 MHz.

**Transmeta's Crusoe with LongRun** Crusoe is a Transmeta processor family (TM5x00), with a VLIW core and x86 Code Morphing software that provides x86-compatibility. Besides four power management states, the processor supports run-time voltage and clock frequency hopping. Frequency can change in steps of 33 MHz and the supply voltage in steps of 25mV, within the hardware's operating range. The number of available speeds depends on the model. The TM5600 model for example, operates in normal mode between 300-667 MHz and 1.2-1.6V [26], meaning eleven

different speed settings. The corresponding power consumption varies between 1.5W and 5.5W. The speed is decided using feedback from the Code Morphing algorithm, which reports the utilization. The LongRun manager employs this feedback to compute and control the optimal clock speed and voltage. Note that this is a fine grain control, transparent to the programmer. The algorithms we present in this dissertation require direct control over the processor speed, and would substitute or augment LongRun. Nevertheless, the Crusoe architecture is a successful example of a variable speed processor, widely used in low power systems. A comparison with a conventional mobile x86 processor using Intel SpeedStep, running a software DVD player, reported in [26], shows that TM5600 consumes almost three times less power than the mobile x86 (6W for TM5600 vs. 17W for the mobile x86).

**AMD's K6 with PowerNow!** AMD's PowerNow! technology controls the level of processor performance automatically, dynamically adjusting the operating frequency and voltage many times per second, according to the task at hand. Their embedded processors from the AMD-K6-2E+ and AMD-K6-IIIE+ families are all implementing PowerNow!. According to [3], AMD PowerNow! is able to support 32 different core voltage settings ranging from 0.925 V to 2.00 V with voltage steps of 25 mV or 50 mV. Clock frequency can change in steps of 33 MHz or 50 MHz, from an absolute low of 133 MHz or 200 MHz, respectively. The voltage and frequency changes are controlled through a special block, the Enhanced Power Management (EPM) block. At a speed change, an EPM timer ensures stable voltage and PLL frequency, operation which can take at most 200 $\mu$s. During this

time, instruction processing stops. A comparison with a Pentium III 600+ using Intel SpeedStep shows that the AMD's processor with PowerNow! consumes around 50% less power than the Pentium with SpeedStep (3W for AMD-K6-2E+ vs. 7W for Pentium III 600+).

**Intel's XScale** Intel has recently come out with XScale, an ARM core based architecture that supports on-the-fly clock frequency and supply voltage changes [4]. The frequency can be changed directly, by writing values in a register, while the voltage has to be provided from and controlled via an off-chip source. The XScale core specification allows 16 different clock settings, and four different power modes (one ACTIVE and three other). The actual meaning of these settings are dependent on the Application Specific Standard Product (ASSP). For instance, the 80200 processor supports clock frequencies up to 733 MHz, adjustable in steps of 33-66 MHz. The core voltage can vary between 0.95 V and 1.55 V. Switching between speeds takes around 30 $\mu$s. The PXA250 Processor provides 20 different clock and voltage settings (from 100MHz/0.85V to 400MHz/1.3V). Each setting specifies the clock frequencies of bus and memory. Voltage transition time is around 500 $\mu$s.

**IBM's PowerPC 405LP** The 405LP provides the clock and voltage ranging from 152MHz@1.0V to 380MHz@1.8V. The 405LP implements many features to improve power efficiency when the SoC is active. Under software control, both the voltage and the frequency of the processor can be modified, thereby allowing the performance demands of the application to be met while minimizing the dynamic power consumption. Unused storage

and functions are not clocked, eliminating unnecessary energy consumption. In addition, the 405LP implements standby power reduction features to ensure that power is not wasted when the SoC is inactive. This processor, under software control, can enter both a low-leakage sleep state and a state-preserving deep-sleep state to minimize standby power consumption. By applying these techniques, the active performance of the 405LP is not sacrificed, while standby power can be reduced as low as 54 $\mu$W. It can provide the performance of 500 million instructions per second (MIPS) while running at 380 MHz on a 1.8-volt power supply and consuming half a watt, but the power drops to just over 50 mW when the clock frequency is dropped to 150 MHz and the voltage to 1 volt, while still providing the performance of over 200 MIPS [27].

**TI's TMS320C5510 DSK** The TMS320C5510 DSP Starter Kit (DSK) is a low-cost development platform designed to speed the development of power-efficient applications based on TI's TMS320C55x DSP generation. The kit provides new performance-enhancing features such as power management tools. They allow developers to evaluate system power and examine the frequency-voltage scaling feature for future revisions of the TMS320C5510 DSPs. Designers can use the Power Analyzer and Power Scaling Library to confidently tune their systems to maximize efficient power consumption in applications.

The Power Scaling Library (PSL) [25] is a software library that allows embedded systems programmers to manage both frequency and voltage scaling through an easy to use API. The PSL provides hardware abstraction, portability, and a standard API that can be used among different TI devices.

Included in the API are routines that initiate scaling operations, and various query routines that provide information on current settings and available frequency/voltage settings. Frequency changes are initiated directly by the user. Voltage changes are performed indirectly by the PSL when a frequency change occurs. Specifically, the PSL will automatically scale the voltage to the minimum level required by a frequency. Since voltage changes are only initiated indirectly, the PSL can ensure a legal frequency/voltage setting at all times.

The PSL provides the following functionality:

- Scaling operation to scale frequency and voltage.
- Query operations that return current frequency and voltage settings.
- Query operations that return available frequencies settings and the required voltage settings for those frequencies.
- Query operation that returns the latencies associated with a scaling operation.
- Callbacks to user code before and after scaling operations. These callbacks will enable users to perform any necessary peripheral modifications that may be required as a result of the upcoming/just completed scaling operation.

The Figure 8.1 shows the power savings that were obtained when scaling only the frequency, and the power savings that were obtained when scaling both the frequency and the voltage. In this example, the power savings achieved by lowering the frequency from 200 MHz to 72 MHz was 62 percent.

Additional savings were realized when the voltage was also lowered from 1.6 V to 1.1 V. In this case, the overall power savings were 77 percent.

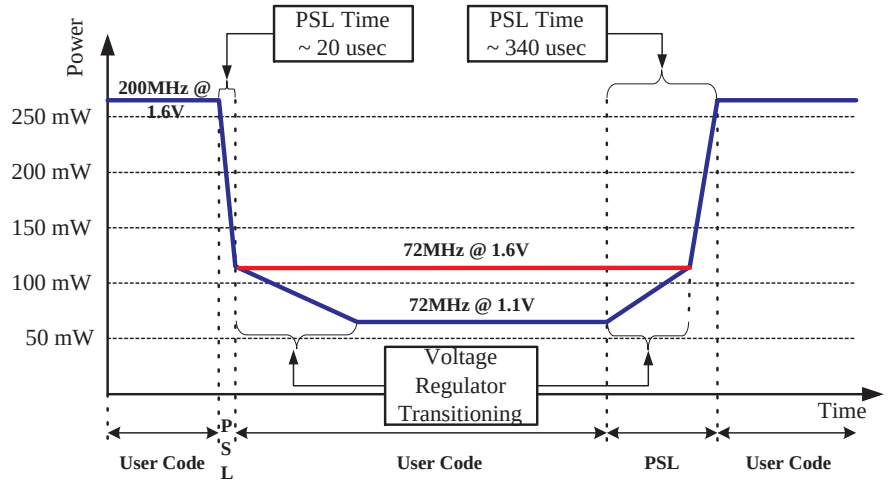



Figure 8.1: Dynamic voltage and frequency scaling traces.

The graph also shows the execution flow of a scaling operation and the amount of time spent in the PSL. In the case where both the frequency and voltage are scaled, the user code calls into the PSL to lower the frequency from 200 MHz to 72 MHz. The PSL lowers the frequency to 72 MHz and automatically lowers the voltage to 1.1 V, which is the lowest voltage required for 72 MHz. We can see that the PSL waits for the new frequency to be reached, but not the voltage. The time spent in the PSL in this case was $\sim$20 $\mu$s, which is the amount of time it takes the PLL to lock to the new frequency. Looking at the second call to the PSL, the user code calls into the PSL to scale the frequency from 72 MHz to 200 MHz. First, the PSL will automatically increase the voltage to 1.6 V, and then increases the frequency to 200 MHz. The PSL does not return until both the voltage and frequency have been increased. In this case, the PSL must wait for

the voltage increase to complete because it must be increased before the frequency is increased. The time spent in the PSL was ∼340 $\mu$s (∼300 for the voltage increase, and ∼40 for the PLL to lock to the new frequency).

**Compaq's Itsy Pocket Computer** The Itsy pocket computer is a research platform developed at Compaq's Western Research Laboratory. Its aim is to enable hardware and software research in pocket computing, including low-power hardware, power management, operating systems, wireless networking, user interfaces, and applications. The platform is equipped with a StrongARM SA-1100 processor as a main processor. The SA-1100 processor uses the phase-locked loop (PLL), allowing to change the CPU core frequency to one of 11 levels between 59.0 MHz and 226.4 MHz. Furthermore, Itsy version 2.6 has a programmable core voltage regulator; supply voltage can scale to one of 30 levels between 1.0 V and 2.0 V. To change the clock and voltage level, there is a overhead time during change. The overhead time is different depending on the current and target value of clock level, and is 189 $\mu$s at maximum. Itsy runs the Linux operating system (ver. 2.0.30) with a kernel support for dynamic voltage scaling. Applications can access the DVS function by the ioctl system call to the '/dev/clkspeed' device file.

# B. Automatic Voltage Scaler

Fig. 8.2 shows the overall structure of Automatic Voltage Scaler (AVS). It imports a high-level language program (such as source codes) and its timing requirements (such as a deadline), and converts them into DVS-

aware program. The converted program satisfies the same functional and temporal requirements of $P$, but it consumes much less energy than $P$. The AVS consists of four main modules, i.e. *Compiler*, *Timing Analyzer*, *VSE Selector*, and *Code Transformer*.

The *Compiler* surveys the input program structure and generates the inputs for the timing analyzer, i.e., syntax tree, call graph and assembly code.

The *Timing Analyzer* analyzes the timing behavior of the program, estimates the predicted remaining execution cycles of all the basic blocks in the input program, and transfers the results to the VSE selector. It is responsible for estimating $C_{RWEC}(b_i)$'s or $C_{RAEC}(b_i)$'s of all the basic blocks in an input program. In order to estimate $C_{RWEC}(b_i)$ or $C_{RAEC}(b_i)$ of a given basic block $b_i$, AVS uses a modified version of a timing tool developed by Lim *et al.* [21]. Lim *et al.*'s original timing tool estimates the WCET of a whole program traversing the program's syntax tree. Since AVS needs the RWEC from each basic block, we have modified the original timing tool accordingly to our purpose.

The *VSE Selector* is responsible for determining the locations of VSEs. For this work, it requires the processor's scaling information, i.e., voltage scaling overhead. The VSE selector modifies the syntax tree by inserting VSEs. Since the timing behavior of a target program can be changed by the inserted VSEs, the VSE selector transfers the information about VSEs to the timing analyzer. The timing analyzer reexamines the timing behavior and the informs the result to the VSE selector. This loop iterates until there is no change on the selected VSEs. This process is described in Figure 4.5.

If we use the look-ahead IntraDVS, the selected VSEs are post-processed by the *Look-ahead VSE Analyzer*. The look-ahead VSE analyzer uses the data dependency graph generated by compiler.

The *Code Transformer* generates a converted program with the informations about the selected voltage scaling locations and its speed update ratios. More detailed codes for VSE are described at Chapter 3. The interfaces between modules are defined to easily substitute each module with another one implemented by other algorithm.

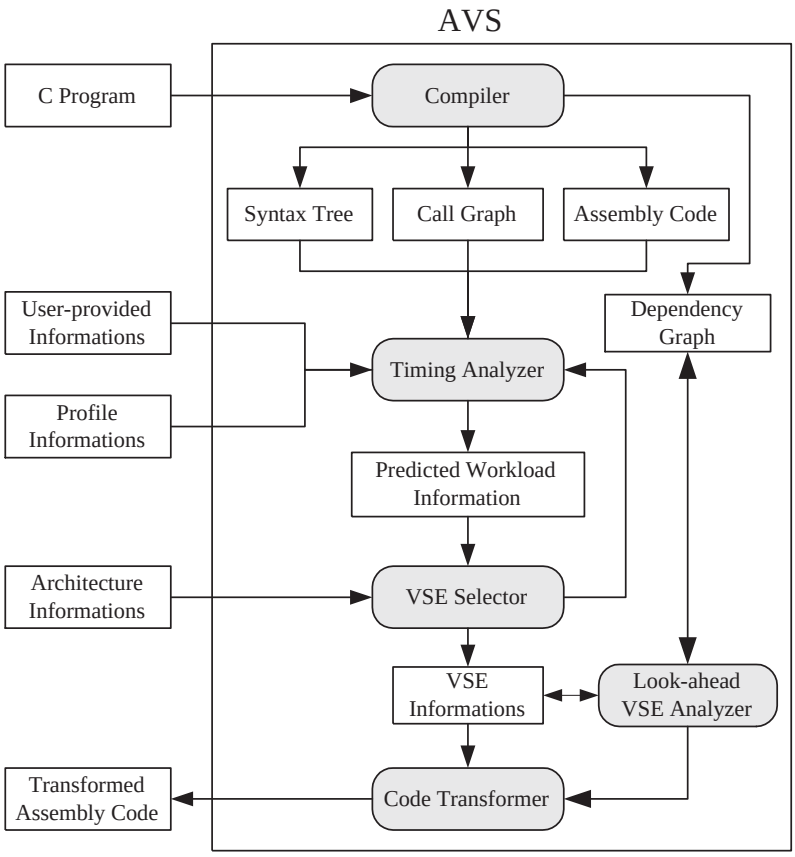



Figure 8.2: Automatic Voltage Scaler.