Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis[*]

Dongkun Shin
School of Computer Science and Engineering
Seoul National University
sdk@davinci.snu.ac.kr

Jihong Kim
School of Computer Science and Engineering
Seoul National University
jihong@davinci.snu.ac.kr

Seongsoo Lee
Department of Information Electronics
Ewha Woman's University
sslee1@mm.ewha.ac.kr

ABSTRACT

We propose an intra-task voltage scheduling algorithm for low-energy hard real-time applications. Based on a static timing analysis technique, the proposed algorithm controls the supply voltage within an individual task boundary. By fully exploiting all the slack times, a scheduled program by the proposed algorithm always complete its execution near the deadline, thus achieving a high energy reduction ratio. In order to validate the effectiveness of the proposed algorithm, we built a software tool that automatically converts a DVS-unaware program into an equivalent low-energy program. Experimental results show that the low-energy version of an MPEG-4 encoder/decoder (converted by the software tool) consumes less than 7∼25% of the original program running on a fixed-voltage system with a power-down mode.

1. INTRODUCTION

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.

When the required performance of a given task is lower than the maximum performance of a VLSI system, the clock speed and its corresponding supply voltage can be dynamically lowered to the lowest possible level while meeting the task's deadline constraint. This is the key idea of dynamic voltage scaling (DVS) technique. For example, consider a task with a deadline of 25msec, running on a processor with the 50MHz clock speed and 5.0V supply voltage. If the task requires 5×10^5 cycles for its execution, the processor executes the given task in 10msec, and becomes idle for the remaining 15msec. However, if the clock speed and supply voltage are lowered to 20MHz and 2.0V, it finishes the given task just at its deadline (=25msec), resulting in 84% energy reduction.

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 (so that the energy consumption is minimized) while not violating the timing requirements. Recently, several researchers have investigated the DVS problem [12, 4, 9, 11, 7]. Most research effort focused on real-time systems with multiple tasks. Given multiple tasks, the key question is how to assign the proper speed to each task dynamically while guaranteeing all their deadlines. Note that 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 voltage scheduling*.

However, inter-task voltage scheduling has several practical limitations. For example, it requires OS modifications, since a task scheduler in OS determines the supply voltage of a task. Furthermore, it cannot be applied to a single-task environment, since the supply voltage is determined as a constant value for a given task. This implies that for many practical small-sized embedded mobile applications, DVS cannot be used.

Even in a multi-task environment, inter-task voltage scheduling 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, consider a typical mobile videophone application with four tasks shown in Table 1. Using an inter-task voltage scheduling of [11], only 17% of energy reduction is observed while an off-line (theoretical) optimal voltage scheduling can achieve 90% power reduction.

These limitations lead to the idea of *intra-task voltage scheduling*, where a given task is partitioned into several segments, and different supply voltages are assigned for each segment. Lee and Sakurai [6] proposed an intra-task voltage scheduling where each task is partitioned into fixed-length timeslots. Although [6] 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 se-

[*]This work is supported by the Ministry of Information & Communication of Korea (Support Project of University foundation research<'00> supervised by IITA).

	MPEG-4 Video Encoding	MPEG-4 Video Decoding	VSELP Speech Encoding	VSELP Speech Decoding
Period (= Deadline) (msec)	66.667	66.667	40.000	40.000
Average Slack Time (msec)	37.287	8.366	0.937	0.703
Average Execution Time (msec)	13.099	1.460	0.907	0.680
NEC(Inter)[a]	0.826			
NEC(Ideal)[b]	0.106			

[a]Normalized energy consumption by an inter-task voltage scheduling [11].
[b]Normalized energy consumption by an off-line optimal voltage scheduling.
*The base case of normalization is DVS-unaware systems using only power-down mode.

Table 1: A typical videophone application.

lecting the best program locations where voltage scaling code is inserted. Since average programmers are generally not familiar with low-energy software issues as well as timing analysis techniques, it is in practice very difficult to use intra-task voltage scheduling for real-time applications without support for any systematic programming methodology.

In this paper, we propose a novel intra-task voltage scheduling algorithm based on a static timing analysis of a target application program. The proposed algorithm has the following features: It fully exploits *all* slack times coming from execution time variations within a single task, achieving a significant improvement in the energy consumption. Since it does control the supply voltage *within* each task, it is applicable to a single-task environment. It provides a systematic methodology for developing an *automatic* conversion tool that converts DVS-unaware programs into DVS-aware ones. This means that a programmer of the original program requires no knowledge on DVS, making the proposed algorithm very practical. 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.

The rest of this paper is organized as follows. Basic notations and the target variable voltage processor model are described in Section 2. In Section 3, the proposed intra-task voltage scheduling is presented. Software framework and simulation results are discussed in Section 4. Section 5 concludes with a summary and directions for future work.

2. PRELIMINARIES

2.1 Program Representation

We consider a real-time task τ with the deadline D_τ[1]. The task τ is represented by its control flow graph (CFG) G_τ. If the task τ has N basic blocks, $b_1, b_2, \cdots, b_N$, G_τ consists of N nodes. (We assume that b_1 is the entry basic block of the task τ.) 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 the following three entries: $C_{EC}(b_i)$, $C_{RWEC}(b_i)$, and $S(b_i)$. $C_{EC}(b_i)$ denotes the number of clock cycles needed to execute b_i. $C_{RWEC}(b_i)$ represents the remaining worst case execution cycles (RWEC) among all the execution paths that start from b_i. $S(b_i)$ represents the processor clock speed in frequency at which b_i is executed. Note that, in the BBI definition, we did use execution cycles instead of execution times. This is because, as we adjust the clock speed on a variable voltage processor, the execution time is changing 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 with the number of execution cycles.

[1]Since our discussions are limited to a single or dominant task τ as explained Section 1, for the description purpose, we omit the subscript τ from our notations.

We also define similar notations for execution paths. p_i denotes an execution path of a task τ. p_i can be expressed as a sequence of basic blocks. The worst case execution path is denoted as p_{worst}. $C_{EC}(p_i)$ represents the number of execution cycles when p_i is executed. The number of execution cycles of p_{worst} is denoted as C_{WCEC}.

2.2 Variable Voltage Processor Model

Throughout this paper, we make the following assumptions on a target variable voltage processor: (1) The processor provides a special instruction, `change_f_V(`f_{CLK}`)`, that can dynamically control clock frequency f_{CLK} and its corresponding voltage V_{DD} of the processor. (2) f_{CLK} and V_{DD} can be set continuously 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[2]. (4) During clock/voltage transition, the processor stops running (denoted as Processor Type I) or runs at f_{CLK}=min(f_{CLK1}, f_{CLK2}) (denoted as Processor Type II). Assumptions (1)-(3) are valid for most existing variable voltage processors [1, 2, 6]. Recent frequency synthesizers and DC-DC converters achieve clock/voltage transition time of less than 200μsec, corresponding to 20,000 cycles at 100MHz. Assumption (4) specifies the target processor's execution status during clock/voltage transition. Processor Type depends on its hardware architecture. For example, the DVS architecture appropriate for off-the-shelf processors [6] belongs to Type I while Transmeta's Crusoe processor [2] is of Type II. Our work described in this paper can support both processor types with a slight modification of clock/voltage transition overhead modeling.

3. INTRA-TASK VOLTAGE SCHEDULING

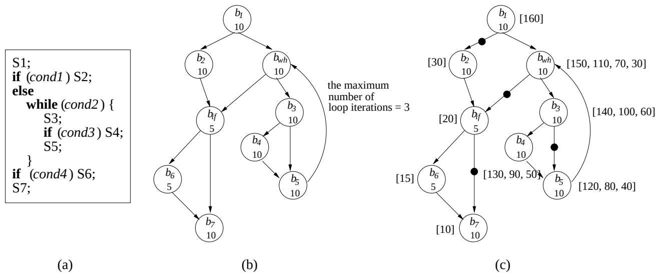
3.1 Basic Idea

Consider a hard real-time program P with the deadline of 2μsec shown in Figure 1(a). The CFG G_P of the program P is shown in Figure 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 required timing constraints are met. Such WCETs can be predicted by existing WCET analysis tools that produce safe and accurate WCET prediction results [3, 8]. 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,

[2]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.

b_4,b_5,b_{wh},b_{if},b_6,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, 160 cycles, which is the worst case execution cycles (WCEC). If a target processor operates at the maximal clock frequency of 80MHz, the program P completes its execution in 2μsec, resulting in no slack time.



```
S1;
if (cond1) S2;
else
   while (cond2) {
       S3;
       if (cond3) S4;
       S5;
   }
if (cond4) S6;
S7;
```

Figure 1: An example program P; (a) an example real-time program with the 2μsec deadline, (b) its CFG representation G_P, and (c) an augmented CFG G_P^A with $C_{RWEC}(b_i)$ values.

Intra-task voltage scheduling is based on a simple observation that there are large execution time variations among different execution paths. In particular, it exploits the fact that the average case execution paths (ACEPs) complete their executions much earlier than the WCEP(s) does [11]. For the example program shown in Figure 1(b), there exist 32 different execution paths. While the WCEP p_{worst} takes 160 cycles, eight of 32 possible execution paths take less than 80 cycles. For such short execution paths, 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 $p_1 = (b_1, b_2, b_{if}, b_6, b_7)$ of Figure 1(b) whose execution takes 40 cycles. In the ideal case, when we can perfectly predict *in advance* that the actual execution path will be p_1, we can start the execution with the clock speed of 20MHz without violating the 2μsec deadline. Although this will improve the energy efficiency significantly, we cannot start with the 20MHz clock speed from b_1 because we do not generally know in advance which execution path will be taken by the next program execution.

In the proposed intra-task voltage scheduling technique, we take an adaptive approach with the help of a static program analysis technique on worst case execution times. Using a modified WCET analysis tool such as one in [8], for each basic block b_i, we compute $C_{RWEC}(b_i)$ in *compile time*. For example, Figure 1(c) shows an augmented CFG G_P^A with $C_{RWEC}(b_i)$ values. The G_P^A graph is statically constructed by a modified WCET tool. 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. Once G_P^A is constructed, we can statically identify branching edges (of a CFG G) that drops the remaining worst case execution cycles *faster* than the current execution rate. For example, in Figure 1(c), we can identify four such edges, i.e., (b_1, b_2), (b_{wh}, b_{if}), (b_{if}, b_7) and (b_3, b_5). In Figure 1(c), these edges are marked by the symbol $\bullet$. When the thread of execution control branches to the next basic block through one of these edges, say (b_1, b_2), the clock speed can be lowered because the remaining work is reduced by the difference between $C_{RWEC}(b_{wh})$ and $C_{RWEC}(b_2)$. By reducing the clock speed so that the $C_{RWEC}(b_2)$ cycles can be completed exactly at the deadline, the proposed technique always meets the required timing constraint. Since the voltage scaling decisions are made in compile time, not run 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).

Figure 2 compares how the speed and voltage change depending on whether the intra-task voltage scheduling is used or not. Assuming that no energy is consumed in an idle state and $E \propto C_L \cdot N_{cycle} \cdot V_{DD}^2$, when the execution follows the path $p_1 = (b_1, b_2, b_{if}, b_6, b_7)$, the energy consumption ratio of Figure 2(b) to Figure 2(a) is 0.31. With the intra-task voltage scheduling, the energy consumption is reduced by 69%.

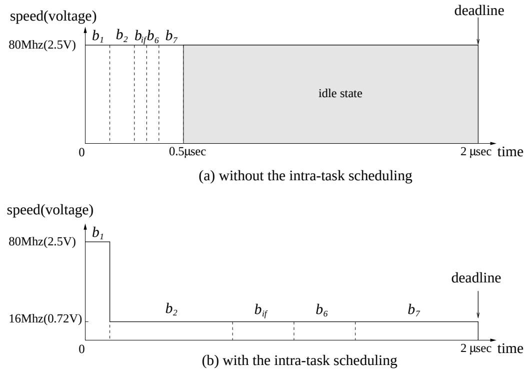


Figure 2: Speed and voltage changes by the intra-task scheduling.

3.2 Speed Assignment Algorithm

The speed assignment algorithm assigns to each basic block a proper speed at which the basic block is executed. For a hard real-time task, the goal of the speed assignment algorithm is to assign the speed to each basic block so that the energy consumption is minimized while the timing requirements are satisfied.

If the actual execution path p_{act} of a task τ were known in advance, the optimal execution speed could be easily computed as shown in [5]: for each basic block b_i in p_{act}, $S(b_i) = C_{EC}(p_{act})/D$. However, since the exact execution path is generally unknown until the program execution is completed, we adjust $S(b_i)$ based on the remaining worst case execution cycles $C_{RWEC}(b_i)$. By a modified version of the static WCET prediction algorithm such as one in [8], we can estimate $C_{RWEC}(b_i)$ for each basic block b_i. $S(b_i)$ is set to the clock speed S at which the remaining $C_{RWEC}(b_i)$ cycles can be completed exactly at the deadline.

At the entry basic block b_1, $C_{RWEC}(b_1)$ is set to C_{WCEC} and the starting speed is set to C_{WCEC}/D. If we denote $C_{RWEC}(t)$ to indicate the remaining worst case execution cycles at time t, as the execution proceeds, $C_{RWEC}(t)$ is linearly decreased at the rate of clock speed when 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 a basic block b_j not in p_{worst}, $C_{RWEC}(t)$ drops after the execution of b_i is completed by the difference between $C_{RWEC}(b_i) - C_{EC}(b_i)$ and $C_{RWEC}(b_j)$.

Figure 3 shows how $C_{RWEC}(t)$ dynamically changes as the path $p = (b_1, b_2, b_{if}, b_7)$ of the example program P shown in Figure 1 is executed. In Figure 3(a) which uses no speed scheduling, $C_{RWEC}(t)$ drops at two points, $C_{EC}(b_1)$/80MHz and $(C_{EC}(b_1) + C_{EC}(b_2) + C_{EC}(b_{if}))$/80MHz. Since no speed scheduling is used at Figure 3(a), $C_{RWEC}(t)$ is decreased at the rate of 80MHz, resulting in a

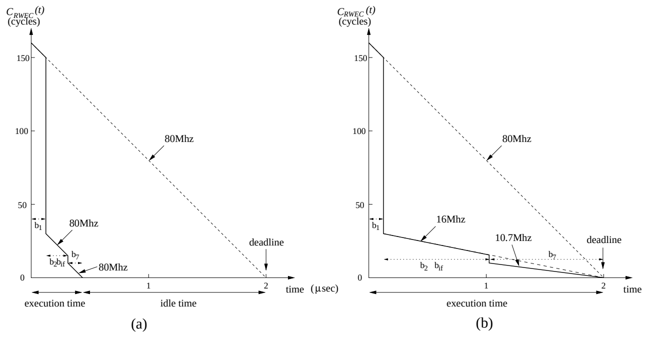


Figure 3: The changes of $C_{RWEC}(t)$ over different speed scaling algorithms: (a) no intra-task scheduling and (b) RWEC-based intra-task scheduling.

slack time interval of 1.5625μsec. Figure 3(b) shows the effect of speed scheduling for the same execution path. Since $C_{RWEC}(t)$ drops right after the execution of b_1 is completed, speed is changed from 80MHz to 16MHz, which is the minimum speed at which the processor can complete the remaining program execution before the deadline. When $C_{RWEC}(t)$ drops right after b_{if}, speed is also changed due to the same reason.

Since the proposed RWEC-based intra-task scheduling makes all the execution paths to complete their executions near the required deadline, the RWEC-based technique provides two benefits: 1) There is little slack times, thus it is energy efficient and 2) it is guaranteed that the scheduled program always meets the timing constraint. We call the points at which $C_{RWEC}(t)$ is dropped vertically in Figure 3, Voltage Scaling Edges (VSEs), because the speed and voltage can be scaled at these points. We denote the number of cycles reduced at VSEs as C_{saved}.

3.2.1 B-type Voltage Scaling Edges

VSEs can be classified into two categories, i.e., B-type VSEs and L-type VSEs. 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 $\frac{C_{RWEC}(b_{then})}{C_{RWEC}(b_{else})}$. We call this ratio a *speed update ratio* and represent it by $r(b_{cond} \rightarrow b_{then})$.

In adjusting speed/voltage at VSEs, several instructions are (other than `change_f_V(`f_{CLK}`)`) are required. We denote the number of cycles needed to execute these 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 speed update ratio $r(b_i \rightarrow b_j)$ for a B-type VSE (b_i, b_j) is calculated as follows:

$$r(b_i \rightarrow b_j) = \frac{C_{RWEC}(b_j)}{C_{RWEC}(succ_{worst}(b_i)) - C_{overhead_B}} \tag{1}$$

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.

Though the proposed scheduling can support both types (i.e., Type I and Type II in Section 2.2) of variable voltage processors, we assume that the processor stops while voltage is being scaled for a simpler description. If $C_{RWEC}(b_j) \geq C_{RWEC}(succ_{worst}(b_i)) - C_{overhead_B}$, that is $r(b_i \rightarrow b_j) \geq 1$, the edge (b_i, b_j) is not selected as a VSE. For a VSE between b_i and b_j, a speed update ratio $r(b_i \rightarrow b_j)$ is multiplied to the current speed before b_j starts its execution. For example, assuming $C_{overhead_B}$ as 0, $S(b_2)$ in Figure 1 is updated as follows:

$$S(b_2) = S(b_1) \cdot r(b_1 \rightarrow b_2) = S(b_1) \cdot \frac{30}{150}$$

So the clock speed is changed from 80MHz to 16MHz (=80MHz $\times \frac{30}{150}$).

3.2.2 L-type Voltage Scaling Edges

Although C_{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 exists 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. By the L-type scaling, the number of saved cycles C_{saved} for a loop l is given by

$$C_{saved}(l) = C_{WCEC}(l) \cdot (N_{worst}(l) - N_{exec}(l)) \tag{2}$$

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. For the L-type scaling, consider the edge (b_{wh}, b_{if}) in Figure 1 which is an example L-type VSE. Assuming $N_{exec}(l)$ = 1, and $C_{overhead_L} = 0$, $S(b_{if})$ is updated as follows:

$$\begin{aligned} S(b_{if}) &= S(b_{wh}) \cdot \frac{C_{RWEC}(b_{if})}{C_{RWEC}(b_{if}) + C_{saved}(l) - C_{overhead_L}} \qquad (3) \\ &= S(b_{wh}) \cdot \frac{20}{20 + 40 \cdot (3-1)} \\ &= S(b_{wh}) \cdot r(b_{wh} \rightarrow b_{if}) \end{aligned}$$

When $S(b_{wh})$ is 80MHz, $S(b_{if})$ is reduced to 16MHz before executing b_{if}.

Unlike a B-type VSE, calculating the speed update ratio for an L-type VSE requires the run-time information such as $N_{exec}(l)$[3]. The speed update ratio 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 speed update ratio 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 if $N_{exec}(l) = N_{worst}(l)$ or not. Among the L-type candidates, we choose the final L-type VSEs by the algorithm explained in Section 3.3. Although L-type VSEs are more complex than B-type VSEs, since slack times from loop executions are generally much larger than those from conditional statements, the contribution of L-type VSEs on the overall energy reduction is bigger.

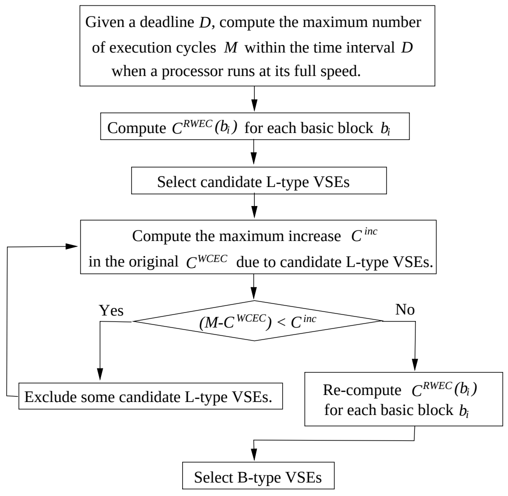
3.3 VSE Selection Algorithm

While voltage scaling code for B-type VSEs does not increase C_{WCEC} of a given program, voltage scaling code for L-type VSEs

[3]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.

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 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.

In order to avoid this situation, we select the final L-type VSEs from the candidate L-type VSEs by the algorithm shown in Figure 4. Assuming that the target processor can execute M cycles (at its full speed) within the given the deadline interval, we check if candidate L-type VSEs (if all selected) will violate the required timing constraint with the extra cycles added by the L-type VSEs. If the deadline may not be satisfied in the worst case, we exclude some candidate L-type VSEs until the increase in C_{WCEC} will satisfy the given deadline. After L-type VSEs are decided, $C_{RWEC}(b_i)$'s are recomputed and B-type VSEs are selected.



Given a deadline D, compute the maximum number of execution cycles M within the time interval D when a processor runs at its full speed.
Compute $C^{RWEC}(b_i)$ for each basic block b_i
Select candidate L-type VSEs
Compute the maximum increase C^{inc} in the original C^{WCEC} due to candidate L-type VSEs.
Yes
$(M-C^{WCEC}) < C^{inc}$
No
Exclude some candidate L-type VSEs.
Re-compute $C^{RWEC}(b_i)$ for each basic block b_i
Select B-type VSEs

Figure 4: Overall VSE selection algorithm.

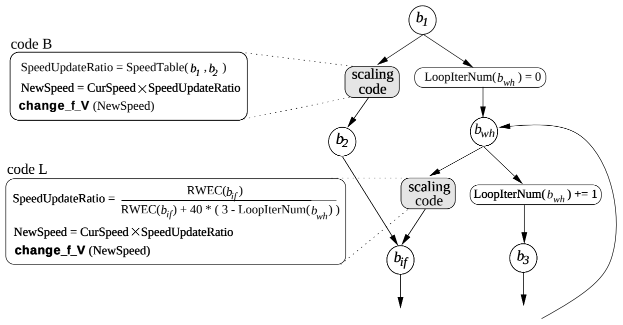
After VSEs are selected, the voltage scaling code should be inserted into a program code. Figure 5 shows an example of code generated for VSEs. Since the edge (b_1, b_2) is a B-type VSE, B-type scaling code, code B, is inserted into (b_1, b_2) as shown in Figure 5. To change speed and voltage, the speed update ratio of the corresponding VSE is read from the speed table which is constructed at compile time using Equation (1). The current speed value is multiplied by the speed update ratio and its result is set to the new speed to use. The special instruction `change_f_V` (that changes the speed and voltage of the target processor) is called with the new speed value.

Figure 5 also shows an example voltage scaling code, code L, for an L-type VSE. Because the L-type VSE requires the loop iteration number when the loop exits, extra code is necessary to maintain the current loop iteration number. In Figure 5, two boxes in the edges (b_1, b_{wh}) and (b_{wh}, b_3) include this extra code.

4. EXPERIMENTAL RESULTS

4.1 Software Framework

We have developed a software tool, the Automatic Voltage Scaler (AVS), that *automates* the development of hard real-time programs on a variable voltage processor using the intra-task scheduling al-



b_1
code B
SpeedUpdateRatio = SpeedTable(b_1, b_2)
NewSpeed = CurSpeed × SpeedUpdateRatio
change_f_V (NewSpeed)
scaling code
LoopIterNum(b_{wh}) = 0
b_2
b_{wh}
code L
SpeedUpdateRatio = $\frac{\text{RWEC}(b_{if})}{\text{RWEC}(b_{if}) + 40 * (3 - \text{LoopIterNum}(b_{wh}))}$
NewSpeed = CurSpeed × SpeedUpdateRatio
change_f_V (NewSpeed)
scaling code
LoopIterNum(b_{wh}) += 1
b_{if}
b_3

Figure 5: Code generation for VSEs.

gorithm described in section 3. AVS takes as an input a *DVS-unaware* (thus regular) program P and its timing requirements, and produces a *DVS-aware* low-energy program P_{DVS} that satisfies the same timing requirements of P. The converted program P_{DVS} contains voltage scaling code that handles all the idiosyncrasy of scaling speed/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. In the current version of AVS, the MIPS R3000 instruction set architecture was used as a target processor.

The organization of AVS can be divided into two main modules, i.e., the WCET Predictor (WP) module and the Voltage Scaler (VS) module. The WP module is responsible for estimating $C_{RWEC}(b_i)$'s of all the basic blocks in an input program. In order to estimate $C_{RWEC}(b_i)$ of a given basic block b_i, AVS uses a modified version of a timing tool developed by Lim *et al.* [8]. 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 WP module, like the original timing tool [8], takes as an input a high-level language program and the user-provided information (e.g., loop bound) to estimate $C_{RWEC}(b_i)$'s.

The VS module identifies the VSEs based on $C_{RWEC}(b_i)$'s with the program syntax tree, assigns proper speeds to these edges, and generates a converted program.

4.2 Simulation Results

To evaluate the power reduction performance of AVS, we have experimented with an MPEG-4 video encoder and decoder. Since we don't have proper hardware platform with a variable voltage processor, we developed an energy simulator for the experiment. The energy simulator takes an assembly program and its execution trace as inputs, and calculates total energy consumption of the program execution. In this simulation, 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 $f_{CLK} = 1/T_D \propto (V_{DD} - V_T)^{\alpha}/V_{DD}$ [10] where V_{DD}, V_T, and α are assumed to be 2.5V, 0.5V, and 1.3, respectively. Clock/voltage transition overhead C_{VTO} is assumed to be 0∼20,000 cycles, corresponding to 0∼200μsec of transition time with 100MHz clock frequency. For non-zero C_{VTO} values, the processor stops its execution and enters into a power-down mode during clock/voltage transition (Processor Type I in Section 2.2).

Figures 6(a) and 6(b) show the energy consumption of the AVS-converted MPEG-4 encoder and decoder program. (In converting the MPEG-4 encoder and decoder program, it took less than 100msec for AVS.) Results were normalized over the energy consumption of the original program running on a DVS-unaware system. For each program, we experimented assuming that a power-down mode consumes 5% of the energy consumed by a normal mode [11]. The AVS-converted MPEG-4 encoder and decoder programs consume less than 25% and 7% of the original program, respectively. The large difference of energy efficiencies between two programs is due to the different timing behaviors of the programs. There is a large difference between WCET and ACET (average case execution time) of the MPEG-4 decoder while WCET of the MPEG-4 encoder is relatively close to ACET. Figure 6(c) and 6(d) shows 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 (=30μsec) in the MPEG-4 encoder, the number of voltage transitions decreases sharply, and energy consumption increase rapidly. When $C_{VTO} > 5{,}000$ cycles (=50μsec) in both the MPEG-4 encoder and decoder, the energy consumption does not increase rapidly. This is because the number of discarded voltage scaling edges (due to clock/voltage transition overhead) is small.

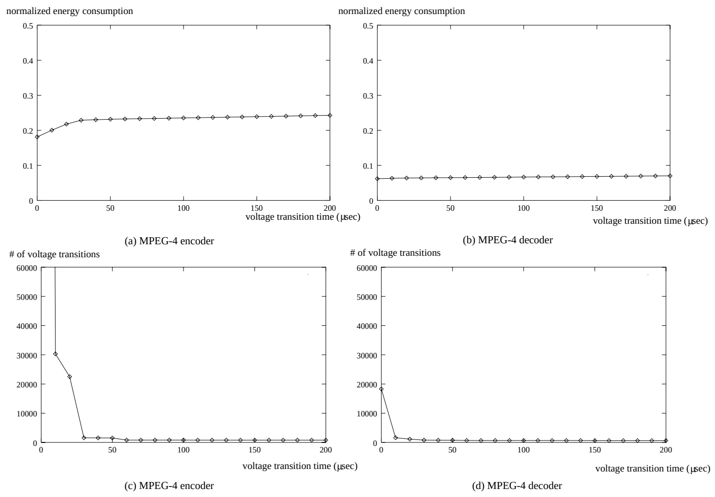


Figure 6: The normalized energy consumption and the number of voltage transitions of the AVS-converted MPEG-4 encoder and decoder program.

The number of VSEs, which represents how many copies of voltage scaling code were inserted into the AVS-converted program, indicates the degree of code size increment by inserting voltage scaling code using an in-line expansion. For the AVS-converted MPEG-4 encoder and decoder, 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 are responsible for quite a large portion of the total power reduction.

5. CONCLUSION

In this paper, we have proposed an intra-task voltage scheduling algorithm for low-energy hard real-time applications. Based on the RWEC information of each basic block, which is computed statically in compile time using a variant of the WCET analysis technique, 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 is scaled down. 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.

We have built an automatic voltage scaling tool, AVS, based on the proposed intra-task voltage scaling algorithm. AVS automatically transforms a DVS-unaware program to a DVS-aware low-energy program with the same functional behavior and timing requirement. The experimental results using an MPEG-4 video encoder and decoder show that AVS improves the energy efficiency of the programs by a factor of 4$\sim$14 over the programs running on a non-DVS system with a power-down mode.

The intra-task voltage scheduling algorithm described in this paper can be extended in several directions. For example, we have used RWECs in selecting VSEs, but different measures such as ACETs may be more effective in reducing the energy consumption. We are currently investigating a scheduling technique based on ACETs that guarantees the required timing constraint. In addition, the energy efficiency of a scheduling technique may be improved significantly by utilizing more run-time information (e.g., the elapsed time). We plan to integrate different run-time information to proposed scheduling algorithms so that the energy efficiency can be further improved.

6. REFERENCES

[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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Y. Lee and C. M. Krishna. Voltage-clock scaling for low energy consumption in real-time embedded systems. In *Proc. of the Sixth International Conference on Real-Time Computing Systems and Applications*, pages 272–279, 1999.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.