Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications

Dongkun Shin

Jihong Kim
Seoul National University

Seongsoo Lee
Ewha Woman's University

A novel intra-task voltage-scheduling algorithm controls the supply voltage within an individual task boundary. By fully exploiting slack time, it achieves a high-energy reduction ratio. Using this algorithm, a software tool automatically converts an application into a low energy version.

IN MODERN VLSI SYSTEM DESIGN, power consumption is one of the most important design constraints. For battery-powered portable systems such as digital cellular phones, personal digital assistants, and mobile videophones, low power consumption is a primary design goal because the battery operation time is one of the most critical performance measures. Energy consumption E of CMOS circuits, which is dominated by total dynamic power consumption in most VLSI systems, is given by $E \propto C_L \times N_{cycle} \times V_{DD}{}^2$.[1] C_L is the load capacitance, N_{cycle} is the number of executed cycles, and V_{DD} is the supply voltage. Because energy consumption E has a quadratic dependency on supply voltage V_{DD}, lowering V_{DD} is the most effective way of reducing energy consumption. However, lowering the supply voltage also decreases the clock speed, because CMOS circuit delay T_D is given by $T_D \propto V_{DD}/(V_{DD} - V_T)^{\alpha}$,[2] where V_T is threshold voltage, and α is a velocity saturation index.

When a given task's required performance is lower than a VLSI system's maximum performance, the clock speed and its corresponding supply voltage can be dynamically controlled to the lowest possible level while meeting the task's deadline constraint. This is the key idea behind the dynamic voltage-scaling (DVS) technique.

For example, consider a task with a deadline of 25 ms, running on a 50-MHz processor with a 5.0-V supply voltage. If executing the task requires 5×10^5 cycles, the processor executes it in 10 ms and idles for the remaining 15 ms. However, if the clock speed and supply voltage are lowered to 20 MHz and 2.0 V, the processor finishes the given task just at the task's deadline (25 ms), resulting in an 84% energy reduction.

Recently, several research groups have investigated the DVS problem for hard real-time systems.[3-6] Most research focused on real-time systems with multiple tasks; in this case, the key question is how to assign the proper speed to each task dynamically while guaranteeing all task deadlines. These techniques exploit the *run-calculate-assign-run* strategy for the supply voltage determination. The steps in that strategy include

1. running the current task,
2. calculating the maximum allowable execution time for the next task,
3. assigning the supply voltage for the next task, and

0740-7475/01/$10.00 © 2001 IEEE

Table 1. Typical videophone application.

Characteristic	MPEG-4 video encoding	MPEG-4 video decoding	VSELP speech encoding	VSELP speech decoding
Period or deadline (s)	66.667	66.667	40.000	40.000
Worst-case execution time (s)	50.386	9.826	1.844	1.383
Average execution time (s)	13.099	1.460	0.907	0.680
Normalized energy consumption				
Inter-task voltage scheduling[9]		0.826		
Offline optimal voltage scheduling		0.106		

4. running the next task.

These techniques determine the supply voltage on a task-by-task basis, a strategy we call *inter-task* voltage scheduling.

While generally effective in reducing energy consumption of multitask real-time systems, inter-task voltage scheduling has several practical limitations. For example, because a task scheduler in an operating system determines a task's supply voltage, using inter-task voltage scheduling requires OS modifications. Furthermore, these techniques cannot be applied to a single-task environment, because the supply voltage is determined as a constant value for a given task. Because the single-task model is the basis for many small, embedded mobile applications, variable-voltage processors may be difficult to widely use in practice.

Even in a multi-task environment, inter-task voltage scheduling may not be effective in energy reduction if the execution time of one task dominates total execution time. For example, consider a typical videophone application with the four tasks shown in Table 1. In this application, the MPEG-4 video encoding task dominates execution time but has the lowest priority. For these reasons, inter-task voltage scheduling cannot take advantage of the slack time caused by the MPEG-4 video-encoding task; these algorithms are thus ineffective in reducing energy consumption.

For example, using the inter-task-scheduling algorithm of Shin and Choi[5] results in only a 17% energy reduction. In contrast, an offline (theoretical) optimal voltage-scheduling algorithm achieves about a 90% energy reduction.

We propose intra-task voltage scheduling—which adjusts the supply voltage within an individual task's boundary—as a solution to overcome the limitations of inter-task voltage scheduling. For example, a recent work by Lee et al.[7] demonstrates that dynamic voltage scaling within a single task boundary can significantly reduce energy consumption. Because intra-task voltage scheduling does not involve the OS in adjusting the clock speed, it has an advantage in that existing OSs can be used without modification on a variable-voltage processor.

However, at the current state of the art, it is fully a programmer's responsibility to apply intra-task voltage scheduling to applications. For example, there are no systematic guidelines for selecting the best program locations for inserting voltage-scaling code. Average programmers are generally not familiar with low-energy software issues and timing analysis techniques. In practice, therefore, it is difficult to use intra-task voltage scheduling for real-time applications without the support of a systematic programming methodology.

We propose a novel intra-task voltage-scheduling algorithm that can automate the development of DVS-aware, hard real-time programs on variable-voltage processors. It is based on static execution-time analysis techniques commonly used in developing hard real-time programs. Using these techniques, the proposed algorithm selects locations for inserting voltage-scaling code to reduce the overall energy consumption.

The proposed scheduling algorithm exploits *all* the slack time from runtime variations of different execution paths; there is no slack time when the scheduled program completes its execution, thus significantly improving energy efficiency. The novel aspect of our algorithm is

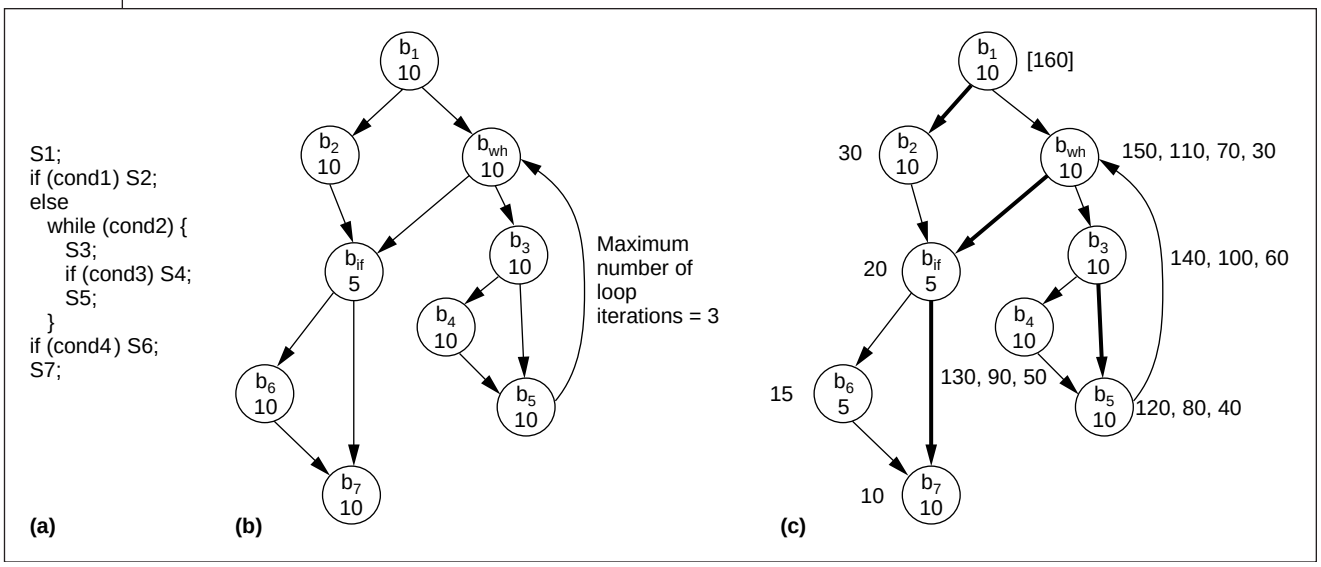

Figure 1. Example program P (a), a real-time program with a 2-µs deadline, has this CFG representation G_P (b) and an augmented CFG G_P^A with $C_{RWEC}(b_i)$ values (c).

that voltage-scaling decisions are made in compile time, although the voltage-scaling code itself can require some runtime information in determining an appropriate clock speed.

Furthermore, the proposed algorithm provides a systematic methodology for developing an automatic program conversion tool to convert DVS-unaware programs into DVS-aware ones. This means the original program's developers need no knowledge of DVS, making the proposed algorithm very practical.

Based on the proposed algorithm, we have developed a software tool called Automatic Voltage Scaler (AVS). It supports a fully automatic conversion of a DVS-unaware program P into a DVS-aware, low-energy program P_{DVS} that satisfies the same timing requirement as P.

Basic idea

Consider hard real-time program P, as shown in Figure 1a, with a 2 µs deadline. The control flow graph (CFG) G_P for program P is shown in Figure 1b. 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 for the basic block. The back edge from b_5 to b_{wh} models the while loop of program P.

In developing hard real-time systems where tasks have strict timing constraints (such as deadlines), the tasks' worst-case execution times (WCETs) are estimated in advance (before runtime) to guarantee that required timing constraints are met. Such WCETs can be predicted by existing WCET analysis tools, which produce safe and accurate WCET prediction results.[8,9]

Using a WCET analysis tool, we can find path $p_{worst} = (b_1, b_{wh}, b_3, b_4, b_5, b_{wh}, b_3, b_4, b_5, b_{wh}, b_3, b_4, b_5, b_{wh}, b_{if}, b_6, b_7)$ as the worst-case execution path (WCEP) for the example program P, assuming that the user sets the maximum number of while loop iterations to three. The predicted number of execution cycles in p_{worst} is 160 cycles, which is the number of worst-case execution cycles (WCEC) of program P.

If a target processor operates at the 80-MHz maximal clock frequency, program P completes its execution in 2 µs, resulting in no slack time. We used execution cycles instead of execution times because as we adjust the clock speed on a variable-voltage processor the execution time changes but the number of execution cycles remains constant.

Intra-task voltage scheduling is based on a simple observation: There are large execution time variations among different execution paths. In particular, this strategy exploits the fact that the average-case execution paths

(ACEPs) complete execution much earlier than the WCEP(s).[5]

The example program shown in Figure 1b has 32 different execution paths. While the WCEP p_{worst} takes 160 cycles, eight of 32 possible execution paths take less than 80 cycles. If we can identify such short execution paths in the early phase of execution, we can substantially lower the clock speed and significantly decrease energy consumption.

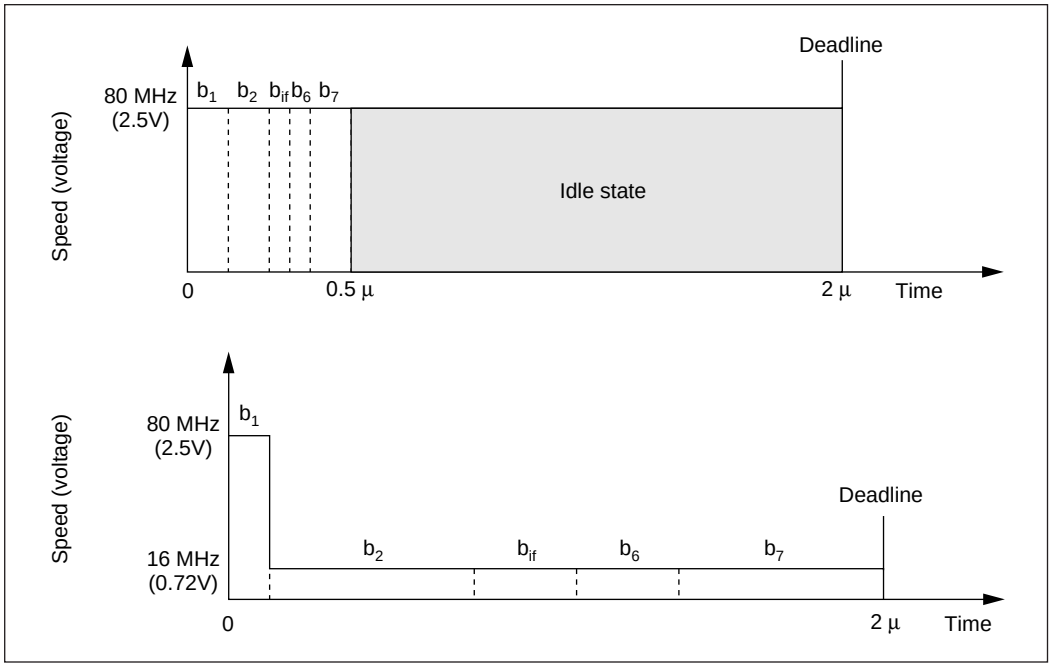

Figure 2. Speed and voltage changes without (a) and with (b) intra-task scheduling.

Consider the path $p_1 = (b_1, b_2, b_{if}, b_6, b_7)$ of Figure 1b; its 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 a clock speed of 20 MHz without violating the 2-μs deadline. Although this will significantly improve energy efficiency, we cannot start with the 20-MHz clock speed from b_1, because we do not generally know in advance which execution path the next program execution will take.

In intra-task voltage scheduling, we take the second best approach, with the help of a static program-analysis technique on worst-case execution times. Assume that $C_{RWEC}(b_i)$ represents the remaining worst-case execution cycles (RWEC) among all the execution paths that start from b_i. Using a modified WCET analysis tool, for each basic block b_i, we compute $C_{RWEC}(b_i)$ at compile time. For example, Figure 1c shows an augmented CFG G_P^A with $C_{RWEC}(b_i)$ values. A modified WCET tool statically constructs the graph G_P^A.

For the basic blocks (such as b_{wh}, b_3, b_4, b_5) related to the while loop, the corresponding nodes are associated with multiple $C_{RWEC}(b_i)$ values, reflecting the while loop's maximum three iterations. Once G_P^A is constructed, we can statically identify branching edges (of CFG G) that drop the remaining worst-case execution cycles faster than the current execution rate.

For example, in Figure 1c, we can identify four such edges, (b_1, b_2), (b_{wh}, b_{if}), (b_{if}, b_7), and (b_3, b_5). In Figure 1c, these edges are bold face. When the execution control thread 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 clock speed so that the $C_{RWEC}(b_2)$ cycles can be completed exactly at the deadline, we ensure that the proposed technique always meets the required timing constraint. Because voltage-scaling decisions are made at compile time—not runtime—there exists no runtime overhead directly related to the selection of voltage-scaling edges. In addition, the compile-time analysis procedure does not require special programmer intervention other than that typically required in developing normal hard real-time programs (such as setting the maximum number of loop iterations).

Figure 2 compares how the speed and voltage changes, with and without the use of intra-task voltage scheduling. Assuming that no energy is consumed in an idle state and $E \propto C_L \times N_{cycle} \times V_{DD}^2$ when the execution follows path

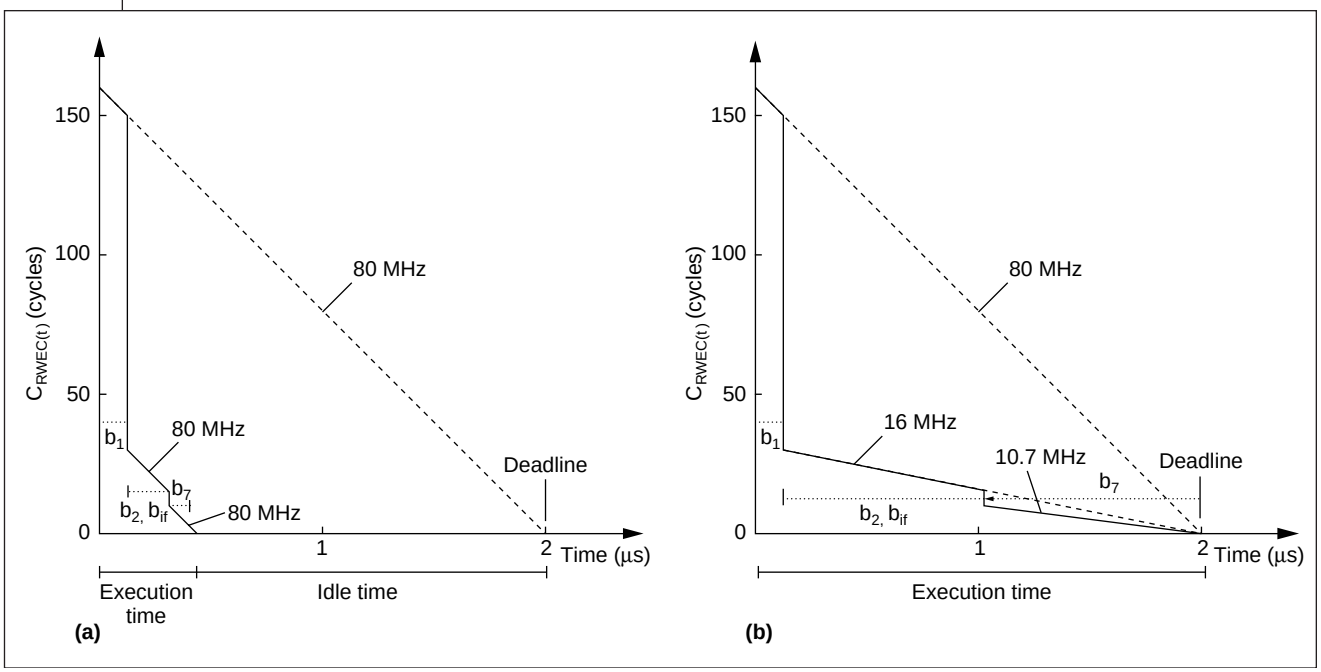

Figure 3. $C_{RWEC}(t)$ changes over different speed-scaling algorithms: no intra-task scheduling (a) and RWEC-based intra-task scheduling (b).

$p_1 = (b_1, b_2, b_{if}, b_6, b_7)$, the energy consumption ratio of Figure 2b to Figure 2a is 0.31. Using intra-task voltage scheduling reduces the energy consumption by 69%.

Intra-task voltage-scheduling algorithm

The intra-task voltage-scheduling algorithm assigns each basic block a proper speed at which to execute. For a hard real-time task, this algorithm's goal is to assign the speed to each basic block to minimize energy consumption while satisfying timing requirements. Throughout this article, we assume the following about the target variable-voltage processor:

- The processor provides special instruction change_f_V(f_{CLK}), which can dynamically control the processor's clock frequency f_{CLK} and its corresponding voltage V_{DD}.
- f_{CLK} and V_{DD} can be set continuously within the processor's operational range. When the processor changes clock and voltage, there is a clock/voltage transition overhead period of C_{VTO} cycles.
- During clock/voltage transition, the processor stops running and enters power-down mode.

Although some processors, such as Transmeta's Crusoe,[10] can run during voltage transition, for simplicity we assume that the processor stops. The techniques described in this article can support both processor types with a slight modification of clock/voltage transition overhead modeling.

Remaining WCET-based speed assignment

If actual execution path p_{act} of task τ were known in advance, the optimal execution speed could be easily computed. For each basic block b_i in p_{act}, $S(b_i) = C_{EC}(p_{act})/D$, where $S(b_i)$ represents the processor clock speed in frequency, $C_{EC}(p_{act})$ denotes the number of clock cycles needed to execute p_{act}, and D denotes the deadline of task τ.

However, because the exact execution path is generally unknown until program execution completes, we adjust $S(b_i)$ based on the remaining worst-case execution cycles $C_{RWEC}(b_i)$. Using a modified version of the static WCET prediction algorithm, such as the one developed by S.-S. Lim and colleagues,[9] we can estimate $C_{RWEC}(b_i)$ for each basic block b_i. $S(b_i)$ is set to clock speed S at which the remaining $C_{RWEC}(b_i)$ cycles can be

completed exactly at the deadline. The quantities $C_{RWEC}(b_i)$ are computed at compile time without incurring any runtime performance penalty.

At entry basic block b_1, $C_{RWEC}(b_1)$ is set to WCEC, and the starting speed is set to WCEC/D. If $C_{RWEC}(t)$ denotes the remaining worst-case execution cycles at time t, as execution proceeds, $C_{RWEC}(t)$ decreases linearly at the same rate as the clock speed when execution follows worst-case execution path p_{worst}.

However, if execution deviates from basic block b_i in 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. It drops by the difference between $C_{RWEC}(b_i) - C_{EC}(b_i)$ and $C_{RWEC}(b_j)$, where $C_{EC}(b_i)$ denotes the number of clock cycles needed to execute b_i.

Figure 3 shows how $C_{RWEC}(t)$ dynamically changes during execution of path $p = (b_1, b_2, b_{if}, b_7)$ from example program P. In Figure 3a, which illustrates an execution path that uses no speed scheduling, $C_{RWEC}(t)$ drops at two points: $C_{EC}(b_1)$/80 MHz and $[C_{EC}(b_1) + C_{EC}(b_2) + C_{EC}(b_{if})]$/80 MHz. Because the execution path of Figure 3a uses no speed scheduling, $C_{RWEC}(t)$ decreases at the rate of 80 MHz, resulting in a slack time interval of 1.5625 µs.

Figure 3b shows the effect of speed scheduling for the same execution path. Because $C_{RWEC}(t)$ drops right after executing b_1, the speed changes from 80 to 16 MHz, 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}, the speed also changes for the same reason.

Because the proposed RWEC-based intra-task scheduling makes all execution paths complete execution exactly at the deadline, the RWEC-based technique provides two benefits. It

- eliminates slack time, thus increasing energy efficiency; and
- guarantees that the scheduled program always meets the timing constraint.

We call the points in Figure 3 at which $C_{RWEC}(t)$ vertically drops voltage-scaling edges (VSEs), because the speed and voltage can be scaled at these points. The number of cycles reduced at VSEs is C_{saved}.

B-type voltage-scaling edges

We classify VSEs into two categories: B and L. B-type VSEs correspond to the CFG edge between two basic blocks that are part of conditional statements such as the **`if`** statement. For the **`if`** statement, 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 **`if`**-statement condition is evaluated in b_{cond}, the **`then`** path starts at b_{cond}, and the **`else`** path starts at b_{else}. If the **`if`** statement is 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, before the b_{then} block is executed, the speed can be decreased by a ratio of $C_{RWEC}(b_{then}) / C_{RWEC}(b_{else})$. We call this ratio a speed update ratio and represent it by $r(b_{cond} \rightarrow b_{then})$.

Because the same basic block can be executed at several different clock speeds, rather than associate each VSE with an absolute speed, we associate it with a speed update ratio. For example, in Figure 1, if we were to assign a fixed speed at the VSE between b_{if} and b_7, 53.3 MHz (80 MHz × 10/15) should be assigned so that p_{worst} can complete before the 2-µs deadline. However, if the executed path up to b_{if} is short—say (b_1, b_2, b_{if})—the execution will end far earlier than the deadline (resulting in a long slack time interval) if path (b_{if}, b_7) uses a fixed 53.3-MHz clock frequency. By assigning a speed update ratio $r(b_{if} \rightarrow b_7) = 2/3$ to the VSE, we avoid this problem.

In adjusting speed/voltage at VSEs, several instructions—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 B-type VSE (b_i, b_j) is

$$r(b_i \rightarrow b_j) = \frac{C_{RWEC}(b_j)}{C_{RWEC}\left[succ_{worst}(b_i)\right] - C_{overhead,B}} \quad (1)$$

In this equation, $succ_{worst}(b_i)$ is basic block b_k, an immediate successor of b_i with the largest $C_{RWEC}(b_k)$ among all b_i's successors.

If $C_{RWEC}(b_j) \geq C_{RWEC}[succ_{worst}(b_i)] - C_{overhead,B}$,

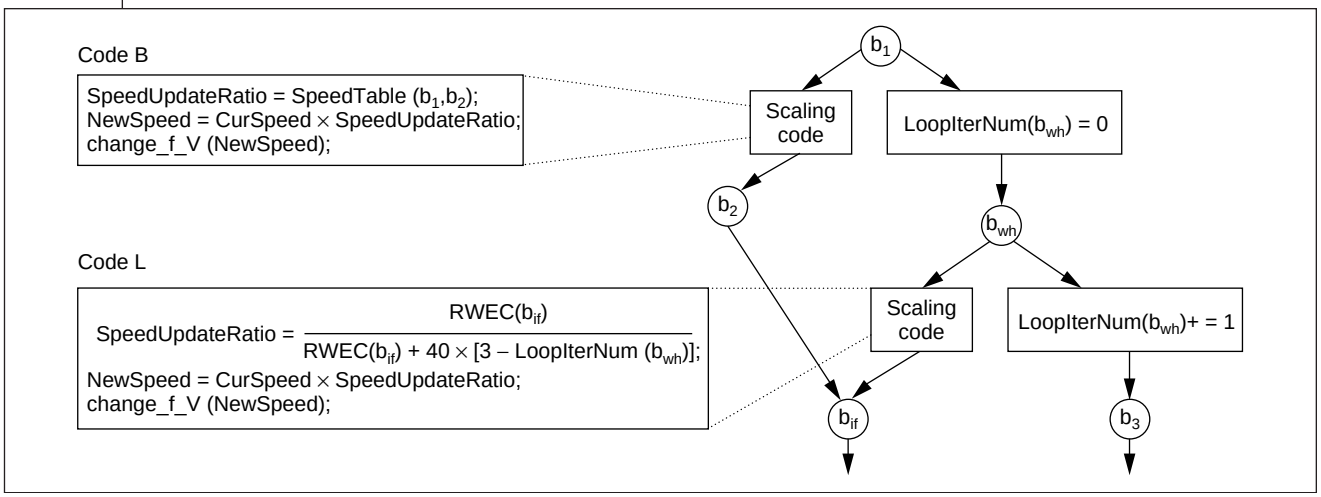

Figure 4. Given the control flow diagram on the right, our technique would insert B- and L-type voltage-scaling code as shown.

that is $r(b_i \rightarrow b_j) \geq 1$, 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 with the current speed before b_j starts its execution. That is, $S(b_j)$ is set to the current speed $\times\, r(b_i \rightarrow b_j)$.

As an example, consider how the speed changes at B-type VSEs as the execution proceeds following the path (b_1, b_2, b_{if}, b_7) of Figure 1, assuming $C_{overhead,B}$ is 0. Just before basic block b_2 executes, RWEC decreases from 150 to 30 cycles, so clock speed changes from 80 to 16 MHz (80 MHz × 30/150). The clock speed for basic block b_7 also changes from 16 to 10.7 MHz (16 MHz × 10/15) because RWEC changes from 15 to 10 cycles. Figure 4 shows the code generated for a B-type VSE (b_1, b_2) in Figure 1.

L-type voltage-scaling edges

Although our technique predicts WCEC assuming that a loop will execute the user-provided maximum number of iterations, a loop is generally iterated fewer times than the maximum loop bound. In this case, slack time exists and clock speed can be scaled down—a technique we call L-type scaling. L-type VSEs correspond to the loop exit edges in a CFG. In L-type scaling, saved cycles C_{saved} for loop l equal

$$C_{saved}(l) = C_{WCEC}(l) \times [N_{worst}(l) - N_{exec}(l)] \quad (2)$$

$C_{WCEC}(l)$ is the worst-case number of execution cycles to execute loop l once, $N_{worst}(l)$ is the user-provided maximum number of loops for loop l, and $N_{exec}(l)$ is the number of actual loop iterations measured at runtime. For L-type scaling, consider the edge (b_{wh}, b_{if}) in Figure 1, which is an example L-type VSE. When we denote the total number of overhead cycles at an L-type VSE as $C_{overhead,L}$, $S(b_{if})$ is updated as

$$S(b_{if}) = S(b_{wh}) \times \frac{C_{RWEC}(b_{if})}{C_{RWEC}(b_{if}) + C_{saved}(l) - C_{overhead,L}} \quad (3)$$

Assuming $S(b_{wh})$ = 80 MHz, $N_{exec}(l) = 1$, and $C_{overhead,L} = 0$, then $S(b_{if})$ decreases to 16 MHz before executing b_{if}.

Unlike for a B-type VSE, calculating the speed update ratio of an L-type VSE requires runtime information such as $N_{exec}(l)$. 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 a loop exit edge of loop l as an L-type VSE if $C_{WCEC}(l) > C_{overhead,L}$. Doing so 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.

Despite the increased code complexity for L-type VSEs, the overall reduction in energy consumption is still significant. This is because slack time arising from the execution of loops is generally far larger than that from conditional statements. For an L-type VSE (b_{wh}, b_{if}) in

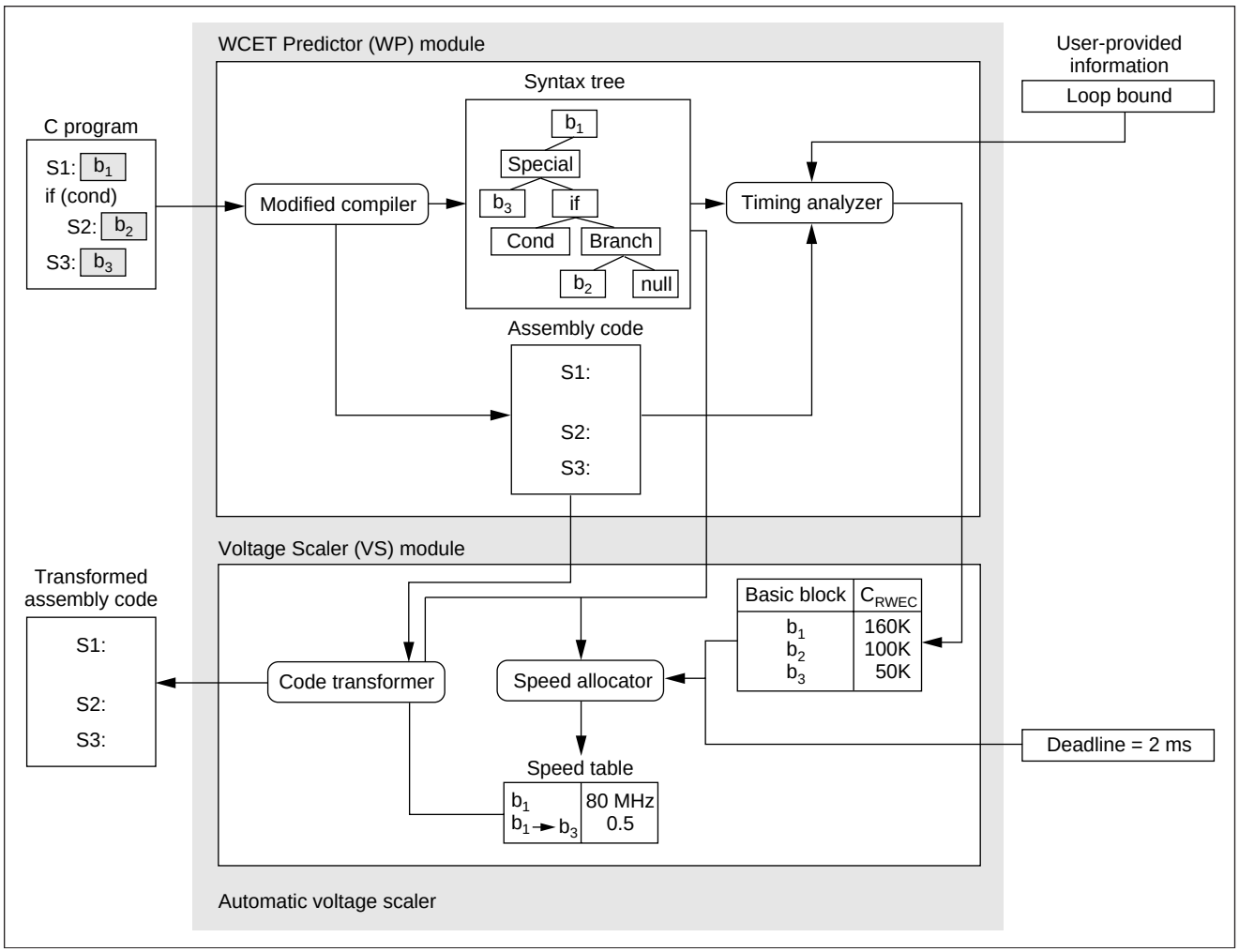

Figure 5. Automatic Voltage Scaler tool's overall structure.

Figure 1, Figure 4 shows the code sequence generated.

Automatic Voltage Scaler

We developed a software tool, the Automatic Voltage Scaler, to automate the development of hard real-time programs on a variable-voltage processor. This tool uses the intra-task scheduling algorithm. AVS takes as an input DVS-unaware program P and its timing requirements. It produces DVS-aware low-energy program P_{DVS}, which satisfies the same timing requirements as P. Converted program P_{DVS} contains voltage-scaling code that handles all the idiosyncrasies 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 way completely transparent to software developers. In the current version of AVS, we used the MIPS R3000 instruction set architecture as the target processor.

The WCET Prediction module estimates the $C_{RWEC}(b_i)$ values of all the basic blocks in an input program. To estimate $C_{RWEC}(b_i)$ of given basic block b_i, AVS uses a modified version of a timing tool developed by S.-S. Lim and his colleagues.[9] Their original timing tool estimates the WCET of an entire program by traversing the program's syntax tree bottom-up and applying the timing formulas of the extended timing schema (ETS). Because AVS uses RWEC from each basic block, we modified the original timing tool accordingly. As shown in Figure 5, the WP module, like the original timing tool,[9] takes as an input a high-level language program and

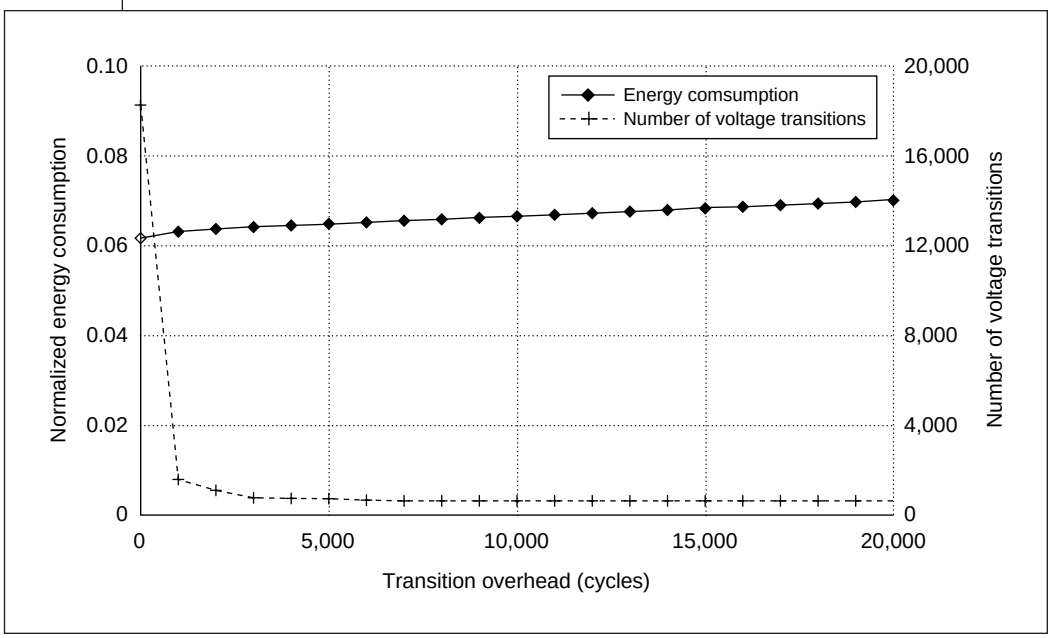

Figure 6. Energy consumption of the AVS-converted MPEG-4 decoder program normalized with respect to a conventional DVS-unaware system.

the user-provided information (such as the loop bound) to estimate $C_{RWEC}(b_i)$ values.

The Voltage Scaler module identifies VSEs based on $C_{RWEC}(b_i)$ values within the program syntax tree, assigns proper speeds to these edges, and generates a converted program. The Speed Allocator module in Figure 5 selects VSEs using the VSE selection algorithm and allocates appropriate speed update ratio r to each VSE. For example, in Figure 5, the speed update ratio of 0.5 is assigned to edge (b_1, b_3).

Experimental results

To evaluate the power reduction performance of AVS, we have experimented with an MPEG-4 video decoder. Because we don't have the proper hardware platform (one 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 the total energy consumption of the program's 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. We assume the energy consumption of power-down mode is 5% of the normal mode running at maximum clock frequency.[1] Supply voltage for a given clock frequency comes from $f_{CLK} = 1/T_D$, which is proportional to $(V_{DD} - V_T)^{\alpha}/V_{DD}$,[2] where V_{DD}, V_T, and α are assumed to be 2.5V, 0.5V, and 1.3. Clock/voltage transition overhead C_{VTO} is assumed to be 0 to about 20,000 cycles, corresponding to 0 to about 200 µs of transition time with a 100-MHz clock frequency.

Figure 6 shows the energy consumption of the AVS-converted MPEG-4 decoder program. (In converting the MPEG-4 decoder program, AVS took less than 100 ms.) Results were normalized over the energy consumption of the original program running on a DVS-unaware system. In Figure 6, the number of voltage transitions represents how many times voltage-scaling code was executed during the program execution. The AVS-converted program consumes less than 7% of the original program's energy consumption.

When $C_{VTO} < 1{,}000$ cycles, the number of voltage transitions decreases sharply, but energy consumption does not increase rapidly because the discarded VSEs have little effect on energy reduction. When $C_{VTO} > 5{,}000$ cycles, the number of voltage transitions remains nearly constant. The increase in energy consumption is due to the increased overhead cycles.

The number of VSEs—which represents how many copies of voltage-scaling code AVS inserted into the converted program—indicates the increase in code size caused by inserting voltage-scaling code via an inline expansion. For the AVS-converted MPEG-4 decoder, about 10 VSEs are sufficient when $C_{VTO} > 5{,}000$ cycles, meaning that insertion of voltage-scaling code hardly increases total code size. This is because only a few voltage-scaling edges are responsible for a large portion of the total power reduction.

By using the RWEC information for each basic block, the proposed technique makes it easier to apply intra-task voltage scheduling to DVS-unaware programs. First, it automatically selects appropriate program locations for performing voltage scaling to decrease overall energy consumption. Second, the proposed technique transparently inserts voltage-scaling code to the selected program locations. By automating these two steps, our algorithm makes it possible for programmers to develop DVS-aware programs on a variable-voltage processor without any knowledge of DVS.

Our work can be extended in several directions. We have based the speed assignment on RWEC but most program executions do not take the WCEP at runtime. We are currently devising the improved algorithm where the speed assignment is based on the ACEP.

We believe that both inter-task voltage scheduling and intra-task voltage scheduling have relative advantages and disadvantages over each other. It will be an interesting research topic to compare the two scheduling approaches quantitatively. ■

Acknowledgment

This work was supported in part by the Ministry of Information & Communication of Korea (Support Project of University foundation research < '00 > supervised by IITA). We thank Sung-Soo Lim for providing us with his WCET tool and explaining the tool's internals.

■ References

1. T. Burd and R. Broderson, "Processor Design for Portable Systems," *J. VLSI Signal Processing*, vol. 13, no. 2, 1996, pp. 203-222.
2. T. Sakurai and A. Newton, "Alpha-Power Law MOSFET Model and Its Application to CMOS Inverter Delay and Other Formulas," *IEEE J. Solid State Circuits*, vol. 25, no. 2, Feb. 1990, pp. 584-594.
3. I. Hong et al., "Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processor," *Proc. 19th IEEE Real-Time Systems Symp.* (RTSS 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 178-187.
4. Y. Lee and C.M. Krishna, "Voltage-Clock Scaling for Low Energy Consumption in Real-Time Embedded Systems," *Proc. 6th Int'l Conf. Real-Time Computing Systems and Applications* (RTCSA 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 272-279.
5. Y. Shin and K. Choi, "Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems," *Proc. 36th Design Automation Conf.*, IEEE Press, Piscataway, N.J., 1999.
6. F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy," *Proc. 36th Ann. Symp. Foundations of Computer Science* (FOCS 96), IEEE CS Press, Los Alamitos, Calif., 1995, pp. 374-382.
7. S. Lee and T. Sakurai, "Runtime Voltage Hopping for Low-Power Real-Time Systems," *Proc. 37th Design Automation Conf.*, IEEE Press, Piscataway, N.J., 2000, pp. 806-809.
8. C.A. Healy, D.B. Whalley, and M.G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proc. 16th IEEE Real-Time Systems Symp.* (RTSS 95), IEEE CS Press, Los Alamitos, Calif., 1995, pp. 288-297.
9. S.-S. Lim et al., "An Accurate Worst-Case Timing Analysis for RISC Processors," *IEEE Trans. Software Eng.*, vol. 21, no. 7, July 1999, pp. 593-604.
10. M. Fleischmann, "Crusoe Power Management: Reducing the Operating Power with LongRun," *Proc. Hot Chips 12 Symp.*, Palo Alto, Calif., 2000.

Dongkun Shin is a PhD student at the School of Computer Science and Engineering, Seoul National University. His research interests include low-power systems, computer architecture, and embedded and real-time systems. Shin has a BS in computer science and statistics and an MS in computer science, both from Seoul National University, Korea. He is a member of the ACM.

Jihong Kim is an assistant professor in the School of Computer Science and Engineering, Seoul National University, Korea. His research interests include computer architecture, embedded systems, Java computing, and multimedia and real-time

systems. Kim has a BS in computer science and statistics from Seoul National University, and an MS and PhD in computer science and engineering from the University of Washington. He is a member of the IEEE and ACM.

Seongsoo Lee is a research professor in the Department of Information Electronics, Ewha Woman's University, Korea. His research interests include low-power VLSI systems, dynamic voltage scaling, and VLSI implementation of MPEG-2 and MPEG-4. Lee has a PhD degree in electrical engineering from Seoul National University, Korea. He is a member of the IEEE Circuits and Systems Society.