Optimizing Intratask Voltage Scheduling Using Profile and Data-Flow Information

Dongkun Shin and Jihong Kim, *Member, IEEE*

***Abstract*—Intratask dynamic-voltage scheduling (IntraDVS), which adjusts the supply voltage within an individual-task boundary, has been introduced as an effective technique for developing low-power single-task applications or low-power multitask applications, where a small number of tasks are dominant in total execution time. The original IntraDVS technique used the remaining worst case execution cycles, and the control-flow information to identify the voltage-scaling points (VSPs) of a program. In this paper, two kinds of improvement techniques enhancing the energy performance of the IntraDVS are proposed. One is to use profile information to optimize the voltage schedule for the remaining average-case execution path (RAEP-IntraDVS). The other is to use data-flow information to optimize the locations of VSPs [look-ahead IntraDVS (LaIntraDVS)]. The experimental results show that the RAEP-IntraDVS can reduce the energy consumption by 20% on average and the LaIntraDVS can reduce the energy consumption by 40%–45% compared with the original IntraDVS.**

***Index Terms*—Dynamic-voltage scaling, low-power design, power management, real-time systems, variable-voltage processor.**

I. Introduction

SINCE ENERGY consumption of CMOS circuits has a quadratic dependence on the supply voltage, lowering the supply voltage is the most effective way of reducing energy consumption. However, lowering the supply voltage also decreases the clock speed. The tradeoff introduced various dynamic-voltage-scheduling (DVS) techniques. DVS techniques change the clock speed and its corresponding supply voltage dynamically to the lowest possible level while meeting the task's deadline constraint.

In real-time systems, the utilization of the processor is frequently less than one even if all tasks run at worst-case execution time (WCET), meaning that there is always some slack time. Moreover, workload of each task may vary from time to time, which results in another kind of slack time. These slack times can be exploited to lower the supply voltage. We denote them as worst case slack time and workload-variation slack time, respectively. For real-time systems, several DVS techniques are proposed to utilize these slack times. There exist two DVS approaches for real-time systems depending on the scaling granularity. Intertask dynamic-voltage scheduling (InterDVS) [1]–[3] determines the supply voltage on task-by-task basis, while intratask dynamic-voltage scheduling (IntraDVS) [4]–[6] adjusts the supply voltage within an individual-task boundary. Given multiple tasks, the InterDVS techniques assign the proper speed to each task dynamically while guaranteeing all their deadlines. 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 the supply voltage is determined as a constant value for a given 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 multitask environment, InterDVS may not be effective in energy reduction if the execution time of one task is dominant in the total execution time [5].

IntraDVS was proposed as a solution to overcome the limitations of InterDVS. Since IntraDVS does not involve OS in adjusting the clock speed, it has an advantage that existing OS can be used without any modifications on a variable-voltage processor. The main issue of IntraDVS algorithm is how to select the program points where the voltage and clock will be scaled. Depending on the selection mechanism, we can separate IntraDVS algorithms into four categories.

A. Segment-Based IntraDVS

Technique partitions a task into fixed-length segments [4], [7]. After executing a segment, it adjusts the clock speed and supply voltage exploiting the slack times from the executed segments of a program. The segment-based IntraDVS was improved into collaborative IntraDVS technique [8]. In the collaborative IntraDVS, OS and compiler collaborate for the voltage scaling within a task. At the offline stage, the compiler annotates an the application's source code with power-management hints (PMHs). During the run time (online stage), the OS periodically changes the processor's clock frequency and voltage based on the temporal information provided by the PMHs.

B. Path-Based IntraDVS

Techniques use the control-flow information to find slack times [5], [6], [9]. It selects all the program locations where we can identify the changes of remaining workload and inserts voltage-scaling codes in compile time. The voltage-scaling code is executed at run time to exploit all slack times coming from run-time variations of different execution paths.

Manuscript received April 12, 2005; revised August 28, 2005 and December 25, 2005. This work was supported by the Ministry of Information and Communication (MIC), Korea, under the Information Technology Research Center (ITRC) support program supervised by the Institute of Information Technology Assessment under Grant IITA-2005-C1090-0502-0031. This paper was recommended by Associate Editor R. Gupta.

D. Shin is with Samsung Electronics Company, Seoul 100-742, Korea (e-mail: dongkun.shin@samsung.com).

J. Kim is with the School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Korea (e-mail: jihong@davinci.snu.ac.kr).

Digital Object Identifier 10.1109/TCAD.2006.883928

In designing a path-based IntraDVS algorithm, two key issues exist. The first issue is how to predict the remaining execution cycles. Depending on the prediction method, several IntraDVS algorithms are proposed. We have proposed two kinds of IntraDVS algorithms, using the remaining worst case execution path (RWEP-IntraDVS) [5] and using the remaining average-case execution path (RAEP-IntraDVS) [6]. Recently, Seo *et al.* proposed the IntraDVS algorithm using the remaining optimal-case execution path (ROEP-IntraDVS) [9]. The RAEP-IntraDVS and the ROEP-IntraDVS generate more energy-efficient schedules over the RWEP-IntraDVS, because they exploit the profile information of a task execution. While the remaining execution cycles are predicted based on the most frequent execution path in the RAEP-IntraDVS [6], the ROEP-IntraDVS uses the optimal predicted execution cycles considering the profile information [9].

The second issue of path-based IntraDVS is how to determine the voltage-scaling points (VSPs) in the program code. The optimal points are the earliest points where we can detect the slack times. For this purpose, the original path-based IntraDVS used the branch and the loop structures. For example, the technique inserts a voltage-scaling code at the termination point of the loop, because we can know the difference between the actual number and the user-provided maximum number of loop iteration. However, if we can know the bound on loop before executing the loop, there is no need to scale clock and voltage after the loop. Walsh *et al.* [10] proposed a parametric IntraDVS technique, which places voltage-scaling code at a point in the program where the bound on loop is set at run time.

C. Memory-Aware IntraDVS

Memory-aware IntraDVS utilizes the CPU idle times which are due to external memory stalls. While the compiler-driven IntraDVS [11] identifies the program regions where the CPU is mostly idle due to memory stalls at compile level, the event-driven IntraDVS [12], [13] uses several performance-monitoring events to capture the CPU idle time at run time.

D. Stochastic IntraDVS

Stochastic IntraDVS utilizes the stochastic information of a program's execution time [14], [15]. 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. This technique finds a speed schedule that minimizes the expected energy consumption while still meeting the deadline. A task starts executing at a low speed and then gradually accelerates, to meet the deadlines. Since the task might not follow the worst-case execution path (WCEP), it can happen that high-speed (and power eager) regions are avoided.

The main contributions of this paper are as follows: First, we raise the question whether ROEP-IntraDVS is an energy-efficient and practical solution when the voltage-transition overhead is considered. From this motivation, we derive three kinds of new RAEP-IntraDVS techniques. Two of them try to use the optimal formulation, but one adapts a simple and practical solution. Then, we examine the energy performances of the techniques by taking the transition overhead into account. From the evaluation, we propose a new practical and simple policy, which can reduce the energy consumption without increasing the voltage-scaling overhead. We also describe how we can guarantee the deadline constraint in the path-based IntraDVS using profile information.

Second, we propose optimization techniques, which move VSPs to earlier program points. While the existing path-based IntraDVS algorithms find the VSPs of a program using the control-flow information of the program, the proposed technique identifies the VSPs using the data-flow information of the program as well as the control-flow information [look-ahead IntraDVS (LaIntraDVS)]. While the parametric IntraDVS by Walsh *et al.* [10] focused on only the loop, the LaIntraDVS finds the earliest VSP for both the branch structure and the loop structure, thus, it is more general. We also consider the tradeoff between the code overhead and the energy reduction due to the VSP change.

The rest of this paper is organized as follows. The generic intratask dynamic voltage scheduling is described in Section II, and three previous IntraDVS techniques are presented in Section III. While the optimizing techniques for RAEP-based IntraDVS technique are introduced in Section IV, the LaIntraDVS techniques are proposed in Section V. In Section VI, experimental results with randomly generated control-flow graphs and MPEG-4 programs are discussed. Section VII concludes this paper with a summary and a future work.

II. Generic IntraDVS

A. Power and Energy in Variable-Voltage Processors

The power consumption of CMOS circuit is composed of the static power P_{static} and the dynamic power P_{dynamic}

$$P_{\text{CMOS}} = P_{\text{static}} + P_{\text{dynamic}}. \tag{1}$$

The static power is dissipated due to the leakage current, which depends on the threshold voltage and on the technological process. At the current process technologies, the static-power consumption can be ignored, but it will be more critical at future systems.

The dynamic-power consumption P_{dynamic} is represented by the following equation:

$$P_{\text{dynamic}} = \alpha \cdot C_L \cdot V_{\text{dd}}^2 \cdot f_{\text{clk}} = C_{\text{eff}} \cdot V_{\text{dd}}^2 \cdot f_{\text{clk}}. \tag{2}$$

α 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. C_{eff} is the effective load capacitance.

This means that the dynamic-power consumption in a CMOS circuit is proportional to 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 dependence to power.

In this paper, we focus on energy rather than power consumption. Formally, the energy consumed by a system is the amount of power used during a certain period of time. We can denote the dynamic-energy consumption E_{dynamic} during the time interval T as follows:

$$E_{\text{dynamic}} = \int_0^T P_d(t)dt = C_{\text{eff}} \cdot V_{\text{dd}}^2 \cdot f_{\text{clk}} \cdot T = C_{\text{eff}} \cdot V_{\text{dd}}^2 \cdot N_c. \tag{3}$$

$P_d(t)$ is the dynamic power at the time t, and N_c is the number of clock cycles during the time interval T. Equation (3) tells us that reducing only the clock frequency makes no change in the energy consumption, although it reduces the power dissipation (when we ignore the static-power consumption).

Unfortunately, we cannot reduce the supply voltage for free. The circuit delay T_d, which sets the clock frequency, depends on the supply voltage [16]

$$\frac{1}{f_{\text{clk}}} \propto T_d \propto \frac{V_{\text{dd}}}{(V_{\text{dd}} - V_t)^\gamma} \tag{4}$$

where V_t is the threshold voltage and γ is the saturation velocity index (γ is between one and two). For a sufficiently small V_t, we can rewrite the relation between clock frequency and supply voltage as

$$f_{\text{clk}} \propto {V_{\text{dd}}}^{(\gamma-1)}. \tag{5}$$

For this reason, supply voltage and clock frequency should be scaled together. Consequently, dynamic-voltage scaling provides the energy reduction but leads to a slow system. We use the processor speed S instead of f_{clk} as follows:

$$S = \frac{f_{\text{clk}}}{f_{\max}} \propto {V_{\text{dd}}}^{(\gamma-1)} \tag{6}$$

where $f_{\max}$ is the maximum clock speed. In this paper, for a simple description, we assume that the effective capacitance C_{eff} is one and $\gamma = 2$ and use the following equation to calculate the energy consumption:

$$E = N_c \cdot S^2. \tag{7}$$

The energy consumption E is not an absolute energy value but a relative value. In this paper, we use the relative energy consumption E instead of the real energy consumption.[1]

Throughout this paper, we make the following assumptions on a target variable-voltage processor. The processor provides a special instruction, `change_f_V`(f_{clk}), that can dynamically control clock frequency f_{clk} and the corresponding voltage V_{dd} of the processor. f_{clk} and V_{dd} can be set continuously within the operational range of the processor. This assumption is not realistic, because most of variable-voltage processors provide only discrete voltage/clock levels. We use this assumption for a simple explanation, but we provide the experimental results under the variable-voltage processors with finite-voltage levels at Section VI.

[1]If $\gamma < 2$, (7) should be changed into $E = N_c \cdot S^{2/\gamma-1}$.

We assume that there is no upper bound on processor speed to simplify the description of the generic intratask dynamic voltage scheduling, i.e., S can be larger than one. This unrealistic assumption is removed at Section III-B1. When the processor changes clock and voltage, there is a transition time overhead Δt and the power overhead Δp. For the time and power overheads, we assume a fixed value for all transitions.[2] During clock/voltage transition, the processor stops running and enters into power-down mode.

B. Details of IntraDVS

IntraDVS consists of two key steps: to predict a task's workload and to adjust the clock speed at run time depending on the real workload. At the prediction step, we calculate the remaining predicted execution cycles (RPEC) at a basic block b_i, which is a branching node in the control-flow graph, $\delta(b_i)$, as follows:

$$\delta(b_i) = c(b_i) + \mathcal{P}\left(\delta(b_j), \delta(b_k)\right). \tag{8}$$

$c(b_i)$ is the execution cycles for the basic block b_i, and $\mathcal{P}$ is the prediction function. The basic blocks b_j and b_k are the immediate successor nodes of b_i in the control-flow graph. Depending on the function $\mathcal{P}$, the RPEC are determined. In this paper, we assume that the target processor has a simple architecture without instruction pipelining and cache memory. Therefore, $c(b_i)$ is the same to the number of instructions in b_i, and $\delta(b_i)$ is the sum of $c(b_i)$ and $\mathcal{P}(\delta(b_j), \delta(b_k))$.[3]

There can be several methods to predict the execution cycles. With the predicted value of $\delta(b_i)$, 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.

For a loop, we use a following equation to predict the remaining execution cycles for a loop L:

$$\delta(L) = c(H_L) + \left(c(H_L) + c(B_L)\right) \cdot N_{\text{pred}}(L) + \delta(\text{post}_L). \tag{9}$$

$c(H_L)$ and $c(B_L)$ are the execution cycles of the header and the body of the loop L, respectively.[4] $N_{\text{pred}}(L)$ is the predicted number of loop iterations, and post_L denotes the successor node of the loop, which is executed just after the loop termination.

When the actual execution deviates from the (predicted) reference path (say, by a branch instruction), the clock speed

[2]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. The voltage transition time (overhead) is used to select the voltage-scaling points. This is done at the compile time. We cannot know the exact transition time at the compile time, because the real transition time is different depending on the executed path. Therefore, we assume the worst case transition time.

[3]For a complex architecture, we should use a more sophisticated operation such as timing schema instead of the simple addition. Lim *et al.* [17] proposed the timing schema, which can model the effects of instruction pipelining and cache memory. They presented a timing tool, which estimates the WCET of a program traversing the program's syntax tree. To extend this paper for the complex architecture, we can use the timing tools by inserting the prediction function $\mathcal{P}$ into the timing schema.

[4]H_L and B_L can be composed of several basic blocks.

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 identify 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, voltage-scaling codes are inserted into the selected program locations at compile time. The branching edges of the control-flow graph (CFG), i.e., branch or loop statements, are the candidate locations for inserting voltage-scaling codes, because it is where the prediction miss for the reference path can occur. They are called as VSPs, because the clock speed and voltage are adjusted at these points. While the VSP due to the branch statement is called a B-type VSP, the VSP due to the loop statement is called an L-type VSP. Moreover, the VSP can be categorized into Up-VSP and Down-VSP, where the clock speed is raised and lowered, respectively. At each VSP (b_i, b_j), the clock speed is determined using $\delta(b_i)$ and $\delta(b_j)$ as follows:

$$S(b_j) = \frac{\delta(b_j)}{T} = S(b_i) \cdot \frac{\delta(b_j)}{\delta(b_i) - c(b_i) - \Delta t} = S(b_i) \cdot r(b_i, b_j). \tag{10}$$

$S(b_i)$ and $S(b_j)$ are the clock speeds at the basic blocks b_i and b_j, respectively. T is the remaining time until the deadline from the edge (b_i, b_j). $r(b_i, b_j)$ is called as the speed-update ratio of the edge (b_i, b_j). Δt is the voltage-transition overhead. If $\Delta t > 0$, $r(b_i, b_j)$ can be larger than one even when $\delta(b_j) < \delta(b_i) - c(b_i)$. In this case, the edge (b_i, b_j) is excluded from the VSPs. That is, an edge is selected as a Down-VSP only when the number of saved cycles is larger than Δt. If $\delta(b_j) > \delta(b_i) - c(b_i)$ (in this case, $r(b_i, b_j)$ is always larger than one), we select the edge as an Up-VSP regardless of the value of $r(b_i, b_j)$ to meet the deadline. In this paper, we ignore the overhead time of voltage scaling for a brevity afterward.

For a loop, we should consider two cases. If the actual number of loop iterations N_{actual} is smaller than the predicted number of loop iterations N_{pred}, the clock speed is reduced after the loop as follows:

$$S(\text{post}_L) = S(\text{pre}_L) \cdot \frac{\delta(\text{post}_L)}{(c(H_L) + c(B_L)) \cdot (N_{\text{pred}}(L) - N_{\text{actual}}(L)) + \delta(\text{post}_L)}. \tag{11}$$

$S(\text{pre}_L)$ is the clock speed before executing the loop L. When the actual number of loop iterations N_{actual} is larger than the predicted number of loop iterations N_{pred}, we can take two kinds of approaches. The first one is to increase the clock speed only at the exit point of the loop. At this approach, the clock speed is increased as follows:

$$S(\text{post}_L) = S(\text{pre}_L) \cdot \frac{(c(H_L) + c(B_L)) \cdot (N_{\text{actual}}(L) - N_{\text{pred}}(L)) + \delta(\text{post}_L)}{\delta(\text{post}_L)}. \tag{12}$$

The second approach is to increase the clock speed after every execution of the loop's header once N_{actual} exceeds N_{pred}. At this approach, expecting optimistically that the loop will execute one more iteration, the clock speed is increased as follows:

$$S\left(B_L^{i+1}\right) = S(B_L^i) \cdot \frac{c(H_L) + c(B_L) + \delta(\text{post}_L)}{\delta(\text{post}_L)}. \tag{13}$$

$S(B_L^i)$ means the start speed of ith execution of the loop body.

III. IntraDVS Algorithms

A. RWEP-IntraDVS

Among the prediction policies for the remaining execution cycles, the simplest and most conservative one is to use the remaining worst case execution cycles (RWEC) for RPEC. We call this technique as RWEP-IntraDVS [5]. For the prediction function of (8), the following function is used to calculate the RWEC δ_w:

$$\delta_w(b_i) = c(b_i) + \text{Max}\left(\delta_w(b_j), \delta_w(b_k)\right). \tag{14}$$

The predicted total execution cycles calculated by this equation is the same to the worst case execution cycles (WCEC), and there are only Down-VSPs in CFG. Therefore, the speed is dropped at all VSPs, and there is no possibility of deadline miss.

For L-type VSP, we should use the maximum loop iteration number N_w. We insert a voltage-scaling code at the exit point of the corresponding loop. The RWEC of a loop L is determined as follows:

$$\delta_w(L) = c(H_L) + (c(H_L) + c(B_L)) \cdot N_w + \delta(\text{post}_L). \tag{15}$$

Although the RWEP-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. Therefore, it is inefficient.

B. RAEP-IntraDVS

RAEP-IntraDVS utilizes the profile information of a program execution to optimize the energy efficiency of IntraDVS. The average-case execution path (ACEP) is used as a reference path. ACEP is defined to be an execution path with the largest possibility to be executed. The average-case remaining execution

cycle δ_a is defined by the following equation, where b_j and b_k are the immediate successor nodes of a branching node b_i:

$$\delta_a(b_i) = c(b_i) + \begin{cases} \delta_a(b_j), & \text{if } p_j \geq p_k \\ \delta_a(b_k), & \text{otherwise.} \end{cases} \quad (16)$$

p_j and p_k are the probabilities of b_j and b_k, respectively, are executed after b_i at run time.

For L-type VSPs, we should use the average iteration number N_a

$$\delta_a(L) = c(H_L) + (c(H_L) + c(B_L)) \cdot N_a(L) + \delta_a(\text{post}_L). \quad (17)$$

The Down-VSPs in the RAEP-IntraDVS are selected with (10), considering the voltage-scaling overhead. However, all the edges where the clock speed should be raised should be selected as Up-VSPs, because the deadline miss can occur if the clock speed is not raised at the edges.

It is easily understood that using ACEP instead of WCEP is more energy efficient. For a typical program, about 80% of the program execution occur in only 20% of its code, which is called the hot path [18]. 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 [19]. 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.

1) 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 because there is an upper bound in the maximum clock frequency. 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 the 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 that can guarantee the timing constraint.

To overcome the deadline-miss problem of the pure RAEP-IntraDVS algorithm, we find the remaining safe execution cycles (RSEC), denoted by δ_s. 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 $S_{\text{LB}}(b_i)$. For a given start time of b_i, $\sigma(b_i)$, we can represent $S_{\text{LB}}(b_i)$ as follows:

$$S_{\text{LB}}(b_i) = \frac{c(b_i)}{d(b_i) - \sigma(b_i)}$$

where $d(b_i)$ is the deadline of b_i. The deadline of a basic block b_i means the latest completion of b_i to satisfy the deadline of the overall program and can be defined as follows:

$$d(b_i) = D - \frac{\delta_w(b_i) - c(b_i)}{f_{\max}}$$

where D is the deadline of overall program and $f_{\max}$ is the maximum clock speed. The deadline of b_i is estimated by subtracting the worst case execution time, which is required to execute all basic blocks after b_i under $f_{\max}$ from the relative deadline D.

The clock speed of b_i, $S(b_i)$ should be larger than $S_{\text{LB}}(b_i)$. Then, the RSEC can be represented as follows:

$$\begin{aligned} S(b_i) &= \frac{\delta_s(b_i)}{D - \sigma(b_i)} \\ &\geq S_{\text{LB}}(b_i) \\ &= \frac{c(b_i)}{d(b_i) - \sigma(b_i)} \quad (18) \end{aligned}$$

$$\delta_s(b_i) \geq \frac{D - \sigma(b_i)}{d(b_i) - \sigma(b_i)} c(b_i). \quad (19)$$

The right-hand side of (19) 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, $\text{lst}(b_i)$, which is defined as follows:

$$\text{lst}(b_i) = \min\left(D - \frac{\delta_a(b_i)}{S_{\max}(b_i)}, d(b_i) - \frac{c(b_i)}{f_{\max}}\right) \quad (20)$$

where $S_{\max}(b_i)$ is the maximum clock speed that b_i can have under the original RAEP-IntraDVS. Since the deadline miss is inevitable when the start time of b_i is larger than $d(b_i) - c(b_i)/f_{\max}$, $\text{lst}(b_i)$ should be estimated as the minimum value between $D - \delta_a(b_i)/S_{\max}(b_i)$ and $d(b_i) - c(b_i)/f_{\max}$.

Consequently

$$\delta_s(b_i) = \frac{D - \text{lst}(b_i)}{d(b_i) - \text{lst}(b_i)} c(b_i). \quad (21)$$

We should estimate both $\delta_a(b_i)$ and $\delta_s(b_i)$ for a basic block b_i. If $\delta_s(b_i)$ is larger than $\delta_a(b_i)$, the clock speed should be determined based on $\delta_s(b_i)$. Therefore, the clock speed is changed as follows at an edge (b_i, b_j):

$$S(b_j) = S(b_i) \frac{\max(\delta_a(b_j), \delta_s(b_j))}{\max(\delta_a(b_i), \delta_s(b_i)) - c(b_i)}. \quad (22)$$

When $\delta_s(b_i)$ is used for b_i, we can represent the latest end time of b_i, $\text{let}(b_i)$, as follows:

$$\begin{aligned} \text{let}(b_i) &= \text{lst}(b_i) + \frac{c(b_i)}{S(b_i)} \\ &= \text{lst}(b_i) + c(b_i) \cdot \frac{D - \sigma(b_i)}{\delta_s(b_i)} \text{ (by (18))} \\ &\geq \text{lst}(b_i) + c(b_i) \cdot \frac{D - \text{lst}(b_i)}{\delta_s(b_i)} \\ &\quad \text{(by the definition of lst}(b_i)) \\ &= \text{lst}(b_i) + c(b_i) \cdot \frac{d(b_i) - \text{lst}(b_i)}{c(b_i)} \text{ (by (21))} \\ &= d(b_i). \quad (23) \end{aligned}$$

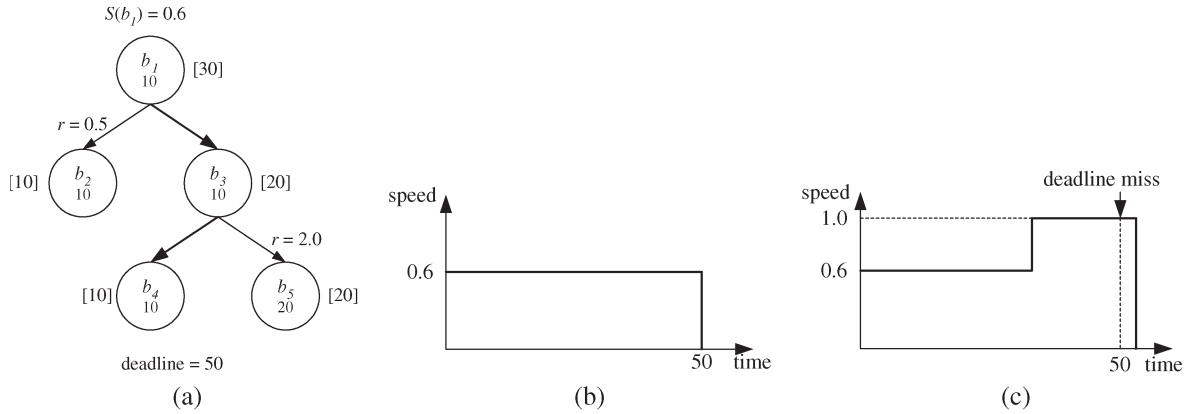

Fig. 1. Pure RAEP-IntraDVS. (a) Task graph. (b) Speed change at (b_1, b_3, b_4). (c) Speed change at (b_1, b_3, b_5).

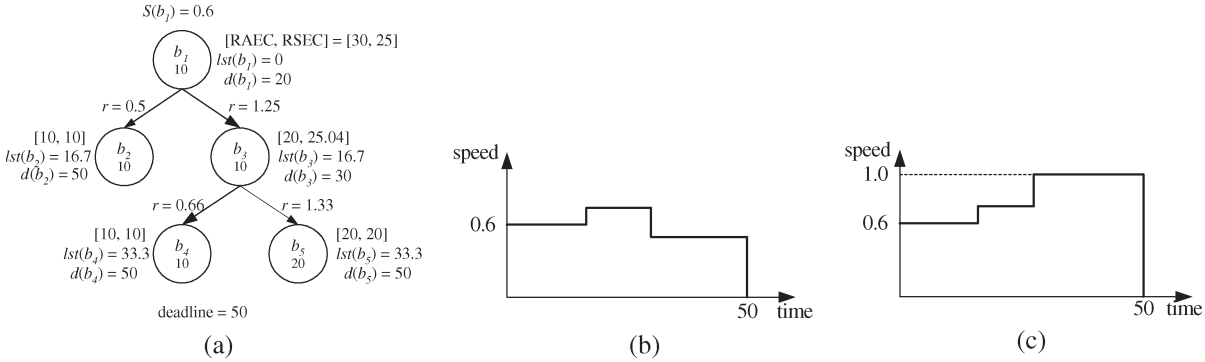

Fig. 2. Safe RAEP-IntraDVS. (a) Task graph. (b) Speed change at (b_1, b_3, b_4). (c) Speed change at (b_1, b_3, b_5).

Since the latest end time of b_i is smaller than the deadline of b_i, we can guarantee the safeness of this scheduling technique.

Fig. 1(a) shows a control-flow graph with a deadline 50 and the remaining average-case execution cycles (RAEC) 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 shown in Fig. 1(c), the deadline miss occurs for the execution path (b_1, b_3, b_5), because there is an upper bound of clock speed. Fig. 2(a) shows the RSEC as well as the RAEC of basic blocks. For example, the basic block b_3 has a deadline of $d(b_3) = 50 - (20/1.0) = 30$ and a latest start time of $\text{lst}(b_3) = 50 - (20/0.6) = 16.7$. Therefore, $\delta_s(b_3)$ is 25.04 $(= (50 - 16.7)/(30 - 16.7) \cdot 10)$ by (21). A basic block b_i uses the maximum between $\delta_s(b_i)$ and $\delta_a(b_i)$. Therefore, 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 $1.25 (= \delta_s(b_3)/(\delta_a(b_1) - c(b_1)) = 25.04/(30 - 10))$.

Using this safe RAEP-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. Although the speeds of basic blocks b_1 and b_3 are different in Fig. 2(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 for the path to be taken at run time. If the basic blocks $b_i, \ldots, b_j$ compose a reference path, we estimated the deadline and the latest start time of the group of basic blocks. The deadline and the latest start time of basic blocks $b_i, \ldots, b_j$, $d(b_i, \ldots, b_j)$ and $\text{lst}(b_i, \ldots, b_j)$ are same to $d(b_j)$ and $\text{lst}(b_i)$, respectively. For example, in Fig. 3(a), the group of basic blocks (b_1, b_3) has the deadline $d(b_1, b_3) = d(b_3) = 30$, and the latest start time $\text{lst}(b_1, b_3) = \text{lst}(b_1) = 0$. Using these values, we estimate $\delta_s(b_i), \ldots, \delta_s(b_j)$ as follows:

$$\delta_s(b_k) = \begin{cases} \frac{D - \text{lst}(b_i, \ldots, b_j)}{d(b_i, \ldots, b_j) - \text{lst}(b_i, \ldots, b_j)} \\ \quad \cdot (c(b_i) + \cdots + c(b_j)), & \text{if } k = i \\ \delta_s(b_p) - c(b_p), & \text{otherwise} \end{cases}$$

where b_p is the predecessor basic block of b_k. The speed schedules in Fig. 3(b) and (c), where the basic blocks b_1 and b_3 have a same speed, consume less energy than the schedules in Fig. 2(b) and (c). Moreover, the task graph in Fig. 3(a) has less voltage-scaling edges than the task graph in Fig. 2(a). This technique is called the profile-aware safe RAEP-IntraDVS to separate from the original safe RAEP-IntraDVS. For a brevity, we use the terminology of the safe RAEP-IntraDVS to denote the profile-aware safe RAEP-IntraDVS.

The safeness-guarantee process can also be applied to the basic blocks in a loop. A basic block b_i in a loop, which has N number of maximum loop iterations, has N number of $\delta_a(b_i)$ values. We denote $\delta_a(b_i)$ and $\delta_s(b_i)$ in the jth loop iteration as $\delta_a^j(b_i)$ and $\delta_s^j(b_i)$, respectively. By estimating $\delta_a^1(b_i), \ldots, \delta_a^N(b_i)$ and $\delta_s^1(b_i), \ldots, \delta_s^N(b_i)$ after unrolling the

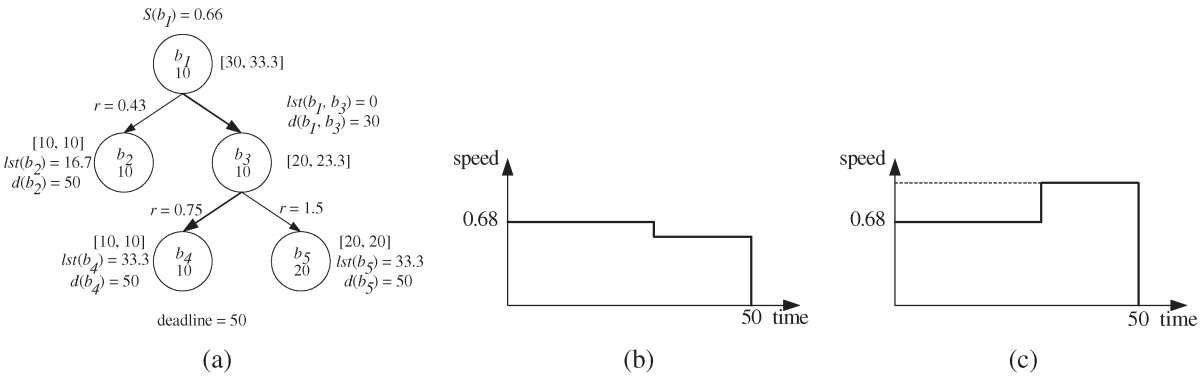

Fig. 3. Profile-aware Safe RAEP-IntraDVS. (a) Task graph. (b) Speed change at (b_1, b_3, b_4). (c) Speed change at (b_1, b_3, b_5).

loop, we can determine the speed-update ratios guaranteeing safeness. However, in the original IntraDVS, it is not necessary to unroll a loop to insert a voltage-scaling code. By representing the remaining execution cycles of a basic block in a loop with the loop-iteration index, a single voltage-scaling code can be inserted in a loop [5]. Therefore, it is required to insert the voltage-scaling code without unrolling the loop.

If there are j and $k(j \neq k)$, which satisfy the following equation:

$$\delta_a^j(b_i) \geq \delta_s^j(b_i) \text{ and } \delta_a^k(b_i) < \delta_s^k(b_i) \tag{24}$$

it is difficult to use a single voltage-scaling code in a loop. In the profile-aware safe RAEP-IntraDVS, all the basic blocks in a reference path are treated as a group of basic block, which share the same deadline and the same latest start time. Therefore, using the profile-aware safe RAEP-IntraDVS, we can guarantee that there are no j and $k(j \neq k)$ that satisfy (24).

C. Optimal RAEP-IntraDVS

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. Especially when the average-case execution cycle is significantly smaller than the worst case execution cycle, the energy consumption in RAEP-IntraDVS can be larger than RWEP-IntraDVS if WCEP is executed at run time.

Therefore, Seo *et al.* [9] proposed a profile-based optimal IntraDVS algorithm, which considers both the probability and the execution cycles of a path. They showed that we can get the optimal voltage schedule using the following prediction rule under the provided profile information[5]:

$$\delta_a(b_i) = c(b_i) + \sqrt[3]{\delta_a(b_j)^3 \cdot p_j + \delta_a(b_k)^3 \cdot p_k}. \tag{25}$$

[5]The general form of this equation is $\delta_a(b_i) = c(b_i) + (\delta_a(b_j)^{3/(\gamma-1)} \cdot p_j + \delta_a(b_k)^{3/(\gamma-1)} \cdot p_k)^{(\gamma-1)/(\gamma+1)}$ where γ is the saturation-velocity index.

Though this prediction rule generates the optimal energy schedule, it has some serious drawbacks. Since both the out edges of a branching node are not the reference path, all branching edges become VSPs. This generates two critical problems. One is that the code size of the transformed application increases, because we should insert the voltage-scaling codes at all branching edges. The other is that the clock speed and voltage should be changed at all edges generating the voltage-transition overhead. In addition, since Seo *et al.* [9] assumed that all loop is unrolled for L-type VSPs, the clock speed and voltage should be changed at every loop iteration. While some Down-VSPs can be excluded when the speed-update ratio is larger than one in (10), all Up-VSPs should be used as noted in Section III.

IV. Considering Voltage-Transition Overhead

A. Near-Optimal RAEP-IntraDVS

To solve the problem of optimal RAEP-IntraDVS, we modify it such that only one of branching edges should be the VSP. In this approach, the path whose δ_a is closer to the optimal remaining execution cycles is selected as a reference path

$$\delta_a(b_i) = c(b_i) + \begin{cases} \delta_a(b_j), & \text{if } g(b_j, b_k, p_j, p_k) \leq 0 \\ \delta_a(b_k), & \text{otherwise} \end{cases} \tag{26}$$

$$g(b_j, b_k, p_j, p_k) = \left| \delta_a(b_j) - \sqrt[3]{\delta_a(b_j)^3 \cdot p_j + \delta_a(b_k)^3 \cdot p_k} \right| - \left| \delta_a(b_k) - \sqrt[3]{\delta_a(b_j)^3 \cdot p_j + \delta_a(b_k)^3 \cdot p_k} \right|. \tag{27}$$

B. Edge-Optimal RAEP-IntraDVS

Another technique against the problem of optimal RAEP-IntraDVS is to find the optimal VSP under the assumption that only one of branching edges should be a VSP. On determining a reference path, we have two choices in selecting one edge among two out-edges of a branch node. The case that generates lower energy consumption is optimal.

In general, we can represent the prediction rule of edge-optimal RAEP-IntraDVS as follows:

$$\delta_a(b_i) = c(b_i) + \begin{cases} \delta_a(b_j), & \text{if } h(b_i, b_j, b_k, p_j, p_k) \le 0 \\ \delta_a(b_k), & \text{otherwise} \end{cases} \tag{28}$$

$$h(b_i, b_j, b_k, p_j, p_k) = \left(\frac{c(b_i)}{\delta_a(b_j)} + 1\right)^2 \left(p_j \delta_a(b_j)^3 + p_k \delta_a(b_k)^3 + c(b_i)\delta_a(b_j)^2\right) - \left(\frac{c(b_i)}{\delta_a(b_k)} + 1\right)^2 \left(p_j \delta_a(b_j)^3 + p_k \delta_a(b_k)^3 + c(b_i)\delta_a(b_k)^2\right). \tag{29}$$

C. Weighted Probability-Based RAEP-IntraDVS

In this approach, we use the weighted probability using the following equation:

$$\delta_a(b_i) = c(b_i) + \begin{cases} \delta_a(b_j), & \text{if } \delta_a(b_j) \cdot p_j \ge \delta_a(b_k) \cdot p_k \\ \delta_a(b_k), & \text{otherwise.} \end{cases} \tag{30}$$

Using this equation, we can consider both the probability and the execution cycles of the remaining paths. For a loop, we use the same technique to the probability-based RAEP-IntraDVS, i.e., the average-loop iteration number is used. Under this technique, only one of two out edges is selected as a VSP. As will be shown in the experimental results, the RAEP-IntraDVS using weighted probability has similar energy performances to the optimal RAEP-IntraDVS.

In addition, if $\gamma < 2$, other RAEP-IntraDVS techniques should use more complex equations. However, the weighted-probability-based approach does not depends on the saturation-velocity index γ. Therefore, the weighted-probability-based RAEP-IntraDVS is more advantageous than other RAEP-IntraDVS techniques.

V. IntraDVS Using Data-Flow Information

A. Motivation

The original IntraDVS techniques select the VSPs using the control-flow information (i.e., branch and loop) of a target program. For example, in Fig. 4(a), the IntraDVS algorithm inserts the voltage-scaling code, change_f_$V()$, at line 19. At line 19, we can know that the RWEC is reduced because the function `func8` is not executed. However, we can decide the direction of the branch of the line 16 at an earlier point, because the values of x and y are not changed after the line 8 or the line 11. Fig. 4(b) shows the modified program, which adjusts the clock speed and the supply voltage at line 10 or line 15. The program in Fig. 4(b) consumes less energy than the one in Fig. 4(a), because the clock speed is flat after line 10 or line 15 if $w > 0$ and $x + y \le 0$.

This example shows that we can improve the energy performance of IntraDVS further if we can move VSPs to the earlier instructions. To change the VSPs, we should identify the data dependence using a data-flow-analysis technique. The data-flow analysis provides the information about how a program manipulates its data [20]. 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 LaIntraDVS technique.

The LaIntraDVS finds earlier VSPs than Walsh's parametric IntraDVS [10], because LaIntraDVS uses the multistep approach. In addition, LaIntraDVS handles both the branch structure and the loop structure while the parametric IntraDVS considered only the loop structure. LaIntraDVS also considers the overhead in moving a voltage-scaling code to an earlier point.

B. Single-Step LaIntraDVS

For LaIntraDVS, we need following postprocessing steps after the VSPs are selected by the original IntraDVS algorithm.

1) Given an original VSP s, we identify the branch condition $C(s)$, which is the necessary condition for s to be executed at run time. Using the variables in the expression of $C(s)$, we compose a set of condition variables $V(s)$. For example, in Fig. 4(a), the branch condition for the VSP at the 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., $V(s) = \{v, w, x, y\}$).
2) 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. The data predecessors are the program points that directly affect the variable v_i. The look-ahead points are the earliest program points, where we can get the value of v_i, which will not be changed until s. The data predecessor and the look-ahead point are defined formally as follows.

Definition 1: Given a program location `t`, a definition point 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 2: 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.

a) 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.
b) There is no other program location `p`$'$ 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`. 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 $\mathbb{L}(s, y) = \{8, 12\}$.

3) The look-ahead VSPs LaVSP(s) are identified. It is the earliest program point where we can identify the branch direction of the original VSP s. We are to move the

```
 1:  v = func1();
 2:  if (v > 0) {
 3:        w = func2();
 4:        x = 3;
 5:        y = -3;
 6:        z = func3();
 7:        if (z > 0)  {
 8:                  x = func4();
 9:        }
10:        else {
11:                  y = func5();
12:                  func6();
13:        }
14:        func7();
15:        if (w > 0)  {
16:                  if (x+y > 0)
17:                            func8();
18:                  else
19:                            change_f_V();
20:                  func9();
21:        }
22: }
```

(a)

```
 1:  v = func1();
 2:  if (v > 0) {
 3:        w = func2();
 4:        x = 3;
 5:        y = -3;
 6:        z = func3();
 7:        if (z > 0)  {
 8:                  x = func4();
 9:                  if (w>0 && !(x+y>0) )
10:                            change_f_V();
11:        }
12:        else {
13:                  y = func5();
14:                  if (w>0 && !(x+y>0) )
15:                            change_f_V();
16:                  func6();
17:        }
18:        func7();
19:        if (w > 0)  {
20:                  if (x+y > 0)
21:                            func8();
22:                  func9();
23:        }
24: }
```

(b)

Fig. 4. Example program for LaIntraDVS. (a) Original IntraDVS. (b) LaIntraDVS.

original VSP to the look-ahead VSPs. The look-ahead VSP is defined formally as follows.

Definition 3: Given a VSP s and the set of condition variables $V(s) = \{v_1, \ldots, v_n\}$ of s, a look-ahead point $\mathtt{p} \in \mathbb{L}(s, v_1) \cup \cdots \cup \mathbb{L}(s, v_n)$ is a look-ahead VSP (LaVSP) of s if there is no other look-ahead point $\mathtt{p}' \in \mathbb{L}(s, v_1) \cup \cdots \cup \mathbb{L}(s, v_n)$ along the path from p to s. The set of all look-ahead VSPs is denoted by LaVSP(s).

From this information, we can know that LaVSP$(s) = \{9, 12\}$.

4) We insert the voltage-scaling codes at the look-ahead VSPs. Fig. 4(b) shows the modified program with LaVSPs. At 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 lines 9 and 14, it is unnecessary to insert a control expression for the condition.

With the LaVSPs, one needs to determine the speed-update ratio. For example, if an original VSP (b_i, b_j) has the LaVSP p in RWEP-IntraDVS, the speed-update ratio at p is

$$r(p) = \frac{\delta_w(p) - (\delta_w(b_i) - c(b_i) - \delta_w(b_j))}{\delta_w(p)} \tag{31}$$

assuming there is no voltage-transition overhead time. The clock speed is adjusted as the amount of the reduced execution cycles at the VSP (b_i, b_j) (i.e., $\delta_w(b_i) - c(b_i) - \delta_w(b_j)$).

In Fig. 4(a), if the clock speed is f_{15} at line 15, the clock speed at line 19, f_{19}, will be

$$f_{19} = f_{15} \times \frac{C_{\text{func9}}}{C_{\text{func8}} + C_{\text{func9}}}$$

(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 Fig. 4(b), the clock speeds at line 10 and line 15 are

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

and

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

respectively.

C. Multistep 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.[6] This is the limitation of the single-step LaIntraDVS approach, where a look-ahead point is directly used as a VSP. To solve this problem, we propose the multistep look-ahead IntraDVS technique, where the look-ahead point is recursively processed to find earlier scaling points.

Fig. 5 shows an example of the multistep LaIntraDVS algorithm. For the program generated by the original IntraDVS algorithm [shown in Fig. 5(a)], the single-step

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

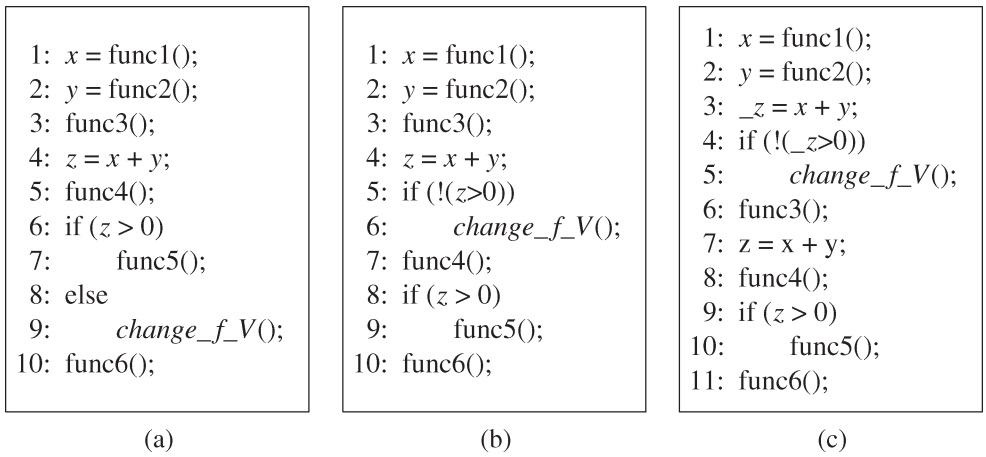

Fig. 5. Example program for multistep LaIntraDVS. (a) Original IntraDVS. (b) Single-Step LaIntraDVS. (c) Multistep LaIntraDVS.

```
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: }
```

Fig. 6. Multistep LaVSP search algorithm.

LaIntraDVS algorithm moves the scaling location to the line 6, as shown in Fig. 5(b). Since the variable z is defined at line 4, LaIntraDVS inserted the voltage-scaling code at 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 line 2. Fig. 5(c) shows the program modified using this idea. Since the variable z could be used before the definition point at line 7, we use the variable $\_z$ at the lines 3 and 4 (if the variable z is not used before 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 Fig. 5(c) compared with Fig. 5(b).

Fig. 6 summarizes the detailed steps of the multistep LaIntraDVS algorithm. The algorithm has two functions.

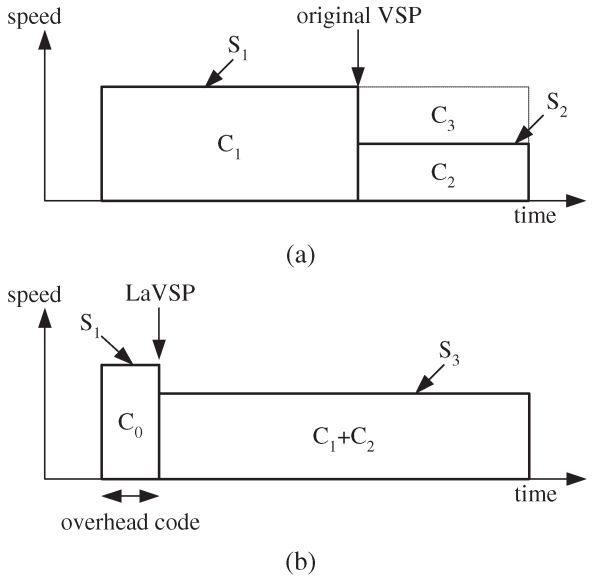

Fig. 7. Overhead in LaIntraDVS. (a) Original IntraDVS. (b) LaIntraDVS.

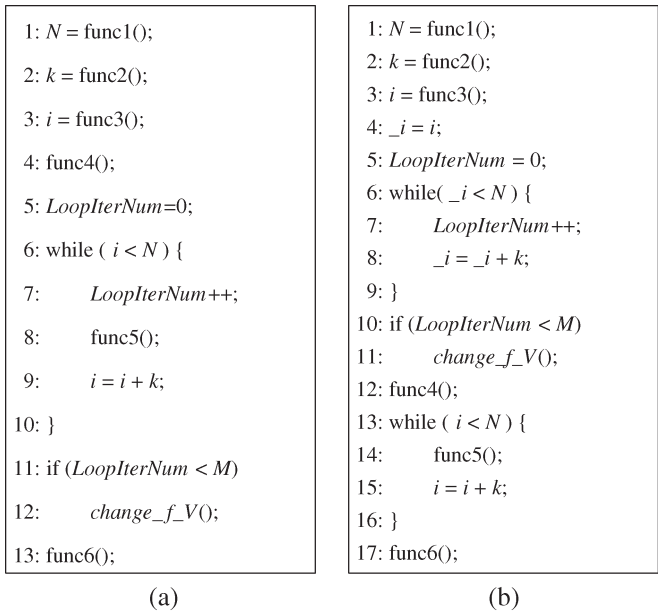

Fig. 8. Example program for L-type VSP. (a) Original IntraDVS. (b) LaIntraDVS.

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 multistep 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 multistep LaVSP technique such as the line 3 in Fig. 5(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 the intermediate data predecessor. Then, the variables in the data predecessor p are identified. For the data predecessor at the line 4 in Fig. 5(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, we should use the intermediate data predecessors as well as the conditions of the original VSP. For a variable that is defined in the intermediate data predecessors, the copy of the variable [e.g., $\_z$ in Fig. 5(c)] should be used to preserve the program behavior.

Fig. 7 shows how to estimate whether there is an energy gain when an LaVSP is used. In Fig. 7(a), the clock speed is changed from S_1 to $S_2 = S_1 \cdot C_2/C_3$ at the original VSP, because the remaining workload is changed from C_3 to C_2. In this case, the energy consumption can be computed as $E_{\text{org}} = C_1 S_1^2 + C_2 S_2^2$ by (7).

In Fig. 7(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 $E_{\text{La}} = C_0 S_1^2 + (C_1 + C_2) S_3^2$, where S_3 is $S_1 \cdot (C_1 + C_2)/(C_1 + C_3 - C_0)$.

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

$$E_{\text{org}} - E_{\text{La}} = C_1 S_1^2 + C_2 S_2^2 - C_0 S_1^2 - (C_1 + C_2) S_3^2 > 0.$$

The function EnergyGain in Fig. 6 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 Fig. 8(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 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 easy 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 [21], we can know the number of loop iterations. But, there is a simpler technique. For L-type VSPs, the loop-termination condition and the multistep LaIntraDVS technique are used.

For example, in Fig. 8(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 VSP LaVSP$(s) = \{3\}$. Therefore, we can insert the voltage-scaling codes after line 3. Fig. 8(b) shows the modified program by the multistep 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. All the variables

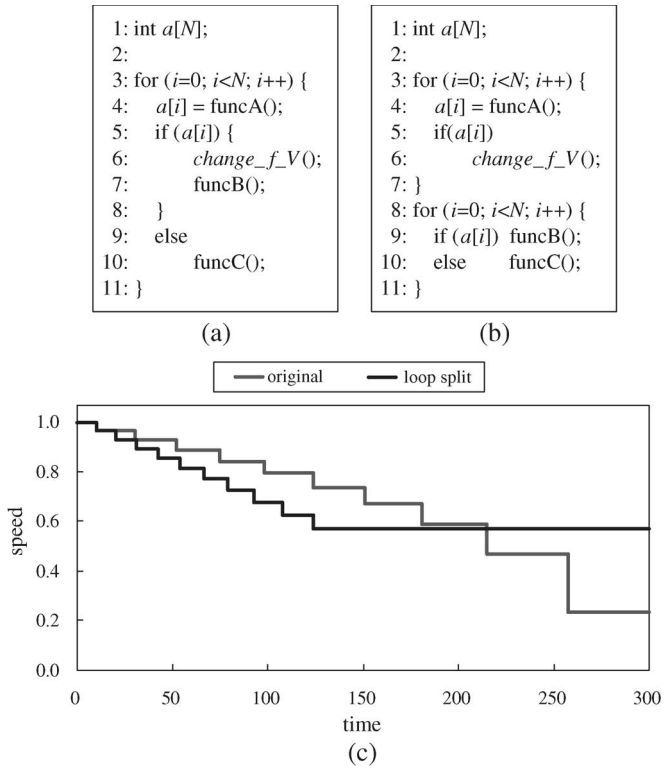

Fig. 9. Code transformation (loop splitting). (a) Original program. (b) Transformed program. (c) Speed-change graph.

defined at intermediate data predecessors are cloned like the variable $\_i$ at the line 4.

D. Further Enhancements

The LaIntraDVS is to move VSPs 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 is long. Therefore, it is better to schedule the look-ahead VSPs 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.

Fig. 9 shows the code transformation by loop splitting. In Fig. 9(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 to one, the VSP at line 6 reduces the clock speed. If the clock speed at line 5 is f_5, the clock speed is changed to

$$f_5 \cdot \frac{C_{\text{func}B} + (C_{\text{func}A} + C_{\text{func}C}) \cdot (9-i)}{C_{\text{func}C} + (C_{\text{func}A} + C_{\text{func}C}) \cdot (9-i)} = f_5 \cdot \frac{10 + 30 \cdot (9-i)}{20 + 30 \cdot (9-i)}$$

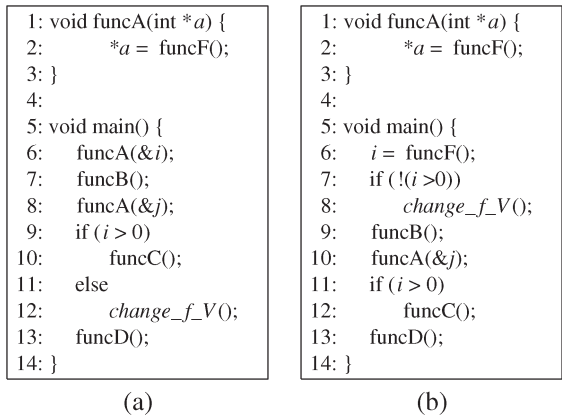

Fig. 10. Code transformation (function inlining). (a) Original program. (b) Transformed program.

at line 6 (assuming the voltage-transition overhead is zero). Since the look-ahead VSP (line 5) is the same as the original VSP, we cannot use the LaIntraDVS technique.

However, if we transform the program using loop splitting, as shown in Fig. 9(b), we can take full advantage of LaIntraDVS. While the original VSP is located in the second loop, the LaVSP is in the first loop. Whenever the value of each $a[i]$ is determined to be one at the first loop, we can reduce the clock speed at the LaVSP at line 6. If the clock speed at 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. Fig. 9(c) shows the speed-change graphs of two programs when all $a[i]$s are one. The clock speed of the program transformed by the loop splitting is reduced more quickly and does not change during the execution of the second loop. If we assume that the energy consumption is proportional to the square of 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 Fig. 10(a), there is a VSP in line 12 because the function funcC is not executed when $i > 0$. The data predecessor of the variable i is the line 2 in the function funcA. But, line 2 is not a look-ahead point of i, because the function funcA is called at line 8 with the input variable j. Therefore, we cannot move the VSP to line 3. If we inline the function funcA to the line 6 as shown in Fig. 10(b), the line 6 becomes a look-ahead point of the variable i. LaIntraDVS inserted the LaVSP to lines 7 and 8.

VI. Experiments

A. RAEP-IntraDVS

To compare the RAEP-IntraDVS with the RWEP-IntraDVS in the energy performance, we have experimented with randomly generated control-flow graphs (cfg1–cfg10). To make the random control-flow graph with a behavior similar to a real program's CFG, they were generated as follows. First, a sequential graph is generated, which has an N_{init} number of

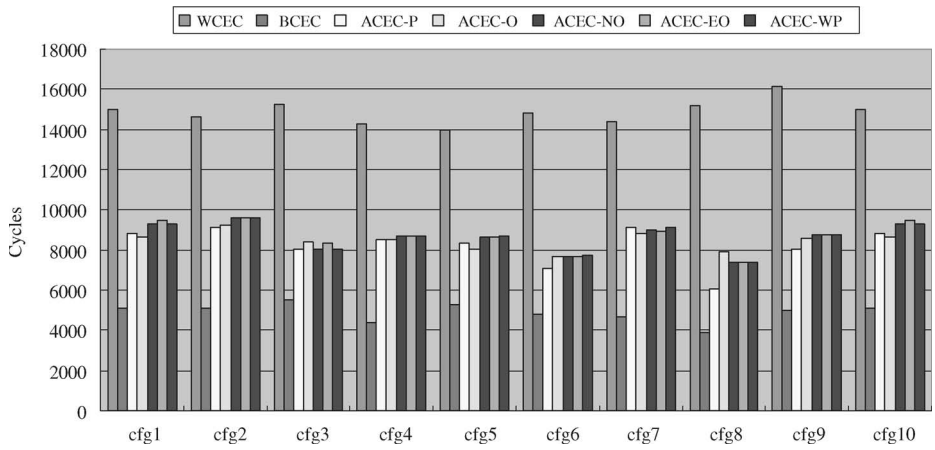

Fig. 11. Predicted execution cycles of programs under IntraDVS algorithms.

basic blocks. Each basic block was assigned a random value between $\mathrm{BB}_{\min}$ and $\mathrm{BB}_{\max}$ for the execution cycles. The average number of execution cycles is $\mathrm{BB}_{\mathrm{avg}}$. For the experiments, we used the values of 30, 5, and 100 for N_{init}, $\mathrm{BB}_{\min}$, and $\mathrm{BB}_{\max}$, respectively. Second, two adjacent basic blocks b_i and b_j, which are directly connected by an edge are randomly selected and other two basic blocks are inserted between them to make b_i as a branching node. This step is repeated until the number of basic blocks becomes the maximum number of basic blocks $N_{\max}$. The value of 600 is used for $N_{\max}$ in the experiments. Third, two separated basic blocks b_i and b_j, which are not connected directly, are randomly selected and a back edge from b_j to b_i is inserted to make b_i a header node of a loop. Fourth, we assign all branching edges the probabilities to be selected at run time. The maximum number and the average number of loop iteration are also assigned to all loops. Using this information, we make the profile information of the control-flow graph. For the run-time simulation, an execution path is generated according to the profile information. The simulation was performed 100 times generating 100 execution paths based on the given profile information. We assumed that the deadline of each task graph is 1.5 times of the worst case execution time.

Fig. 11 shows the numbers of predicted execution cycles of the control-flow graphs under the various IntraDVS techniques. The average-case execution cycles $\mathsf{ACEC}-\mathsf{P}$, $\mathsf{ACEC}-\mathsf{O}$, $\mathsf{ACEC}-\mathsf{NO}$, $\mathsf{ACEC}-\mathsf{EO}$, and $\mathsf{ACEC}-\mathsf{WP}$ are estimated by the probability-based, optimal, near-optimal, edge-optimal, and weighted probability-based RAEP-IntraDVS algorithms, respectively. All of these ACECs are between the WCEC and the best case execution. The average-case execution cycles have a similar value for a control-flow graph.

Fig. 12 shows the energy consumptions under various RAEP-IntraDVS techniques normalized by the energy under the RWEP-IntraDVS. For all RAEP-IntraDVS algorithms, the safeness-guarantee process proposed in Section III-B was applied. We have experimented varying the voltage-transition time Δt. Fig. 12(a)–(d) are the results when Δt are zero, $\mathrm{BB}_{\min}$, $\mathrm{BB}_{\mathrm{avg}}$, and $\mathrm{BB}_{\max}$, respectively. The energy consumptions include the power overhead Δp. We assumed that the power overhead Δp is the same to the power dissipated during the time Δt at the full speed. For all IntraDVS techniques, the Down-VSPs are selected considering the voltage-transition overhead as shown in (10).

Generally, the energy consumptions in the RAEP-IntraDVS techniques are smaller than the RWEP-IntraDVS technique. When the voltage-transition time and power overhead are ignored $(\Delta t = 0)$, the optimal RAEP-IntraDVS technique (RAEP-O) shows the best results. However, the energy performance of RAEP-O becomes worse as the value of Δt increases. This is because the optimal RAEP-IntraDVS technique involves many voltage transitions as commented in Section III.

Other RAEP-IntraDVS techniques (RAEP-NO, RAEP-EO, and RAEP-WP) show better results than the RWEP-IntraDVS irrespective of the voltage-transition overhead. Even when $\Delta t = 0$, they show little differences in the energy reduction with the optimal RAEP-IntraDVS technique, which shows the best results.

From the graph, we can know that the probability-based RAEP-IntraDVS (RAEP-P) can generate a worse energy performance than the RWEP-IntraDVS for some control-flow graphs (cfg8). This is because the probability-based RAEP-IntraDVS considers only the probability but not the execution cycles of an execution path.

Fig. 13 shows the normalized voltage-transition numbers of various RAEP-IntraDVS algorithms over the RWEP-IntraDVS. This result confirms the problem of the optimal RAEP-IntraDVS. The number of voltage transitions in the optimal RAEP-IntraDVS technique is two times the numbers in the RWEP-IntraDVS technique and three times the numbers in the other RAEP-IntraDVS techniques on average. The number of voltage transition is smallest in the probability-based RAEP-IntraDVS (RAEP-P), because the technique considers only the probability. Therefore, we can conclude that the weighted probability-based RAEP-IntraDVS is the most suitable, because it is the simplest but has a similar energy performance to the optimal RAEP-IntraDVS technique.

We also experimented the performance of proposed IntraDVS algorithms using a real processor model. Although we assumed that we could use any speed (and voltage) between the minimum speed and the maximum speed, real variable-voltage processors provide only finite number of speed levels. Fig. 14 shows the experimental results under four kinds of

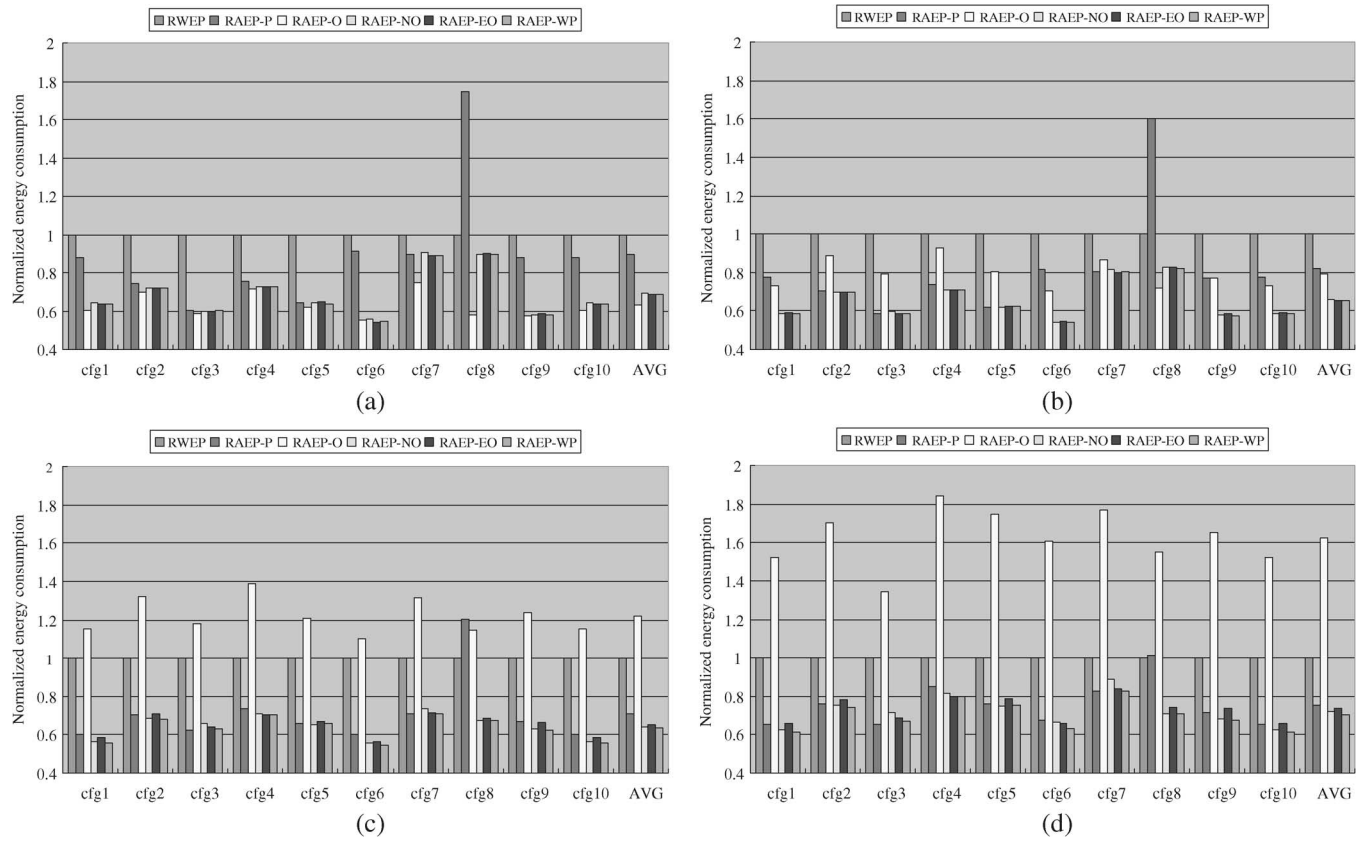

Fig. 12. Normalized energy consumptions of various RAEP-IntraDVS algorithms over the RWEP-IntraDVS: (a) $\Delta t = 0$, (b) $\Delta t = \mathrm{BB}_{\min}$, (c) $\Delta t = \mathrm{BB}_{\mathrm{avg}}$, and (d) $\Delta t = \mathrm{BB}_{\max}$.

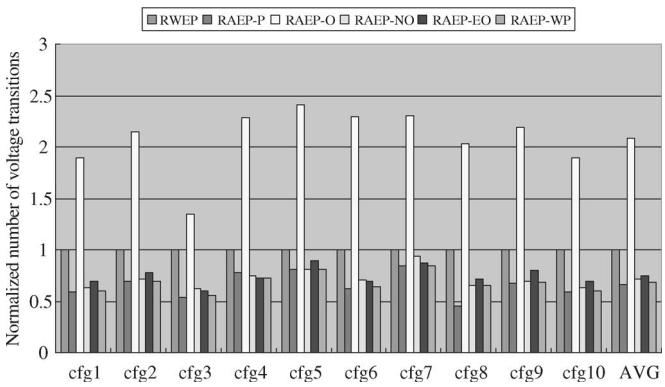

Fig. 13. Normalized voltage-transition numbers of various RAEP-IntraDVS algorithms over the RWEP-IntraDVS.

processor models, which provide 100, 50, 10, and 4 voltage levels, respectively. The processor with four levels of voltage is the PowerPC 405LP [22]. Under these real processor models, the target clock speed is first calculated as if the processor provides continuous speed levels. Then, the clock speed is changed to the value, which is smallest among the speed values larger than the target speed.

The energy consumptions decrease when the number of speed levels is larger than ten but increase when the available levels are smaller than ten. This is because the number of voltage transitions, which gives bad effects on the energy performance, decreases as the number of speed levels decreases. From this result, we can also know that it is not necessary to provide too many clock and voltage levels because a processor with many speed levels does not always give a good energy performance. As shown in Fig. 14(b), the number of voltage transitions sharply decreases as the number of speed levels decreases.

We also compared the energy reductions due to the RWEP-IntraDVS, the RAEP-IntraDVS, and the optimal DVS algorithms. The optimal DVS algorithm is to adjust the clock speed based on the actual execution time of a task at the start of the task. For the execution times of a task, we used artificial workloads, which are drawn from a random Gaussian distribution. Fig. 15 shows the normalized energy consumptions under the three kinds of DVS techniques. The energy gain of the RAEP-IntraDVS over the RWEP-IntraDVS increases as the B/W ratio (=BCET/WCET) decreases.[7] This is because the difference between ACEC and WCEC becomes larger as the B/W ratio decreases. When the B/W ratio is 0.1, the RAEP-IntraDVS reduces the energy consumption by 21% over the RWEP-IntraDVS.

For a real application, we have experimented with an MPEG-4 SP@L1 video decoder and encoder used in [23]. In the RAEP-IntraDVS, the probabilities of branch edges and the average numbers of loop iterations in the control-flow graph of MPEG-4 video programs 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 bit streams. For the deadlines of MPEG-4 programs, we used the worst case execution times.

[7]BCET = best-case execution time.

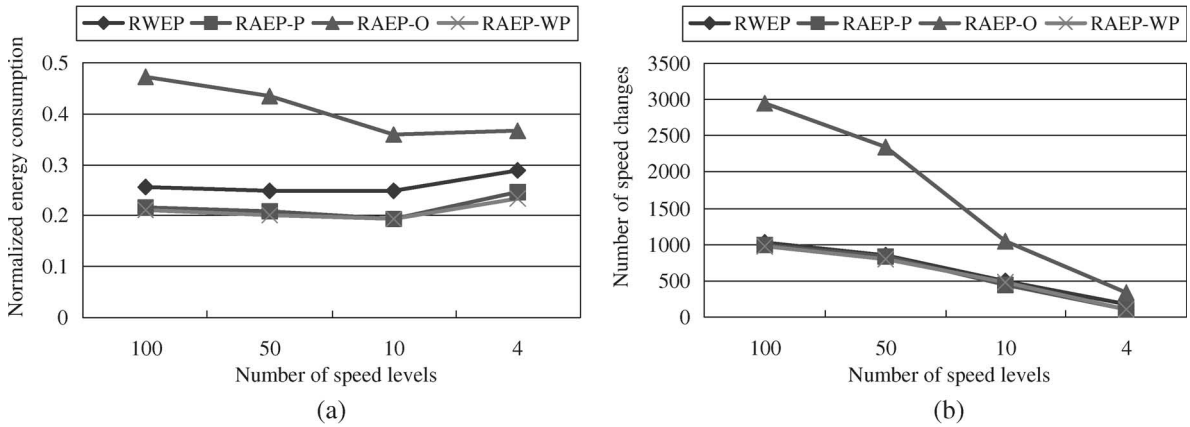

Fig. 14. Experimental results under finite-voltage levels.

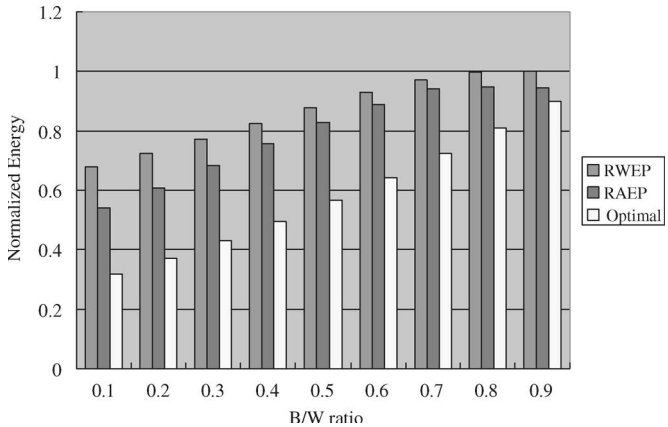

Fig. 15. Comparison among the RWEP-IntraDVS, the RAEP-IntraDVS, and the optimal DVS.

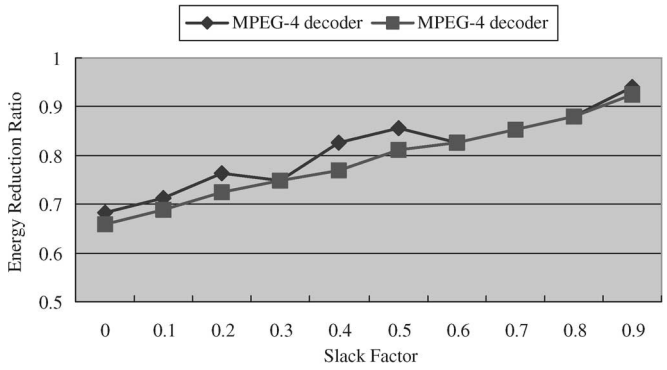

Fig. 16. Normalized energy consumption of the RWEP-IntraDVS and the RAEP-IntraDVS versus the slack factor.

Fig. 16 compares the energy consumption of two IntraDVS-scheduling algorithms, varying the slack factor. The slack factor, defined by $(\text{deadline} - \text{WCET}/\text{deadline})$, represents the fraction of time that a processor becomes idle after WCET. All the results were normalized over the energy consumption of the original program running on a DVS-unaware system. As the slack factor increases, the worst case slack time increases. To utilize the worst case slack time, we used WCET/deadline for the start speed of MPEG-4 programs.

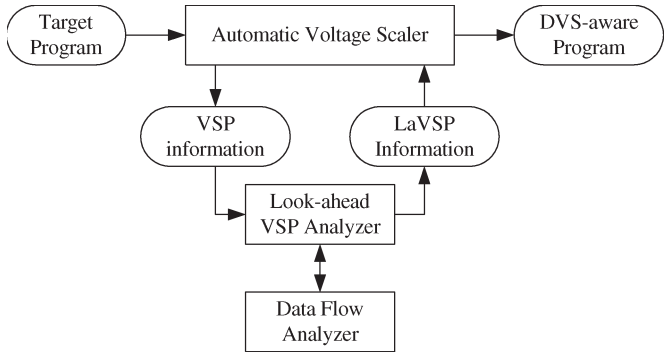

Fig. 17. Framework for LaIntraDVS.

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 value results in a smaller difference in the energy consumption. We can also know that there is no need to use an aggressive IntraDVS technique when the worst case slack time is significantly large because most of energy gains are generated from the worst case slack time.

Note that there is a large gap between energy consumptions of RWEP-IntraDVS and RAEP-IntraDVS algorithms, even when the slack factor is zero (i.e., deadline = WCET). This is because, although ACEC is same to WCEC, there are many execution paths that can still take advantage of the RAEP-based speed settings. That is, although the blocks in WCEP should use the RSEC in order to meet the timing constraint, other blocks may use the RAEC taking advantage of the RAEP-based speed settings.

B. LaIntraDVS

In order to evaluate the energy efficiency of LaIntraDVS techniques, we have experimented with MPEG-4 video programs. We first made a framework for LaIntraDVS, as shown in Fig. 17. We used the automatic voltage scaler (AVS) introduced in a previous paper [5]. The original AVS takes a target program as an input, finds the VSP information using the original IntraDVS algorithm, and generates the modified DVS-aware program. We added the Look−ahead VSPAnalyzer, which generates the look-ahead VSPs for each VSP using the algorithm

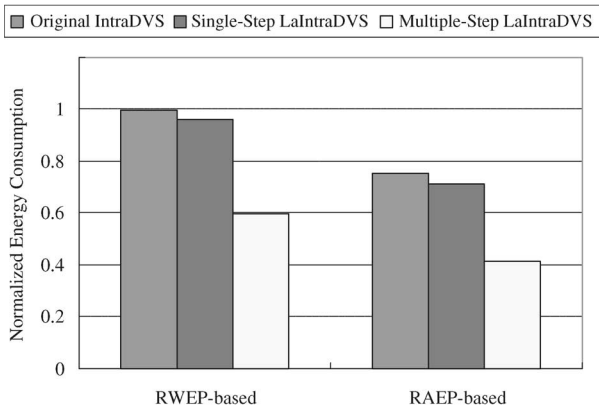

Fig. 18. Experimental results of LaIntraDVS.

in Fig. 6. The Data−Flow Analyzer finds the data predecessors for each VSP using the data-flow-analysis technique. The Data−Flow Analyzer corresponds to the function Find_Data_Predecessor in Fig. 6. Using the look-ahead VSP information, AVS generates the DVS-aware program.

Fig. 18 shows the energy consumption of three kinds of MPEG-4 encoder programs, which employ the original IntraDVS, the single-step LaIntraDVS, and the multistep 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-IntraDVS technique. As shown in Fig. 18, the single-step LaIntraDVS reduced the energy consumption by only 4%–6%. This is because most of look-ahead points are located closely to the original VSPs. However, the multistep LaIntraDVS shows significant energy reductions of 40%–45%. The energy performance of LaIntraDVS is dependent on the application characteristic. For an MPEG-4 decoder program, even the multistep LaIntraDVS showed little energy reductions.

The energy performance of the multistep LaIntraDVS is closely related to the application's slice size. Weiser [24] 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., the backward slice and forward slice. While a 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 multistep LaIntraDVS technique, a portion of a backward slice of a VSP should be copied before the LaVSP. Therefore, if the size of the backward slice is large, the overhead cycles for LaVSPs become large, limiting the energy gain of LaIntraDVS.

The slice size is dependent on the target program point. However, average slice size is considerably smaller compared with the original code size [24], [25]. The multistep LaIntraDVS applied to MPEG-4 encoder program copied only four C-statements for LaVSPs.

VII. Conclusion

The IntraDVS algorithm has two implementation issues, i.e., how to predict the remaining execution cycles and how to detect the change of the remaining execution cycles. There have been three kinds of IntraDVS algorithms, i.e., RWEP-IntraDVS, RAEP-IntraDVS, and ROEP-IntraDVS. These techniques exploited the control-flow information only to detect the workload change. In this paper, we proposed novel techniques for the two implementation issues to optimize the energy performance of IntraDVS.

First, we proposed a novel RAEP-IntraDVS algorithm that uses weighted-probability to consider both the energy performance and the voltage-scaling overhead. We also presented how we can get the safe average-case execution cycle, which guarantees the deadline constraint. In experiments using randomly generated control-flow graphs, we compared the proposed RAEP-IntraDVS with the ROEP-IntraDVS algorithm, which uses the optimal remaining execution cycles but increases the number of voltage transitions, as well as the RWEP-IntraDVS. The proposed RAEP-IntraDVS needed small voltage-transition overheads and showed better energy performances than the RWEP-IntraDVS and the ROEP-IntraDVS.

Second, we proposed the LaIntraDVS algorithm that exploits data-flow information as well as control-flow information of a program. The LaIntraDVS optimizes the VSPs such that we can adjust the clock speed as early as possible. The experimental results using an MPEG-4 encoder showed that the LaIntraDVS can reduce the energy consumption by 40%–45% over the original IntraDVS algorithm.

Although we have focused on the dynamic-power consumption assuming that the static part can be ignored, the static power will become a significant portion of the total power at future processors. As reported in [26], the leakage power currently accounts for about 15%–20% of the total power for high-speed processors. Recently, several techniques have been proposed that minimize the leakage power. Generally, these techniques shut the system down at the idle interval. 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. We have a plan to consider the leakage power as well as the dynamic power in dynamic-voltage scaling as a future work.

References

[1] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Proc. Des. Autom. Conf.*, 1999, pp. 134–139.
[2] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *Proc. IEEE Real-Time Syst. Symp.*, 2001, pp. 95–105.
[3] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. 18th ACM SOSP*, 2001, pp. 89–102.
[4] S. Lee and T. Sakurai, "Run-time voltage hopping for low-power real-time systems," in *Proc. Des. Autom. Conf.*, 2000, pp. 806–809.
[5] D. Shin, J. Kim, and S. Lee, "Intra-task voltage scheduling for low-energy hard real-time applications," *IEEE Des. Test Comput.*, vol. 18, no. 2, pp. 20–30, Mar./Apr. 2001.
[6] D. Shin and J. Kim, "A profile-based energy-efficient intra-task voltage scheduling algorithm for hard real-time applications," in *Proc. Int. Symp. Low Power Electron. and Des.*, 2001, pp. 271–274.
[7] D. Mossé, H. Aydin, B. Childers, and R. Melhem, "Compiler-assisted dynamic power-aware scheduling for real-time applications," in *Proc. Workshop Compilers and Operating Syst. Low-Power*, 2000, pp. 28–39.
[8] N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven, "Energy management for real-time embedded applications with compiler support," in *Proc. Conf. Language, Compiler, and Tool Support Embedded Syst.*, 2002, pp. 284–293.

[9] J. Seo, T. Kim, and K.-S. Chung, "Profile-based optimal intra-task voltage scheduling for hard real-time applications," in *Proc. 41st Des. Autom. Conf.*, 2004, pp. 87–92.
[10] B. Walsh, R. van Engelen, K. Gallivan, J. Birch, and Y. Shou, "Parametric intra-task dynamic voltage scheduling," in *Proc. Workshop Compilers Operating Syst. Low Power*, 2003, pp. 38–45.
[11] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction," in *Proc. ACM SIGPLAN Conf. Program. Languages, Des., Implementation*, 2003, pp. 38–48.
[12] A. Weissel and F. Bellosa, "Process cruise control: Event-driven clock scaling for dynamic power management," in *Proc. Int. Conf. Compilers, Architecture, Synthesis Embedded Syst.*, 2002, pp. 238–246.
[13] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 1, pp. 18–28, Jan. 2005.
[14] F. Gruian, "Hard real-time scheduling using stochastic data and DVS processors," in *Proc. Int. Symp. Low Power Electron. Des.*, 2001, pp. 46–51.
[15] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with PACE," in *Proc. ACM SIGMETRICS Conf.*, 2001, pp. 50–61.
[16] 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, pp. 584–594, Apr. 1990.
[17] 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 Trans. Softw. Eng.*, vol. 21, no. 7, pp. 593–604, Jul. 1995.
[18] T. Ball and J. R. Larus, "Using paths to measure, explain, and enhance program behavior," *Computer*, vol. 33, no. 7, pp. 57–65, Jul. 2000.
[19] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proc. Int. Symp. Low Power Electron. and Des.*, 1998, pp. 197–202.
[20] S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA: Morgan Kaufmann, 1997.
[21] B. Lisper, "Fully automatic, parametric worst-case execution time analysis," in *Proc. Int. Workshop Worst-Case Execution Time Anal.*, 2003, pp. 85–88.
[22] C. Rusu, R. Melhem, and D. Mosse, "Maximizing the system value while satisfying time and energy constraints," *IBM J. Res. Develop.*, vol. 47, no. 5/6, pp. 689–702, 2003.
[23] S. Lee and T. Sakurai, "Run-time power control scheme using software feedback loop for low-power real-time application," in *Proc. Conf. Asia South Pacific Des. Autom.*, 2000, pp. 381–386.
[24] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.
[25] L. Bent, D. C. Atkinson, and W. G. Griswold, "A comparative study of two whole program slicers for C," Dept. Comput. Sci. Eng., Univ. California, San Diego, CA, Tech. Rep. CS2001-0668, 2001.
[26] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proc. 29th Int. Symp. Comput. Architecture*, 2002, pp. 148–157.

Dongkun Shin received the B.S. degree in computer science and statistics, the M.S. degree in computer science, and the Ph.D. degree in computer science and engineering from Seoul National University, Seoul, Korea.

He is currently a Senior Engineer with Samsung Electronics Company, Seoul. His research interests include low-power systems, computer architecture, and embedded and real-time systems.

Jihong Kim (M'00) received the B.S. degree in computer science and statistics from Seoul National University, Seoul, Korea, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Washington, Seattle.

He is currently an Associate Professor with the School of Computer Science and Engineering, Seoul National University. His research interests include embedded systems, computer architecture, Java computing, and multimedia and real-time systems.

Dr. Kim is a member of the Association for Computing Machinery.