Optimizing Intra-Task Voltage Scheduling Using Data Flow Analysis*

Dongkun Shin
School of CSE
Seoul National University
Seoul, Korea 151-742
E-mail: sdk@davinci.snu.ac.kr

Jihong Kim
School of CSE
Seoul National University
Seoul, Korea 151-742
e-mail: jihong@davinci.snu.ac.kr

Abstract— Intra-task voltage scheduling (IntraDVS), which adjusts the supply voltage within an individual task boundary, is an effective technique for developing low-power applications. In IntraDVS, slack times are estimated by analyzing program's control flow information. In this paper, we propose an optimization technique for IntraDVS using data flow information. The proposed algorithm improves the energy efficiency by moving the voltage scaling points to earlier instructions based on the analysis results of program's data flow. The experimental results using an MPEG-4 encoder program show that the proposed algorithm reduces the energy consumption by 40-45% over the original IntraDVS algorithm.

I. Introduction

Dynamic voltage scaling (DVS) is one of the most effective low-power techniques for real-time systems. DVS techniques change the clock speed and its corresponding supply voltage dynamically to the lowest possible level while meeting the task's performance constraint.

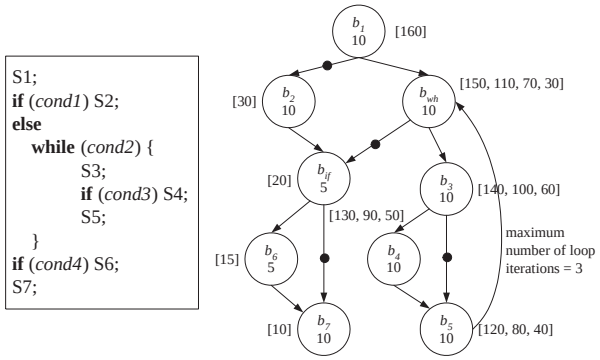
For real-time systems, there exist two DVS approaches depending on the scaling granularity. *Inter-task voltage scheduling* (InterDVS) [9, 1, 5] determines the supply voltage on task-by-task basis, while *intra-task voltage scheduling* (IntraDVS) [8, 7] adjusts the supply voltage within an individual task boundary. IntraDVS algorithms exploit all the slack time from run-time variations of different execution paths. Voltage scaling codes, which are inserted at specific program points (called voltage scaling points) of the target real-time program, adjust the clock speed depending on the execution path taken during run time.

In this paper, we propose two optimization techniques for IntraDVS algorithms, which moves voltage scaling points to earlier instructions. While the existing IntraDVS algorithms find the voltage scaling points of a program using the control flow information of the program, the proposed technique identifies the earlier voltage scaling points using the data flow information of the program as well as the control flow information.

We first briefly describe the original IntraDVS algorithm in Section II. Two proposed techniques are discussed in Section III and IV, respectively. Section V shows program transformation techniques advantageous to the proposed IntraDVS algorithm. Experimental results are given in Section VI. Section VII concludes with a summary.

II. Original Intra-Task Voltage Scheduling Algorithm

*This research was supported by University IT Research Center Project.



(a) program P (b) CFG G_P

Fig. 1. An example program for IntraDVS.

Scheduling Algorithm

Consider a hard real-time program P with a deadline shown in Figure 1(a). The control flow graph (CFG) G_P of the program P is shown in Figure 1(b). In G_P, each node represents a basic block of P and each edge indicates the control dependency between basic blocks. The number within each node indicates $C_{EC}(b_i)$ which is the number of execution cycles of the corresponding basic block. The back edge from b_5 to b_{wh} models the **while** loop of the program P.

For the speed adjustment, intra-task voltage scheduling technique uses an adaptive 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 among all the execution paths that start from b_i. Using a modified WCET analysis tool, for each basic block b_i, we can compute $C_{RWEC}(b_i)$ in *compile time*. In Figure 1(b), the symbol [ ] contains the $C_{RWEC}(b_i)$ values of each basic block. For the basic blocks related to the **while** loop (i.e., b_{wh}, b_3, b_4, b_5), the corresponding nodes are associated with multiple $C_{RWEC}(b_i)$ values, reflecting the maximum three iterations of the **while** loop.

With the $C_{RWEC}(b_i)$ values computed, we can statically identify an edge (b_i, b_j) (of a CFG G) where $[C_{RWEC}(b_i) - C_{EC}(b_i)] \neq C_{RWEC}(b_j)$. For example, in Figure 1(b), we can identify four such edges, i.e., (b_1, b_2), (b_{wh}, b_{if}), (b_{if}, b_7) and (b_3, b_5), which are marked by the symbol $\bullet$. These marked edges form a set of candidate Voltage Scaling Points (VSPs): the VSPs from branch statements such as (b_1, b_2) are called B-type VSPs, and the VSPs from loop statements such as (b_{wh}, b_{if}) are called L-type VSPs. If an edge (b_i, b_j) is selected

as a VSP, it means that the clock speed will change when the thread of execution control branches to b_j from b_i. For example, the clock speed will be lowered when the basic block b_2 is executed after b_1 because the remaining work is reduced by 1/5 (i.e., the ratio of $C_{RWEC}(b_2)$ to $[C_{RWEC}(b_1) - C_{EC}(b_1)]$).

For each selected VSP, the clock speed is changed so that a new speed is fast enough to complete the remaining work at the deadline. For example, when the thread of execution control meets a VSP (b_i, b_j), the clock speed can be lowered because the remaining work is reduced by $[C_{RWEC}(b_i) - C_{EC}(b_i) - C_{RWEC}(b_j)]$. After b_i is executed at the clock speed S, the clock speed can be changed to reflect the reduction in the remaining work. The new clock speed for b_j is set to $S \times \frac{C_{RWEC}(b_j)}{C_{RWEC}(b_i) - C_{EC}(b_i)}$. We call $\frac{C_{RWEC}(b_j)}{C_{RWEC}(b_i) - C_{EC}(b_i)}$ as the speed update ratio (SUR) for the edge (b_i, b_j).

Since there exists the transition overhead during speed changes, not all the candidate VSPs are selected as VSPs. A candidate VSP is selected as a VSP when the number of reduced cycles at the candidate VSP is larger than a given threshold value. The threshold value is determined by a VSP selection policy, which is a function of the transition time overhead, the transition power overhead, and the code size increase (by the added scaling code).

In designing IntraDVS algorithms, two key issues exist. The first issue is how to predict the remaining execution cycles. Depending on the prediction method, the IntraDVS framework can be implemented into different IntraDVS algorithms, e.g., using the remaining worst-case execution path (RWEP) [8], using the remaining average-case execution path (RAEP) [7], and using the remaining optimal-case execution path (ROEP) [6].

The second one is how to determine the voltage scaling points in the program code. The optimal points are the earliest points where we can detect the changes of the remaining predicted execution cycles. The previous works are all based on the control flow information only. In this paper, we propose new IntraDVS techniques which take advantage of program data flow analysis.

III. Look-ahead Intra-Task DVS

A. Motivation

The original IntraDVS techniques select the voltage scaling points using the control flow information (i.e., branch and loop) of a target program. For example, in Figure 2(a), the IntraDVS algorithm inserts the voltage scaling code, *change_f_V()*, at the line 19. At the line 19, we can know that the remaining worst-case execution cycles are reduced because the function func8 is not executed. However, we can decide the direction of the branch at the line 16 earlier because the values of *x* and *y* are not changed after the line 8 or the line 11. Figure 2(b) shows the modified program which adjusts the clock speed and the supply voltage at the line 10 or the line 15. The program in Figure 2(b) consumes less energy than the one in Figure 2(a) because the functions func6 and func7 are executed with a lower speed if $w > 0$ and $x + y \leq 0$.

This example shows that we can improve the energy performance of IntraDVS further if we can move voltage scaling points to the earlier instructions. To change the voltage scaling points, we should identify the data dependency using a data flow analysis technique. The data flow analysis provides 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) Original IntraDVS

```
 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) Look-ahead IntraDVS

Fig. 2. An example program for look-ahead IntraDVS.

information about how a program manipulates its data [4]. Using data flow analysis, we can decide program locations where each variable is defined and used. We call the proposed IntraDVS technique based on data flow information as the **look-ahead IntraDVS** (LaIntraDVS) technique.

B. Single-Step Look-ahead IntraDVS

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

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

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

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

- *There exists one or more paths from* `p` *to* `t` *but there is no path from* `p` *to* `t` *such that the value of* `x` *is changed along the path.*
- *There is no other program location* `p'` *between* P_x^t *and* `p`, *which satisfies the first condition.*

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

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

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

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

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

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

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

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

In Figure 2(a), if the clock speed is f_{15} at the line 15, the clock speed at the line 19, f_{19}, will be

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

(when we consider only the execution cycles for functions), where C_{func8} and C_{func9} are the worst-case execution cycles for the functions func8 and func9 respectively. However, in Figure 2(b), the clock speed at the line 10 and the line 15 are

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

and

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

respectively.

```
 1: x = func1();
 2: y = func2();
 3: func3();
 4: z = x + y;
 5: func4();
 6: if (z > 0)
 7:       func5();
 8: else
 9:       change_f_V();
10: func6();
```

(a) Original IntraDVS

```
 1: x = func1();
 2: y = func2();
 3: func3();
 4: z = x + y;
 5: if (!(z>0))
 6:       change_f_V();
 7: func4();
 8: if (z > 0)
 9:       func5();
10: func6();
```

(b) Single-Step LaIntraDVS

```
 1: x = func1();
 2: y = func2();
 3: _z = x + y;
 4: if (!(_z>0))
 5:       change_f_V();
 6: func3();
 7: z = x + y;
 8: func4();
 9: if (z > 0)
10:       func5();
11: func6();
```

(c) Multi-Step LaIntraDVS

Fig. 3. An example program for multi-step look-ahead IntraDVS.

IV. Multi-Step Look-ahead IntraDVS

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

Figure 3 shows an example of the multi-step look-ahead IntraDVS algorithm. For the program generated by the original IntraDVS algorithm (shown in Figure 3(a)), the single-step LaIntraDVS algorithm moves the scaling location to the line 6 as shown in Figure 3(b). Since the variable z is defined at the line 4, LaIntraDVS inserted the voltage scaling code at the lines 5 and 6. However, the variable z is the sum of x and y, and the values of both x and y are known before the function func3. If the number of execution cycles for func3 is large and the addition operation requires small execution cycles, it is better to insert the addition code and the voltage scaling code just after the line 2. Figure 3(c) shows the program modified using this idea. Since the variable z could be used before the definition point at the line 7, we use the variable $\_z$ at the lines 3 and 4. (If the variable z is not used before the line 7, we do not need to use the variable $\_z$.) If $x + y \leq 0$, the function func3 is executed with a lower speed in Figure 3(c) compared with in Figure 3(b).

Figure 4 summarizes the detailed steps of the multi-step LaIntraDVS algorithm. The algorithm has two functions. The function MS_LaVSP_Search does the same operations with the single-step LaIntraDVS algorithm except that it calls Find_MDP. The function Find_MDP finds the multi-step data predecessors. It first finds the predecessor set, P, for an input variable. Each predecessor p in P is examined whether there is an energy gain when the cycle distance between s and p is $Distance(p,s)$ and the overhead value is $C_{overhead}$. This is to consider the overhead instructions required for the multi-step LaVSP technique such as the line 3 in Figure 3(c).

If there is an energy gain in spite of the overhead cycles $C_{overhead}$, we further examine the data predecessor p. In this

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

case, we call p as the intermediate data predecessor. Then, the variables in the data predecessor p are identified. For the data predecessor at the line 4 in Figure 3(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.

```
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. 4. Multi-step LaVSP search algorithm.

In transforming a program, the intermediate data predecessors are used as well as the conditions of the original voltage scaling point. For the variable which is defined in the intermediate data predecessors, we should use a copy of the variable (e.g., _z in Figure 3(c)) to preserve the program behavior.

Figure 5 shows how to estimate whether there is an energy gain when a LaVSP is used. In Figure 5(a), the clock speed is changed from S_1 to $S_2 = S_1 \cdot \frac{C_2}{C_3}$ at the original voltage scaling point because the remaining workload is changed from C_3 to C_2. In this case, the energy consumption can be computed by $E_{org} = C_1 S_1^2 + C_2 S_2^2$ assuming that supply voltage is proportional to clock speed.

In Figure 5(b), LaIntraDVS found the look-ahead VSP which is executed C_1 cycles earlier than the original VSP. Assuming that we need C_0 overhead cycles to adjust the clock speed at the LaVSP, the energy consumption is given by $E_{La} = C_0 S_1^2 + (C_1 + C_2) S_3^2$ where S_3 is $S_1 \frac{C_1+C_2}{C_1+C_3-C_0}$. The condition for LaIntraDVS to be more energy-efficient than the original IntraDVS technique is $E_{org} > E_{La}$.

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

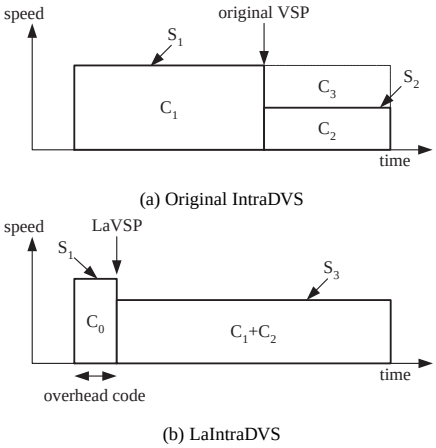


Fig. 5. Overhead in LaIntraDVS.

$$C_0 < C_1 + \frac{C_2^3}{C_3^2} - \frac{(C_1 + C_2)^3}{(C_1 + C_3 - C_0)^2}$$

The function EnergyGain in Figure 4 checks this condition to decide whether there is an energy gain.

For L-type VSPs, it is not trivial to make the condition for the VSPs. In Figure 6(a), the while loop executes $\lceil (N-i)/k \rceil$ times. In the original IntraDVS technique, the variable $LoopIterNum$ is used to know the number of loop iteration. The voltage scaling code at the line 12 reduces the clock speed if $\lceil (N-i)/k \rceil$ is smaller than the maximum number of loop iterations, M. Therefore, the condition for voltage scaling is $C(s) = \lceil (N-i)/k \rceil < M$. If we know the values of i, k, and N in advance, we can reduce the clock speed before the while loop. However, it is not trivial to derive the number of loop iterations $\lceil (N-i)/k \rceil$ from a program. Using a parametric worst-case execution time analysis technique such as [3], we can know the number of loop iterations. But, we can use a simpler technique. For the L-type VSP, the loop termination condition and the multi-step LaIntraDVS technique are used.

For example, in Figure 6(a), the loop termination condition is $C(s) = \neg(i < N)$. (Note that the expression does not have the variable k.) By analyzing data predecessors for the variables in $C(s)$, we can get $\mathbb{P}(s,i) = \{3,9\}$ and $\mathbb{P}(s,N) = \{1\}$. If we handle the data predecessor at the line 9 as an intermediate data predecessor, $\mathbb{P}(s,i)$ is changed into $\{2,3\}$. Using $\mathbb{P}(s,i)$ and $\mathbb{P}(s,N)$, we can get the look-ahead voltage scaling point $LaVSP(s) = \{3\}$. Therefore, we can insert the voltage scaling codes after the line 3. Figure 6(b) shows the modified program by the multi-step LaIntraDVS. To reflect the condition $C(s) = \neg(i < N)$, the while statement is inserted at the line 6. An assignment statement is also inserted at the line 8 because the statement is related to an intermediate data predecessor. All the variables defined at intermediate data predecessors are cloned like the variable _i at the line 4.

V. Further Enhancements

The look-ahead IntraDVS is to move voltage scaling points to the LaVSPs where we can predict the direction of a control flow. The energy reduction by LaIntraDVS is significant

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

(a) Original IntraDVS

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

(b) Look-ahead IntraDVS

Fig. 6. An example program for L-type VSP.

when the distance between the original VSP and the LaVSP is long. Therefore, it is better to schedule the look-ahead voltage scaling points as early as possible at the compiler level. We call this instruction scheduling as an *LaIntraDVS-aware* instruction scheduling.

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

Figure 7 shows the code transformation by loop splitting. In Figure 7(a), we assume that the execution cycles of functions funcA, funcB and funcC are 10, 10 and 20, respectively. When N is 10, the worst-case execution cycles of this loop is 300 (when we consider only the execution cycles for functions). Whenever the function funcA returns 1, the voltage scaling point at the line 6 reduces the clock speed. If the clock speed at the line 5 is f_5, the clock speed is changed to

$$f_5 \cdot \frac{C_{funcB} + (C_{funcA} + C_{funcC}) \cdot (9-i)}{C_{funcC} + (C_{funcA} + C_{funcC}) \cdot (9-i)} = f_5 \cdot \frac{10 + 30 \cdot (9-i)}{20 + 30 \cdot (9-i)}$$

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

However, if we transform the program using loop splitting as shown in Figure 7(b), we can take full advantage of LaIntraDVS. While the original VSP is located in the second loop, the LaVSP is in the the first loop. Whenever the value of each $a[i]$ is determined to be 1 at the first loop, we can reduce the clock speed at the LaVSP at the line 6. If the clock speed at the line 4 is f_4, the clock speed is changed to

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

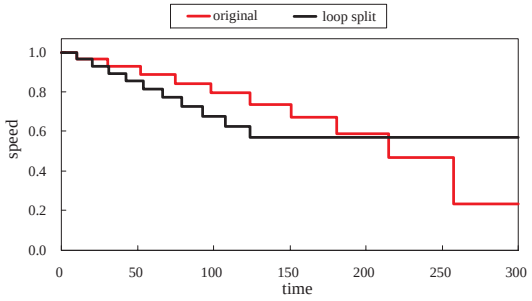
by the LaVSP. Figure 7(c) shows the speed change graphs of

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

(a) Original program

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

(b) Transformed program



(c) Speed change graph

Fig. 7. Code transformation: loop splitting.

two programs when all $a[i]$s are 1. 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 the clock speed, the LaIntraDVS technique with loop splitting reduces the energy consumption by 15% in this example.

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

VI. Experiments

In order to evaluate the energy efficiency of LaIntraDVS techniques, we have experimented with an MPEG-4 video encoder and an MPEG-4 decoder. We first made a framework for LaIntraDVS as shown in Figure 9. We used the automatic voltage scaler (AVS) introduced at the previous works [8]. 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 VSP Analyzer which generates the look-ahead VSPs for each VSP using the algorithm in Figure 4. The Data Flow Analyzer finds the data predecessors for each VSP using the data flow analysis technique. The Data Flow Analyzer

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

(a) Original program

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

(b) Transformed program

Fig. 8. Code transformation: function inlining.

corresponds to the function Find_Data_Predecessor in Figure 4. Using the look-ahead VSP information, AVS generates the DVS-aware program.

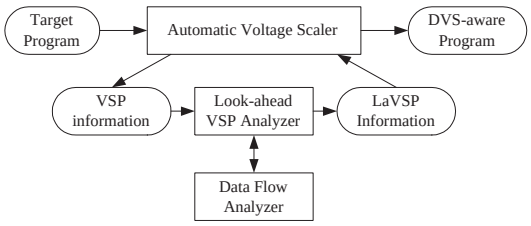


Fig. 9. The framework for look-ahead IntraDVS.

Figure 10 shows the energy consumption of three kinds of MPEG-4 encoder programs, which employ the original IntraDVS, the single-step LaIntraDVS and the multi-step LaIntraDVS, respectively. The figure also compares the results for the RWEP-based techniques and the RAEP-based techniques. The energy consumption is normalized by the result of the RWEP-based IntraDVS technique. As shown in Figure 10, 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 multi-step 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 multi-step LaIntraDVS shows little energy reductions.

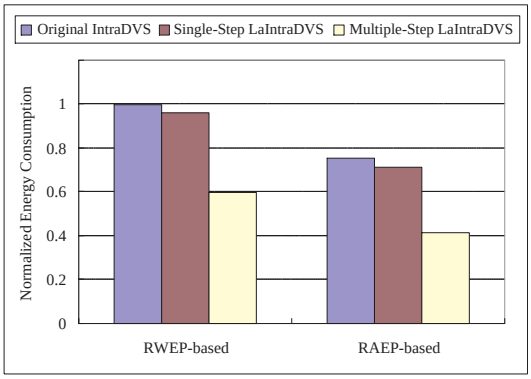


Fig. 10. Experimental results of look-ahead IntraDVS.

The energy performance of the multi-step LaIntraDVS is closely related to the application's slice size. Weiser [10] introduced the concept of program slice, which allows the user to focus on the portion of the program responsible for a particular phenomenon. There are two kinds of slices, i.e., backward slice and forward slice. While a backward slice consists of all program points that affect a given point in a program, a forward slice consists of all program points that are affected by a given point in a program. When we use the multi-step LaIntraDVS technique, a portion of 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 [10, 2]. The multi-step LaIntraDVS applied to MPEG-4 encoder program copied only four C-statements for LaVSPs.

VII. Conclusions

We proposed novel intra-task voltage scheduling algorithms called look-ahead IntraDVS, which exploit data flow information as well as control flow information of a program. The look-ahead IntraDVS optimizes the voltage scaling points such that we can adjust the clock speed based on the workload as early as possible. The experimental results using an MPEG-4 encoder showed that the look-ahead IntraDVS can reduce the energy consumption by 40∼45% over the original IntraDVS algorithm.

References

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

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

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

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

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

[6] J. Seo, T. Kim, and K.-S. Chung. Profile-Based Optimal Intra-Task Voltage Scheduling for Hard Real-Time Applications. In *Proc. of the 41st Design Automation Conference*, pages 87–92, 2004.

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

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

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

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