Parametric Intra-Task Dynamic Voltage Scheduling

Burt Walsh, Robert van Engelen, Kyle Gallivan, Johnnie Birch, and Yixin Shou
Department of Computer Science
and School of Computational Science and Information Technology
Florida State University
Tallahassee, FL 32306-4530
{*walsh,engelen,gallivan,birch,shou*}*@cs.fsu.edu*

Abstract

This paper presents a parametric intra-task dynamic voltage scheduling (IntraVS) method that scales voltage/frequency based upon the parameterization of the remaining worst case exection cycles (RWEC) of a task. The parametric RWEC of the task is determined by static analysis of code. The parameterization of the RWEC of loops with symbolic bounds, whose iteration space sizes are runtime dependent, allows the strategic placement of voltage scaling operations early on the execution path before the loop actually executes. This allows for greater energy reduction than with methods that scale voltage after loops are executed.

1. Introduction

With the proliferation of battery-powered embedded systems, energy has become a significant constraint for both hardware architects and compiler writers. One method of reducing power consumption in real-time systems is Dynamic Voltage Scaling (DVS). DVS algorithms adjust CPU frequency while executing a program so that the program complete execution right at its deadline.

DVS algorithms adjust clock frequency as slack is generated during the execution of the program. Slack is generated when the program flows a path less costly, in cycles, than the current assumed path. Because there is a quadratic dependency between supply voltage and energy consumption, and a linear relationship between clock speed and supply voltage, reducing clock frequency results in a linear slow down of the task completion rate but a quadratic savings in energy consumption. DVS use slack to slow down a task's execution speed so that the task still completes by its deadline but a quadratic savings in energy consumption results.

In *inter-task* DVS [11, 9, 13, 10, 16, 3] the voltage or clock frequency is set just before the next task executes based upon the difference between the next task's worst case execution time (WCET) estimate and its deadline, plus any slack generated from the previous task executing more quickly than its WCET estimation.

In *intra-task* DVS (IntraVS) [7, 15, 14] the voltage/frequency can be adjusted at many points within a task based upon the progress of the task being faster, or slower, than a pre-determined reference path; we will assume this path to be the worst case execution path. At each basic block Shin et al. defines the remaing worst case execution cycles to be the cycle cost of the worst case execution path from the current basic block to the end of the program, inclusive. The voltage/frequency is set such that the CPU finishes executing the task's RWEC exactly at the task's deadline. As slack gets generated, by executing paths less costly than the worst case path, voltage/frequency scaling code can reduce the speed of the processor while still having the task meet its deadline. Intra-task DVS has the benefit of only requiring the host processor to support voltage or clock frequency scaling calls, and does not require support from the OS scheduler [15].

Lee et al. [7] introduced IntraVS to scale voltage/frequency within tasks to achieve higher energy savings. Shin et al. [15, 14] extended this work so that adjustments to the voltage/frequency of the CPU were made as the task deviated from an assummed reference path of execution. This path could be the either the worst case execution path (WCEP) [15] or the average case execution path (ACEP) of the task [14]. AbouGhazealeh et al. [1] consider a combined compiler and operating system approach. The compiler uses the results from profiling to guide the placement of instrumentation, called power management hints (PMHs), that records information about remaining worst case execution cycles. The PMHs are used by the operating system scheduler, at configurable power management points (PMPs), to determine if voltage/frequency should be changed.

This paper presents a parametric intra-task dynamic voltage scheduling method which scales voltage/frequency based upon the parameterized RWEC and places the parametric voltage/frequency scaling code at strategic points as early as possible. The parametric RWEC formulae are determined by static analysis of code. Our approach complements existing methods with parametric representations of loop costs. This allows voltage to be scaled earlier in the execution of a task which can result in greater energy reductions.

The remainder of the paper is structured as follows. In Section 2 we describe IntraVS and motivate the use of a parametric approach. The static analysis and derivation of the parametric voltage scaling code is presented in Section 3. Finally, concluding remarks and directions for further research are given in Section 4.

2. Parametric IntraVS

2.1. IntraVS Methods

The IntraVS method of Shin et al. [15, 14] breaks a task up into a control flow graph (CFG). Static analysis, which takes cache effects and pipeline effects into account, is then used to calculate the worst case execution cycles of each basic block in the CFG [8]. The CFG of the task is then traversed, in a bottom up fashion, to determine the RWEC at each basic block in the task.

After determining the RWEC for each basic block voltage scaling code is placed at if-statements and at loop exit edges of the task. At if-statements, called B-type voltage scaling edges, voltage scaling code is placed immediately after the least costly branch if the cost of the least costly branch and the voltage scaling overhead is less than the more costly branch. At loop-exit edges, called L-type voltage scaling edges, voltage scaling code is added only if the target processor can complete the task by its deadline with the voltage scaling code added. Unlike B-type voltage scaling edges, L-type voltage scaling edges can increase the WCET of the task [15]. At each voltage scaling point, the voltage is scaled based upon any savings (slack) generated from the code running in fewer cycles than was assumed.

2.2. L-Type Voltage Scaling Placement

Figure 1 illustrates the placement of an L-type voltage scaling point at a loop exit edge. It is assumed that the loop takes a maximum number of iterations N, which is used by the static analyzer to determine the worst case execution cycles of the loop. The value N is provided by user annotations when this value is unknown to the static analyzer. When the loop executes fewer iterations and/or cycles, the generated slack is used at the L-type scaling point to scale back the voltage for the rest of the task execution.

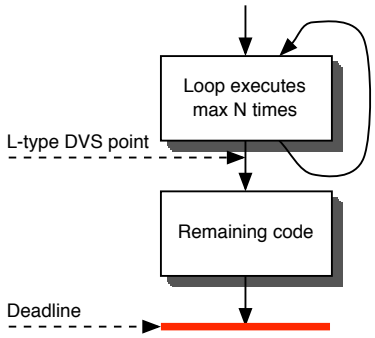


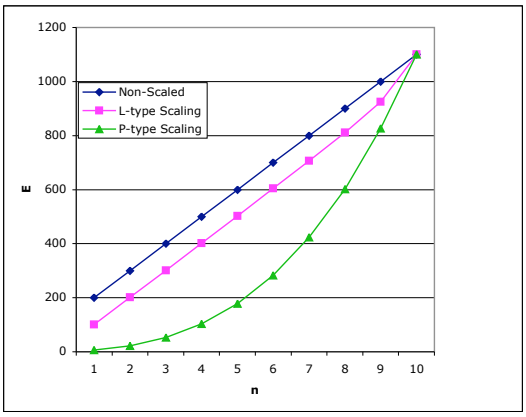
Figure 1. L-Type Dynamic Voltage Scaling

Loops are the major source of computation in scientific and multimedia applications. The L-type voltage scaling is not optimal due to the fact that the voltage, at which the loop runs, is not based upon the true size of the loop; but rather, the maximum possible loop size. This can be particularly significant in the case of nested loops with symbolic loop bounds. Also, this maximum value has to be specified by the user and this information can be error prone.

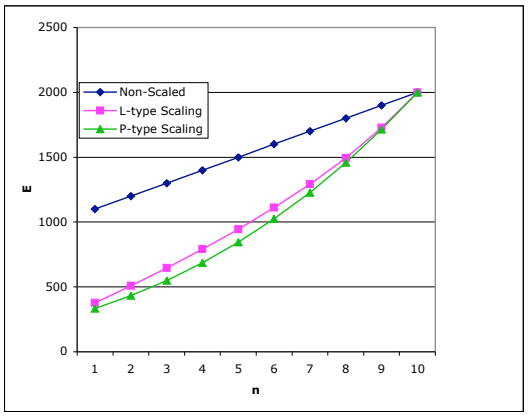
When the actual run-time size of the loop iteration space is much smaller than its maximum possible iteration space, the entire loop execution will run at a higher clock frequency, and consequently higher voltage, than necessary. After the loop has terminated, the remaining unused cycles (slack) are used to slow down the clock frequency for the rest of the program's execution. However, because the rate of energy consumption is proportional to the square of the voltage, a slowdown of the clock frequency at the loop exit cannot make up for the energy lost.

For example, assume that a loop executes at most N times. Furthermore, assume that the number of cycles per iteration is fixed c_L and the number of cycles of the remaining code is fixed c_R. At any point in the execution of the task, to guarantee that the task meets its deadline the following expression $\frac{1}{frequency} * RWEC$ must equal the time till the deadline. If there is no voltage/frequency scaling during code execution the voltage must be set before the loop executes. When computing the $RWEC$ at this point it must be assumed that the loop will execute for the maximum possible number of iterations N. Solving for frequency based upon the $RWEC$ gives us the required voltage and frequency at which to run the processor so that it makes the deadline. Since voltage is proportional to CPU frequency it is easy to get the required voltage. The loop will run at this voltage, called V, for the actual $n \leq N$ run-time iterations. This gives an approximate energy consumption:

$$E_{\text{NonScaled}} \approx V^2(n\,c_L + c_R)$$



E_{Ltype} and E_{Ptype} for $c_L = 100$ and $c_R = 100$



E_{Ltype} and E_{Ptype} for $c_L = 100$ and $c_R = 1000$

Figure 2. Energy Consumption of L-type and P-type Scaling

L-type voltage scaling at the loop exit point reduces the energy consumption by adjusting the voltage for the remainder of the code based upon run-time slack. Slack results when $n < N$. The approximate energy consumption is[1]:

$$E_{\text{Ltype}} \approx V^2(n\,c_L + (\frac{c_R}{(N-n)c_L + c_R})^2 c_R)$$

A parametric formula dependent upon the value of n, would give the exact cost executing the loop before it executes. With such a formula we could set the voltage before the loop executes and realize an energy reduction if the loop executes fewer than the maximum number of times. The energy consumed is approximately:

$$E_{\text{Ptype}} \approx (V\frac{n\,c_L + c_R}{N\,c_L + c_R})^2(n\,c_L + c_R)$$

We call this P-type (parametric) voltage scaling. It is easy to see that $E_{\text{Ptype}} \leq E_{\text{Ltype}} \leq E_{\text{NonScaled}}$.

Figure 2 depicts the approximate energy consumption of L-type versus P-type scaling for a loop with a maximum size of ten iterations ($N = 10$) and a normalized initial voltage of 1.0 ($V = 1.0$). For the case of a loop body of 100 cycles ($c_L = 100$) and a RWEC at the end of the loop of 100 cycles ($c_R = 100$) we see a benefit of P-type verses L-type scaling which is proportional to the difference $N - n$. The savings is not as great in the second graph of Figure 2 were the remaining work is ten times ($c_R = 1000$) that in the first figure since, in this case, the loop comprises a smaller percentage of the total code.

1 The extra cycles incurred by the L-type DVS call are not included in this equation.

2.3. P-Type Voltage Scaling Placement

Our parametric IntraVS method places P-type voltage scaling code at a point in the program where the bounds on (multiple) loop(s) are set at run time, see Figure 3, which may be very early in the path; for example, at the function call boundary when the loop bounds are determined by the parameter values passed to the task's subroutine. Parametric P-type voltage scaling requires the formation of a closed-form function, that is evaluated at run-time with run-time dependent values, to accurately calculate the RWEC of the code. The parametric RWEC is used to scale the frequency at a P-type voltage scaling point.

Figure 3 illustrates the insertion of P-type scaling code that evaluates a function $f(n, m)$ to set the voltage to the optimum level. The function $f(n, m)$ is a low-order multivariate polynomial in the run-time dependent values n, m, and (when applicable) any slack cycles s generated from the previous task executing more quickly than its static RWEC estimation. Note that one P-type scaling point can control the optimum voltage level of one or more loops in advance.

Allowing the P-type voltage scaling point to be moved up an placed early in the execution path, as shown in Figure 3, has the advantage that the voltage can be scaled more uniformly across multiple basic blocks. This is essential for both reducing the energy consumption and keeping DVS overhead low. The function f evaluated at the P-type voltage scaling point can be linear or higher order depending on a tradeoff between speed and the required accuracy of the run-time voltage scaling computed by f. The speed versus accuracy issue is discussed in Section 3.

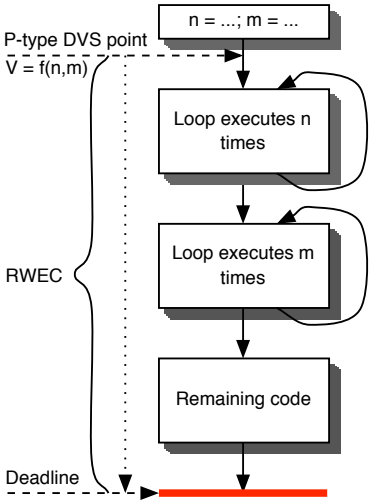


Figure 3. P-Type Dynamic Voltage Scaling

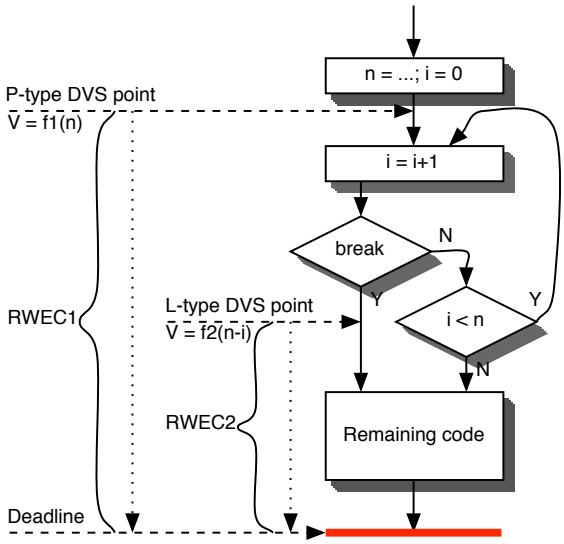


Figure 4. Combined P-Type and L-Type Dynamic Voltage Scaling

2.4. Combining L-Type with P-Type Scaling

The method we use to determine the symbolic loop iteration counts n and m is closely related to our WCET estimation methods [6, 4]. The analysis is relatively easy for most enumeration controlled loops, such as for-loops, that have one loop exit edge, i.e. they have no explicit break from the loop. These enumeration-controlled loops are the simplest and the most common form of loops.

However, logically-controlled loops and loops with explicit breaks require special treatment to exploit the unused cycles generated by an early break from the loop. An example of such a loop structure is:

```
n = ...;
...
for (i = 0; i < n; i++)
{ ...
  if (some_runtime_condition)
    break;
  ...
}
```

To reduce the energy consumption of a loop with multiple exits, we combine the P-type voltage scaling with L-type voltage scaling. However, unlike the approach by Shin et al. [15, 14] we place the L-type voltage scaling code only at the explicit break point branching out of the loop and only when the L-type scaling cost is lower than the cycle cost of one loop iteration execution, see Figure 4. The L-type voltage scaling code at the break point uses the number of remaining cycles calculated from `n-i` to evaluate a parametric scaling function f_2 to run the remaining code at a lower clock frequency, but only when the loop iteration counter `i` is not equal to `n`.

This is done by using the loop iteration counter to determine the minimum number of unused cycles at the break points branching out of the loop, which is $(n - i)$BCET, where BCET is the best case execution time of the loop body [5]. In this way, logically-controlled loops such as while-loops can be augmented with a loop iteration counter that is used to determine the number of unused cycles at the loop exit point(s). The loop counter kept in a register to avoid costly memory accesses.

This approach can be further enhanced if a cycle counter is present on the machine or a register is used to keep track of the actual cycles used throughout loop iterations. In such cases, we would be able to get an exact count of the cost of executing the loop. This cost can be subtracted from the assumed worst case cycle cost to determine the exact savings of exiting the loop early. This approach, which is not used by Shin et al. [15], would be most beneficial when the body of the loop has many paths with greatly varying execution cycles.

2.5. Combining B-Type with P-Type Scaling

Our parametric P-type voltage scaling placement is determined by the set-operations that affect the size of the iteration space of the loop(s). The position of these set-operations in the CFG is determined with use-def analysis [17]. In some cases the set-operations occur in the branches of if-statements. This requires the propagation of the parametric RWEC expressions up the CFG's decision tree. Furthermore, P-type voltage scaling code that is executed earlier in the path does not have access to these values that are available only later in the path. Therefore, a conser-

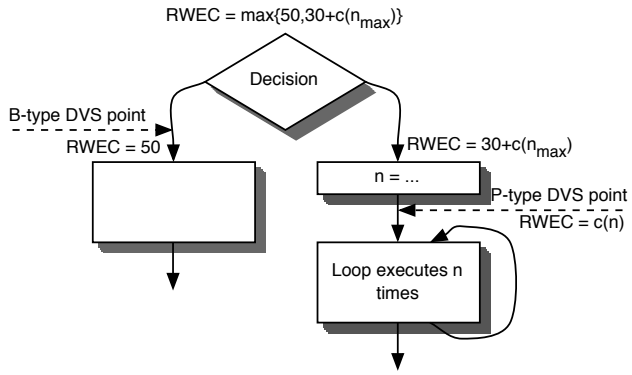


Figure 5. Combined B-Type and P-Type Dynamic Voltage Scaling at a Decision Point

vative bound $n_{\max}$ on the actual loop iteration space size $n \leq n_{\max}$ is used (see the CFG in Figure 5).

B-type voltage scaling code is executed in the least costly branch, similar to the work by Shin et al. [15]. The P-type voltage scaling code is executed in the other branch to uniformly control the clock frequency down this branch. In Figure 5, the parametric RWEC of the loops is denoted by $c(n)$, where n is the run-time iteration space size of the loop. At the P-type scaling point, the parametric function $f(n)$ is evaluated to set the amount of scaling.

Figure 6 depicts a CFG in which the use-def analysis found the set-operations that affect the iteration space size of the loop(s) before the decision point(s). In this case the P-type voltage scaling code will control the clock frequency of the most expensive path, which runs down into the second loop. The clock frequency is uniformly set for both loops with a scaling function $f(n, m)$. When either loop features explicit breaks, appropriate L-type voltage scaling codes are inserted at the loop exit points corresponding to the break operations, as discussed above. This ensures that unused cycles generated by breaks are exploited to reduce energy consumption. B-type voltage scaling code is added to the least costly branch to exploit any unused cycles down the least expensive path.

2.6. Placement Algorithm

The full parameteric IntraVS algorithm is as follows:

1. Generate the CFG of the task.
2. Perform static WCET and BCET analysis on the CFG's basic blocks based on the work by Healy et al. [5].

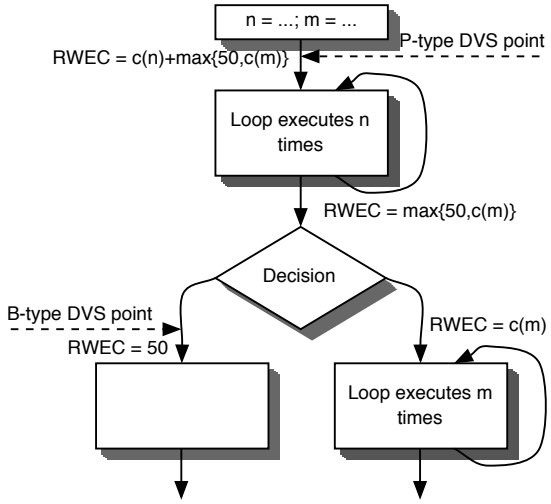


Figure 6. Combined B-Type and P-Type Dynamic Voltage Scaling at a Decision Point

3. Determine the loop iteration space size of each (nested) loop, including loops with unknown/symbolic bounds using parametric WCET analysis [4].
4. Generate an acylic condensation of the graph, while keeping information about the loop nesting structure.
5. Build the loop index table for the cost of each loop during a bottom-up traversal of the acylic condensation.
6. The loop index table has the following structure:
   (a) A loop reference number (starting from 0).
   (b) The current value of the cycle cost of the loop; this is initially set by evaluating the parametric formula for the loop with the maximum iteration values indicated by the user. If the loop has a break statement set the bit to indicate that the true loop value is known.
   (c) The parametric formula which represents the cost of the loop.
7. Using use-def analysis place code that updates loop expressions values, in the loop index table, when all of the dependent variables of the loop are known. Place voltage/frequency scaling code after such loop updates.
8. Place test code after if statement branches and test to see if voltage/frequency scaling calls need to be invoked.

3. P-Type Voltage Scaling Code

In this section we explain how the P-type voltage scaling code is generated from the determination of a closed-form polynomial RWEC cost function. This parametric RWEC function is essential to our approach. The strategic placement of the P-type voltage scaling code early in the execution path requires an accurate assessment of the cost of the code down the path(s).

3.1. Loop Analysis

Our P-type placement is decided by the use-def analysis [17] of variables used in loop bounds and in loop exit conditions. To determine the RWEC of a loop, the worst case execution time of a loop is calculated by static analysis of the code. To this end, a general polytope enumeration method can be used, such as Presburger formulas [12] or Erhart polynomials [2]. However, polytope enumeration methods require exponential time in the number of unknowns and are therefore expensive. Because most loops have a relatively simple structure, we use a fast symbolic counting method to determine the upper bound on the maximum number of iterations a loop executes [4]. An additional advantage of this counting method is that it can determine the execution cost of a loop nest, where a multi-variate polynomial function models the actual cost of the loop body.

3.2. Newton-Gregory Formulae

The Newton-Gregory formulae are used to calculate the parametric size of the iteration space of a loop nest. Most enumeration controlled loops in embedded system software have a relatively simple structure with loop bounds that are usually symbolic. Non-rectangular loops such as those found in FIR filters in speech codecs and streaming imaging software also exist. An example triangular loop is:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n - i; j++)
    ...
```

Figure 7 illustrates the loop structure of the example. Static analysis [5] is used to determine the worst case execution cycle (WCEC) of the basic blocks **B1** to **B7**. For this example, three cycle cost values are determined:

- c_0 is the WCEC of blocks **B1** and **B2**
- c_1 is the WCEC of blocks **B2**, **B3**, and **B7**
- c_2 is the WCEC of blocks **B4**, **B5**, and **B6**

Given these constants, the WCEC of this loop nest is a polynomial function $c(n)$:

$$c(n) = c_0 + (c_1 + c_2(n + \frac{1}{2}))\max(0, n) - \frac{1}{2}c_2 \max(0, n)^2$$

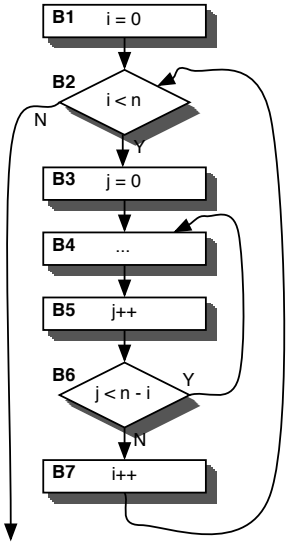


Figure 7. Example Triangular Loop Nest

where c_0 is the constant bound on the cycle cost of the outer loop header, c_1 is a constant bound on the cycle cost of the inner loop header, and c_2 is a constant bound on the cycle cost of the inner loop body.

The WCEC polynomial $c(n)$ is calculated with the Newton-Gregory formulae using the following definition.

Definition 3.1 *The (parametric) worst case execution cycles (WCEC) bound of a (possibly zero-trip) loop with iteration range $i = a, \ldots, b$ and stride $s \neq 0$ is*

$$WCEC = c_0 + \sigma(\max(0, \tfrac{b-a}{s} + 1))$$

where constant c_0 bounds the loop start cycles. The polynomial σ has coefficients $\sigma_0, \ldots, \sigma_{k+1}$ given by the matrix-vector product

$$\sigma(n) = [\sigma_0, \ldots, \sigma_{k+1}]_n = \mathbf{N}_{k+1}^{-1}\langle 0, \phi_0, \ldots, \phi_k\rangle_i$$

where $\Phi(i) = [\phi_0, \ldots, \phi_k]_i = \mathbf{N}_k[p_0, \ldots, p_k]_i$ is the Newton series representation of the polynomial $p(i) = [p_0, \ldots, p_k]_i$ over $i = 0, \ldots, \lfloor \frac{b-a}{s} \rfloor$ that bounds the execution cycles of the normalized loop header and body operations. Matrices $\mathbf{N}$ and $\mathbf{N}^{-1}$ are the Newton triangle and its inverse, respectively.

Given the constants c_0, c_1, and c_2, we derive the $c(n)$ formula of the above example as follows:

$$
\begin{aligned}
&WCEC = c(n) \\
&= c_0 + \sigma(\mathbf{N}_1(c_1 + \sigma(\mathbf{N}_0 c_2, \max(0, n-i))), \max(0, n)) \\
&\quad (0 \le i \le n-1, \text{so replace } \max(0, n-i) \text{ by } n-i) \\
&= c_0 + \sigma(\mathbf{N}_1(c_1 + \sigma(\mathbf{N}_0 c_2, n-i)), \max(0, n)) \\
&= c_0 + \sigma(\mathbf{N}_1(c_1 + \sigma(\langle c_2 \rangle_i, n-i)), \max(0, n)) \\
&= c_0 + \sigma(\mathbf{N}_1(c_1 + c_2 n - c_2 i), \max(0, n)) \\
&= c_0 + \sigma(\langle c_1 + c_2 n, -c_2 \rangle_i, \max(0, n)) \\
&= c_0 + (c_1 + c_2(n + \tfrac{1}{2}))\max(0, n) - \tfrac{1}{2} c_2 \max(0, n)^2
\end{aligned}
$$

This formula is accurate given that the constants c_0, c_1, and c_2 are tight bounds. The constant bounds are tight when the target architecture's cycle count can be accurately determined with static analysis on the basic blocks. Static analysis is used to determine the cycle cost of executing the basic blocks of the program [5].

When the loop bound n in the loop nest is constant or can be determined by constant value propagation in the CFG, the WCEC value of the loop nest is constant. When this constant is small, the loop bound might be improved with simulation of the loop rather than static analysis. This aspect will be investigated in future work.

3.3. Parametric Voltage Scaling Placement

In this section, we look at our voltage/frequency scaling code placement. We are interested in placement of code at L-type edges, B-type edges and P-type edges. The discussion is motivated by looking at the placement of DVS code in the example in Figure 5.

In this example we have a $\mathrm{RWEC} = max(50, 30 + c(n_{\max})$. The compiler will know, either from analysis or an user's annotation, the value of $n_{\max}$. From this, the maximum of the two values can be determined and the exact RWEC at the loop is known. If this maximum is greater than 50 cyles plus the overhead cycles of the DVS call, a DVS call is placed at the B-type edge.

The situation is more complicated in the case of a P-type DVS point. In such situations, the actual cost of the loop is not known until the last dependent variable actual value is known. This occurs at the P-type edge. Therefore, the computation, which determines whether voltage scaling is cost effect, must occur at this edge. The saved cycles are the difference between the loop(s) cost before and after the last loop(s) dependent variable is known. In our particular case, the saved cycles will be equal to $WCET(l) * (c(n_{\max}) - c(n))$; where $WCEC(l)$ is the worst case execution cycles for the loop body. If this difference is greater than the overhead of voltage/frequency scaling we insert a DVS call at this point.

3.4. Voltage/Frequency Scaling Functions

Like Shin, et al., [15, 14] it is assumed that we can scale the voltage/frequency by a system call, called $updateVF(new\_frequency)$. The system call changes the frequency and voltage accordingly, and it is assumed that the frequency range in which we can set the CPU clock is a continuous range between $0Mhz$ and the maximum clock frequency for the CPU. It is assumed that the program does not execute during the execution of the call.

The clock frequency is set to allow the program to finish executing exactly at its deadline. Assuming that the voltage/frequency scaling occurs at the old clock frequency we have the following relationship.

$$frequency_{new} = frequency_{old} * \frac{\mathrm{RWEC}_{current}}{\mathrm{RWEC}_{old} + scaling\_overhead}$$

For the P-type edge in Example 5 we therefore have the following value for the updated frequency

$$frequency_{new} = frequency_{old} * \frac{c(n)}{c(n_{max}) + scaling\_overhead}$$

.

4. Conclusions and Future Work

Our method uses static analysis and tight bounds on loop iteration counts to guide the placement of intra-task dynamic voltage scaling code by the compiler. We break the program into basic blocks and use static analysis to determine the cycle cost of these basic blocks. We generate closed-form formulae that tightly bound the cost of loops. Because we use formulae, which represent the cost of loops, we can potentially know the cost of loops before, not after the loops execute. Unlike previous work on this subject [15] [14], this allows voltage scaling code to be moved earlier in the program which may reduce overhead and allow the running of the program at a lower voltage/frequency for a longer period of time.

Current work on our Newton-Gregory WCET technique could be used to more effectively estimate the cost of loops with many branch statements in their bodies. We are currently working on a technique to tightly bound many different curves, which represent different control flow paths in a loop, by a single curve. This could allow the removal B-type DVS calls from inside of a loop body.

The profiling-based approach by Shin et al. [14] runs code using an average case execution path (ACEP) as a reference path instead of the worst case execution path. This approach may also require increasing clock frequency at certain points in the task to ensure that the task deadline

is satisfied when part of the code requires more time to execute compared to the reference path. To guarantee satisfying a deadline on time, voltage scaling is conservatively applied by ensuring that the static RWEC determined by the worst case execution path does not exceed a certain threshold. Similarly, profile-based IntraVS requires accurate RWEC estimation, so we argue that the parametric approach is also valuable to improve profile-based IntraVS methods.

References

[1] N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven. Energy management for real-time embedded applications with compiler support, 2003. ACM SIGPLAN Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'03), June 2003.

[2] Philippe Clauss and Vincent Loechner. Parametric Analysis of Polyhedral Iteration Spaces. In *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*. IEEE Computer Society, August 1996.

[3] A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback edf scheduling for embedded systems with real-time constraints, 2002. ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02), June 2002.

[4] R. Van Engelen, K. Gallivan, and B. Walsh. Tight timing estimation with newton-gregory formulae. In *Proceedings of the 2003 Compilers for Parallel Computers*, pages 321–329, 2003.

[5] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pages 288–297, 1995.

[6] Christopher A. Healy, Mikael Sjodin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[7] Seongsoo Lee and Takayasu Sakurai. Run-time voltage hopping for low-power real-time systems. In *Design Automation Conference*, pages 806–809, 2000.

[8] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. *Software Engineering*, 21(7):593–604, 1995.

[9] Jiong Luo and Niraj K. Jha. Battery-aware static scheduling for distributed real-time embedded systems. In *Design Automation Conference*, pages 444–449, 2001.

[10] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. 2000.

[11] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[12] William Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 1994.

[13] Daler Rakhmatov and Sarma Vrudhula. Battery-conscious task sequencing for portable devices including voltage/clock scaling.

[14] D. Shin and J. Kim. A profile-based energy-efficient intratask voltage scheduling algorithm for hard real-time applications, 2001.

[15] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Design Automation Conference*, pages 438–443, 2001.

[16] Youngsoo Shin and Kiyoung Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Design Automation Conference*, pages 134–139, 1999.

[17] M. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesly, 1996.