Profile-based Dynamic Voltage Scheduling using Program Checkpoints

Ana Azevedo, Ilya Issenin, Radu Cornea
Rajesh Gupta, Nikil Dutt, Alex Veidenbaum, Alex Nicolau
Center for Embedded Computer Systems
University of California, Irvine
444 Computer Science Building, Irvine, CA 92697-3425
{aazevedo, isse, radu, rgupta, dutt, alexv, nicolau}@ics.uci.edu

Abstract

D*ynamic* **v***oltage* **s***caling (DVS) is a known effective mechanism for reducing CPU energy consumption without significant performance degradation. While a lot of work has been done on inter-task scheduling algorithms to implement DVS under operating system control, new research challenges exist in intra-task DVS techniques under software and compiler control. In this paper we introduce a novel intra-task DVS technique under compiler control using program checkpoints. Checkpoints are generated at compile time and indicate places in the code where the processor speed and voltage should be re-calculated. Checkpoints also carry user-defined time constraints. Our technique handles multiple intra-task performance deadlines and modulates power consumption according to a run-time power budget. We experimented with two heuristics for adjusting the clock frequency and voltage. For the particular benchmark studied, one heuristic yielded 63% more energy savings than the other. With the best of the heuristics we designed, our technique resulted in 82% energy savings over the execution of the program without employing DVS.* [1]

1. Introduction

With today's processors speed reaching gigahertz, the inherent power dissipation level of the order of tens of Watts becomes an important concern in digital design. The dynamic power dissipation for CMOS circuits has a quadratic dependency on the supply voltage V_{DD} ($P_{dynamic} \propto C{V_{DD}}^2 f$, where C is the collective switching capacitance and f is the common switching frequency). Therefore dynamic voltage scaling techniques, by lowering the supply voltage, are effective in reducing CPU power. The main disadvantage of dynamic voltage scaling is its power-performance tradeoff. The supply voltage also determines the CMOS circuit delay τ ($\tau \propto V_{DD}/(V_{DD} - V_T)^{\alpha}$, where V_T is the threshold voltage and α is a velocity saturation index). Lowering the supply voltage increases the circuit delay and causes a decrease in the speed supported by the circuit. As a consequence, dynamic voltage scheduling techniques must carefully consider the resulting impact on system performance. Throughout this paper we will refer to both terms *frequency scaling* and *voltage scaling* interchangeably, meaning that changes to frequency are accompanied by appropriate adjustments to voltage.

Early related work on dynamic voltage scheduling includes theoretical studies [9] and simulations on the potential of DVS techniques [7, 12, 14, 15, 24]. Since then, practical implementation of DVS schemes have already been proved feasible with some commercial processors [1, 6] and academic design efforts [11, 16, 17]. Underlying software support, mainly at the operating system level, includes the development of several static/dynamic *inter-task* [8, 9, 13, 18, 21, 23, 25] and *interval-based* DVS techniques [7, 16, 22, 24]. To scale voltage, these techniques either make use of a prior knowledge of the application workloads or produce predictions for the application demands based on past history. Energy savings come from exploiting the slack from the **w**orst **c**ase **e**xecution **t**ime (WCET) [11]. WCET slack time is the extra time produced as the result of tasks actually running faster than the pessimistic worst case speed assumed by the heuristics. Another type of time slack is exploited by *intra-task* DVS techniques under software and compiler control [4, 5, 10, 11, 19, 20]. Called *workload-variation slack time* [11], it results from the observation that within an individual task boundary the execution time may change significantly depending on the executed program path, representing other opportunities to apply dynamic voltage scaling.

In our studies we observed that most of the above mentioned techniques rather view a whole application program

[1]This research is partly supported by DARPA PAC/C program under contract number F336-15-00 -C-1632.

Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)
1530-1591/02 $17.00 © 2002 **IEEE**

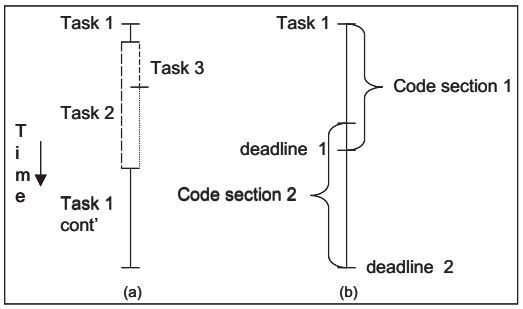


Figure 1. Example of applications with overlapping time constraints.

as a single task. They do not handle multiple tasks specified in the *same* application program that together might result in multiple deadlines depending upon the executed control flow. However, in practice, real applications resort to such computing constructs. As an illustration of the type of time constraints our approach to DVS can handle, consider an application with three time-constrained tasks shown in Figure 1 (a). The program begins executing *Task*1. At some point in *Task*1, two alternative tasks, *Task*2 and *Task*3, might be executed. After either *Task*2 or *Task*3 is executed, *Task*1 can resume its execution. The scheduling problem is how to set clock frequency and voltage so that the time constraints on the two tasks (*Task*1 and *Task*2 or *Task*1 and *Task*3) are both satisfied, saving energy as most as possible.

Other difficulties arise in simply dividing a program into tasks, the unit of computation used in current DVS techniques, when time constraints need to be specified in less than usual ways, as for example the time constraint dependencies for the different code sections depicted in Figure 1 (b). We believe the flexibility in specifying time constraints (or task deadlines) is crucial in any DVS technique. In this work we address this issue and we propose an intra-task DVS technique based on *program checkpoints*. In our scheme, program checkpoints carry time constraint information and we allow code sections within a task to have different deadlines. In contrast, Shin et. al. [20, 19] approach to intra-task DVS fixes the time constraint at task level. We further extend the use of program checkpoints to include new checkpoints in the code with the purpose of exploiting other slack time opportunities that arise at run-time due to workload variations.

The other aspect that distinguishes our work from previous research is the fact that, besides optimizing for average power, we consider that at run-time we operate with a limiting power budget. DVS decisions must take into account an *available power profile* that may vary over time. The available power profile is used as an input to our techniques and may be produced by an operating system which controls the

```
foo(){
read(i);
if (i  > 5) {
  i = i - calc_new_i(i);
} else
  a++;
}
i = 36;
for (j = 0; j < i, j++) {
  k = k*sin(j/100 + k/10);
}
}

calc_new_i(int l){
  for (k = 0; k < limit, k++){
    i += new_i[k];
    show_value(i);
  }
}
```

(a) Original code.

```
foo(){
read(i);
CHECKPOINT(1);
if (i  > 5) do {
  i = i - calc_new_i(i);
} else {
  a++;
}
i = 36;
k = i + a;
CHECKPOINT(2);
for (j = 0; j < i, j++) {
  CHECKPOINT (3);
  k = k*sin(j/100 + k/10);
}
CHECKPOINT(4);
}
```

(b) Transformed *foo* code with checkpoints 1, 2, 3 and 4 carrying time constraints.

CDB		
Checkpoint Transition	**Min Time (ms)**	**Max Time (ms)**
1-2	10	30
2-3	20	20
3-3	50	200
3-4	200	200

(c) Checkpoint Database (CDB).

Figure 2. Code example with program checkpoints imposing time constraints.

available power sources in the system.

This paper is organized as follows. Section 2 introduces a description of program checkpoints. It also briefly discusses the details of our previous DVS algorithm [3] that handles multiple deadlines within a task by using program checkpoints. Section 3 explains how to place extra checkpoints into the application code to better exploit the workload-variation slack time, followed by the description of our new slack-based DVS algorithm. The COPPER power-performance simulation framework [3] was used to obtain the experimental results on our case study, which are summarized in Section 4. Finally Section 5 concludes the paper with future work plans.

2. DVS using Program Checkpoints

A checkpoint represents a specific location in the code marked by a statement label. A checkpoint transition is a direct control flow path between two checkpoints. Figure 2 (b) shows an example of a code with four checkpoints and four possible checkpoint transitions: transition 1-2 comprises all the instructions from the beginning of the if-then-else statement up to the for-loop statement; transition 2-3 includes the for-loop header; transition 3-3 corresponds to the code fragment for one loop iteration; and

1. Create a list of events with the time intervals the events occur.
2. Calculate frequency limit f_{limit} for each of the time intervals in the list of events.

$$f_{limit} = \sqrt[3]{power\_limit * base\_freq^2 / max\_power\_per\_cycle}$$

power_limit = maximum dissipated power derived from run-time power constraint

3. Calculate optimal frequencies and create a schedule for changing frequencies and voltages.

Case 1: Only two checkpoints, frequency limit is high enough.

$$Optimal\_frequency = \frac{profiled\_number\_of\_cycles}{max\_time\_constraint\_between\_checkpoints}$$

Case 2: Only two checkpoints, frequency limit is *not* high enough.

$$Optimal\_frequency = \begin{cases} freq\_limit \\ \frac{profiled\_number\_of\_cycles - limited\_number\_of\_cycles}{max\_time\_constraint - limited\_time} \end{cases}$$

Case 3: Several possible future checkpoints.

For the intervals in the list of events do:

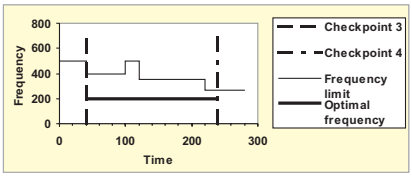
1. *Calculate Optimal_frequency for curr interval, for all checkpoint transitions using Case 1, 2.*
2. *Optimal_frequency for curr interval = max (Optimal_frequency for all checkpoint transitions).*

In all cases voltage is adjusted proportionally to changes in frequency.

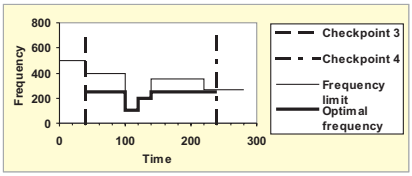
$$Optimal\_voltage = \frac{baseline\_voltage * Optimal\_frequency}{baseline\_frequency}$$

4. Execute the program, changing frequencies and voltages according to the created schedule.
5. As soon as the next checkpoint is encountered, discard the list of events. Go to step 1.

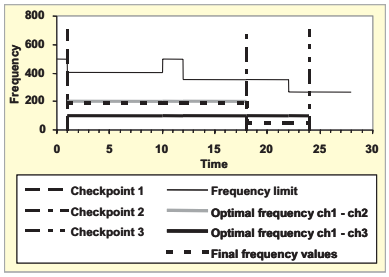
Figure 3. DVS algorithm using program checkpoints.



(a) Calculating optimal frequency, Case 1.



(b) Calculating optimal frequency, Case 2.



(c) Calculating optimal frequency, Case 3.

Figure 4. Optimal frequency calculation.

transition 3-4 represents the control flow taken to exit the for-loop. In our simulation framework, user-defined checkpoints are inserted in the source code and compiled into special machine instructions.

Time constraints are set for the execution of the piece of code between checkpoints in terms of acceptable lower and upper bounds (Min Time, Max Time). Such information is stored in a **c**heckpoint **d**ata**b**ase (CDB), along with the possible checkpoint transitions derived from the program control flow. A CDB example is shown in Figure 2 (c).

An initial version of our DVS algorithm using program checkpoints has been presented in [3]. As this algorithm serves as basis for our new slack-based DVS technique, a brief explanation of the former is given in this section.

Our heuristic for scheduling with power and time constraints is divided in two phases: an *ahead of time profiling* phase and the final *run-time power scheduling* phase. Prior to either of these phases program checkpoints can be inserted anywhere in the application program. Time constraints must be imposed for each checkpoint transition and cannot be specified at any other place. In the profiling phase a **c**heckpoint **p**ower profile **d**ata**b**ase (CPDB) is built. We profile the program using a representative input data set and collect minimum and maximum energy/power dissipated and cycle count for checkpoint transitions. The goal of the run-time power scheduling phase is to follow, in an energy-efficient way, a run-time power profile which represents the available power budget, while simultaneously meeting the time constraints. The power scheduler dynamically adjusts voltage and clock frequency values at program checkpoints. Other *events* at which the scheduler might adjust values include points of abrupt changes in the available power profile and times the scheduler identifies an expected control flow path was not taken (i.e., an expected checkpoint transition did not take place). All these points in time are stored in a structure named *list of events*.

The main phases of the DVS algorithm are described below. The complete algorithm is outlined in Figure 3.

Step 1: Create list of events

The scheduler creates a list of all potential events that might occur from the current checkpoint in execution up to the farthest deadline of all its possible checkpoint transitions.

Step 2: Calculate frequency limit

The frequency limit f_{limit} is the maximum frequency the program should run to dissipate power below the imposed power limit. To calculate f_{limit} values the scheduler uses the CPDB and CDB information, checking the available power (*power_limit*) and the maximum profiled power (*max_power_per_cycle*) consumed from the checkpoint in consideration to all its possible transitions to other checkpoints. The frequency limit is calculated by a formula

which is based on the fact that dynamic voltage scaling combines two CMOS design equations: $energy \propto voltage^2$ and $frequency \propto voltage$.

Step 3: Calculate optimal frequency
The optimal frequency is the frequency the program should run to satisfy both power *and* time constraints. Upon obtaining the frequency limit, the scheduler calculates the range of frequencies the code can be run to satisfy the time constraint upper bounds. In this calculation it uses the CDB information on imposed time constraints for all possible checkpoint transitions from the checkpoint in consideration. It also makes use of the profiling information in the CPDB. At this phase three situations may arise.

In the case there is only one future checkpoint transition (Case 1 in Figure 3, graphically illustrated in Figure 4 (a)), we calculate a potential $Optimal\_frequency$ value by dividing the profiled number of cycles for the checkpoint transition by the maximum time allowed for this transition in the CDB. If this value is less than or equal to f_{limit} calculated in Step 1, then $Optimal\_frequency$ is set to this value. The second case (Case 2 in Figure 3, sketched in Figure 4 (b)) still handles the situation in which there is only one possible future checkpoint transition. However, in this case, the frequency limit values calculated in the previous phase, for some of the events intervals that happen within the checkpoint transition interval, are lower than the potential $Optimal\_frequency$ values calculated by Case 1. For such intervals the $Optimal\_frequency$ is fixed to f_{limit}. The $Optimal\_frequencies$ for the remaining intervals are calculated just like in Case 1, after counting off the cycles (and time) spent executing at f_{limit} speed. In the third case (Case 3 in Figure 3, depicted in Figure 4 (c)), several possible checkpoints can be reached from the checkpoint in consideration. Each checkpoint transition is then handled as either Case 1 or Case 2. After calculating the $Optimal\_frequency$ values for each checkpoint transition, the scheduler selects the maximum $Optimal\_frequency$ among the values. It keeps adjusting the $Optimal\_frequency$ as program paths are executed, always selecting an $Optimal\_frequency$ value that makes the code run as slow as possible within the time constraints. For example, in Figure 4 (c), after checking that there is no checkpoint occurrence after time 18, the scheduler assumes the transition 1-2 did not occur and lowers the speed for the execution of the remaining code until checkpoint 3 is executed. For all cases, after calculating the $Optimal\_frequency$ values, $Optimal\_voltage$ values are computed by scaling the voltage proportionally to the changes in the clock frequency.

3. Slack-based DVS using Program Checkpoints

Our original DVS algorithm in Section 2 calculates the clock frequency and voltage using the maximum profiled cycle counts. At runtime the actual executed path may differ from the profiled one and the number of cycles for executing a code section may be lower. This means that the calculated clock frequency, voltage and the resultant energy consumption may be higher than necessary. In this section we explore how the use of additional checkpoints may improve on this situation. A **h**ierarchical **c**ontrol **f**low **g**raph (HCFG) program representation is used to both simplify the addition of new checkpoints and the bookkeeping of time information related to the checkpoints. The new version of the algorithm also features further flexibility for specifying time constraints. Time constraints can be specified between any checkpoint pairs defined in the same level of the HCFG of the application program.

Program checkpoints are initially inserted at every branch, loop and function call. During the compilation of the application program we generate a HCFG in which nodes are labeled after the checkpoint that initiates the code section contained within the node. Each node has a type and nodes of type $function\ call$ and $loop\ header$ are associated with a sub control flow graph. All other nodes are of type *normal*. Figure 5 shows how the original code in Figure 2 (a) is transformed to include checkpoints. Figure 5 (b) shows the corresponding HCFG. Maximum time constraints for the code example are specified in Figure 5 (c).

A profiling phase collects the maximum and minimum number of execution cycles for each node and the maximum number of iterations for each loop. All this information is stored with the corresponding HCFG node and a CPDB is built.

To reduce the inherent run-time overhead associated with scaling frequency and voltage we prune checkpoints and merge HCFG nodes that satisfy the following conditions: (1) nodes with low maximum execution cycle count or (2) nodes with small variation in the execution cycle count. Nodes that cannot be merged are those that: (1) are at the beginning or end of a time constraint interval or (2) have time constraints in their associated sub CFG.

The dynamic power scheduling phase is executed as outlined by the algorithm in Figure 6. In the original algorithm from Section 2, time constraints for checkpoint transitions are stored in a CDB. This information, together with the profiling data in a CPDB are used to calculate the $Optimal\_frequency$. In the new version of the algorithm, a *new CDB* is built every time a checkpoint is encountered and is used as input to the algorithm described in Section 2. In this case, the CDB contains only the *active*

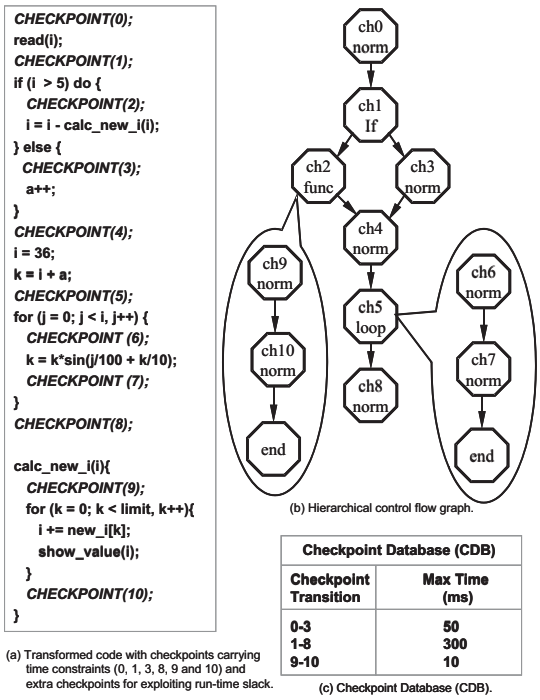


Figure 5. Code example with program checkpoints imposing time constraints and exploiting runtime slack.

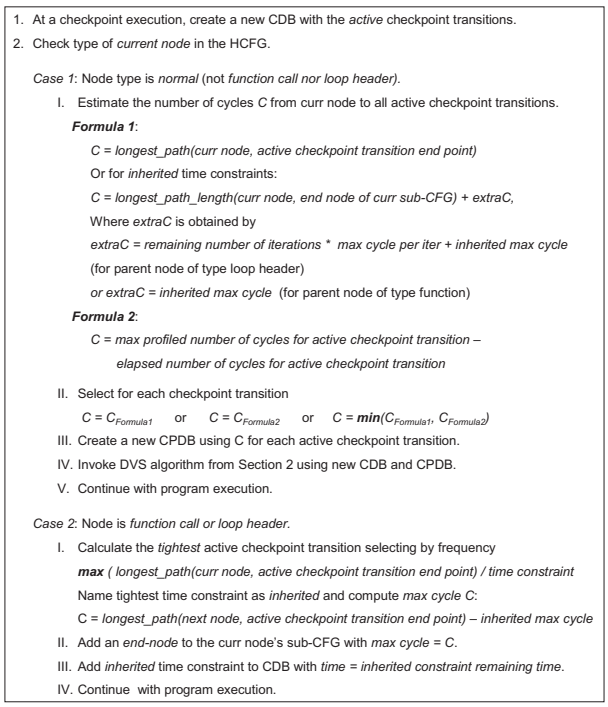


Figure 6. Slack-based DVS algorithm using program checkpoints.

time constraints (or *active* checkpoint transitions) which are the deadlines for the program paths that the current node in execution belongs to (Step 1 in Figure 6). These program paths may either belong to the sub control flow graph that contains the current node or to the higher level control flow graph. In the latter case, we call such deadline an *inherited* time constraint. For easier management of inherited time constraints, a sub control flow graph is augmented with an *end node* that carries information about the tightest time constraint from the higher level CFG. All this data is produced through a reachability analysis on the HCFG which will be explained shortly. Different algorithmic steps are taken depending on the type of the control flow graph node and are discussed below.

Normal nodes: *Normal* nodes do not have sub control flow graphs associated with them. When processing a *normal* node we first make a *new estimation* of the *remaining number of cycles* for each *active* checkpoint transition from the current node (Case 1, Step I in Figure 6). This is similar to constructing a *new* CPDB at run-time. We have experimented with two heuristics for calculating the cycle count, labeled *Formula*1 and *Formula*2 in Figure 6. Under *Formula*1, for each active checkpoint transition the cycle count is given by the length of the longest path from the current node to the end point of the checkpoint transition. For the particular case when the time constraint is inherited from a higher level CFG, the cycle count is determined by the longest path as above, plus the cycle count contribution from the added end node in the current sub CFG, and plus the contribution from the remaining iterations (for loop bodies). The second heuristic for estimating the cycle count keeps track of the cycles elapsed since the checkpoint transition in consideration started to execute. Subtracting this value from the maximum profiled number of cycles for the checkpoint transition under consideration, we obtain the predicted remaining number of cycles via *Formula*2.

We experimented with these two heuristics and we noticed that the best estimation may come from either of them and is very dependent on the run-time program behavior. Therefore, we selected as the best estimation the *minimum* number of cycles obtained by *Formula*1 and *Formula*2 (Case 1, Step II). We create a new CPDB and update the existing CDB with the new estimated values for checkpoint transition cycle counts and time. Finally, the original DVS algorithm from Section 2 is invoked and program execution continues (Case 1, Steps III, IV and V).

Function call or Loop header nodes: Upon the execution of instructions in a node of type *function call* or *loop header*, the first step is to identify the tightest time con-

straint in the current CFG and convey this information to the sub CFG associated with the current node. The tightest time constraint is determined by choosing the maximum among the estimated minimum necessary frequency for executing each active checkpoint transition satisfying the time constraint. This in turn is calculated as the number of cycles in the longest path of execution between the current node and the end point of the active checkpoint transition in consideration divided by the remaining time constraint associated with this checkpoint transition (Case 2, Step I in Figure 6). Information on this inherited time constraint is stored in a special *end node* in the sub control flow graph and is added to the CDB (Case 2, Step II and III). After these steps, program execution continues.

Note that the tasks of creating a CFG, finding reachable nodes and of finding the longest path in a HCFG can be pre-computed. Most of the run-time processing is spent on recomputing cycle count estimates and updating the data structures used by the algorithm: CDB, CPDB and HCFG.

4. Experimental Results

Using the COPPER simulation framework [3] we evaluated our slack-based algorithm. In our simulation framework we make the following assumptions. When decoding a checkpoint operation a power scheduler module, which may be implemented within the operating system, is invoked. This module determines the new operating clock frequency and voltage using the DVS algorithms from Section 2 and 3. An alternative implementation would directly insert extra code for calculating the new speed into the application program. This solution is also used by earlier approaches [20, 19]. Our goal in this section is to show our algorithm *at work*. We do not take into account the frequency and voltage scaling time overhead nor the time and power overhead of the code for determining the new frequency and voltage values. However, for checkpoint transitions that take much longer than the typical frequency and voltage scaling overhead, the latter can be considered negligible. We simulate a MIPS-based out-of-order processor with configuration 600MHz-2.2V as baseline. We assume that frequency and voltage can vary continuously, and that when reducing the clock frequency, the voltage is scaled proportionally to changes in the clock speed.

Figure 7 shows the power profile before dynamic power management for *paraffins* benchmark [2], highlighting the points in time where checkpoints carrying time constraints are executed. Checkpoint transition 4-7 is the code section for which we detected a high variation on the number of execution cycles for different executions of the same program path. This piece of code consists of nested loops in which the number of iterations in the inner loops depends on the iteration variable of the outer loops. We imposed a time constraint of 1.5 milliseconds on this checkpoint transition.

Crucial for the success of our DVS technique is the accurate run-time estimation of the remaining number of cycles for active checkpoint transitions (Case 1 in Figure 6). We experimented with different heuristics to identify the one that consistently gives the most energy-efficient DVS solution. Two of these experiments are described below.

From the time constraint on checkpoint transition 4-7 and the available power profile shown in Figure 8, we obtain the frequency limit and optimal frequency values plotted in Figure 9. The gradual decline in steps observed for a couple of time intervals (100-200, 300-400, 400-600 and 600-800 microseconds) shows the adjustment of frequencies within loop iterations and with the repetition of loop bodies. As time progresses, the remaining number of cycles and remaining time constraint for the execution of the loop decrease, requiring lower frequencies as we approach the end of the loop execution. The resultant power profile after power management is shown in Figure 8.

The frequency values in Figure 9 were determined by estimating the remaining number of cycles for the active checkpoint transitions using *Formula*1 in Figure 6. We observed energy savings of 52% over the original program execution in Figure 7. In Figure 11 we used an alternative heuristic to estimate the remaining cycle counts. We selected the *minimum* value between the estimation given by *Formula*1 and *Formula*2. As can be noticed, the frequencies are much lower than the values shown in Figure 9, and are still satisfying the time constraints. In fact, the calculated frequency is the same, independent of the loop iteration or loop body repetition. This second heuristic led to an optimal frequency for the code execution that dismissed the need for further clock frequency and voltage changes. Comparing Figure 10 to Figure 8, the energy savings difference between the two DVS solutions is 63%. The heuristic based on the minimum value saves 82% more energy than the original case in Figure 7.

We further experimented with the same program under different input conditions and observed very different results for the distinct heuristics. However, the heuristic based on the minimum value resulted as the most efficient.

5. Conclusions

This paper presented an intra-task DVS algorithm that differs from existing ones in the flexibility for specifying time constraints via program checkpoints. We consider this feature relevant given the types of important applications that cannot be modeled to be optimized by current DVS techniques. We also designed a DVS technique that responds to runtime power constraints. In situations where the power budget is very limiting, a power limit can be im-

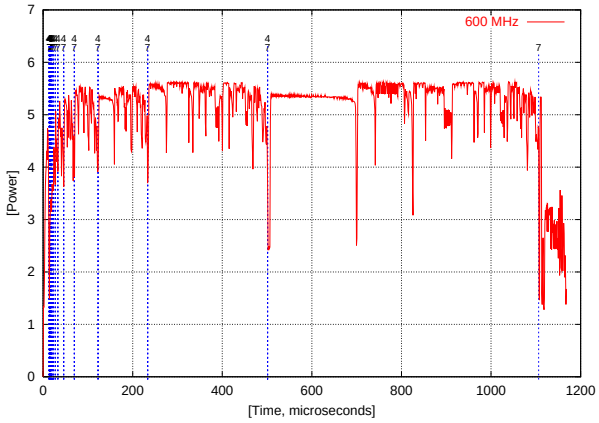


Figure 7. Power consumption profile highlighting checkpoint execution times without DVS for *paraffins*.

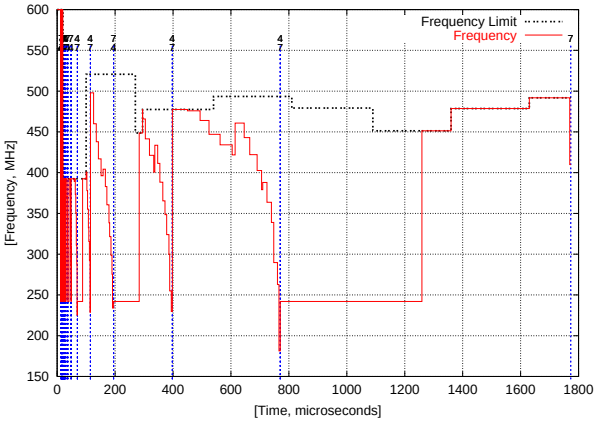


Figure 9. Frequency limit and optimal frequency using *Formula*1 for *paraffins*.

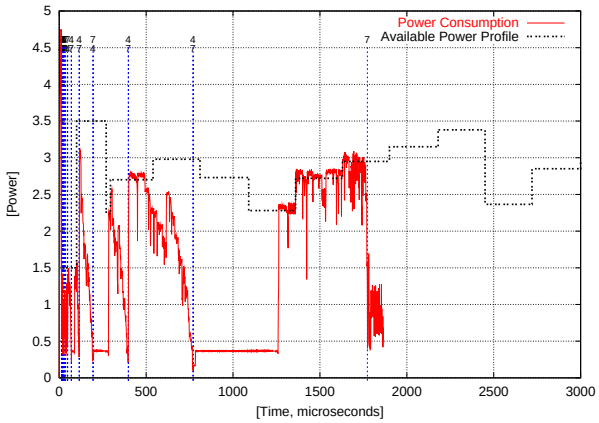


Figure 8. Available power budget and power consumption profile after applying slack-based DVS using *Formula*1 for *paraffins*.

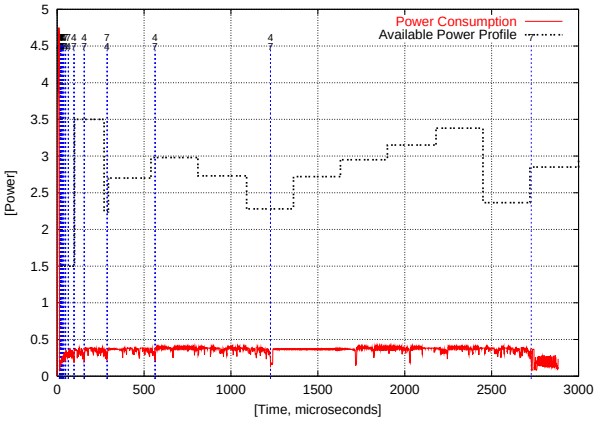


Figure 10. Available power budget and power consumption profile after applying slack-based DVS using *Formula*2 for *paraffins*.

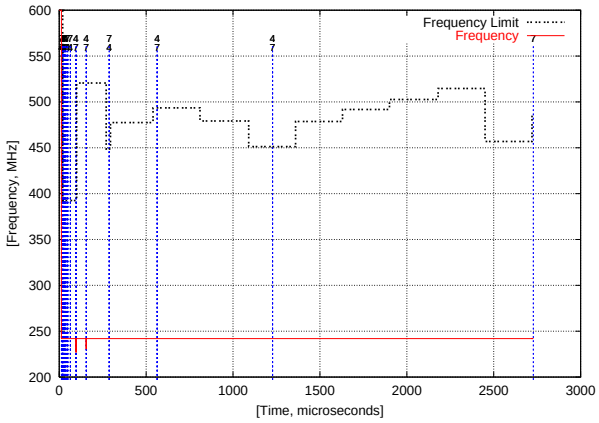


Figure 11. Frequency limit and optimal frequency using *Formula2* for *paraffins*.

posed and deadlines be missed if performance is not a concern. Our DVS technique needs refinement, e.g., modifications to take into account the power and time overhead for the extra run-time voltage scheduling operations and the time overhead of frequency and voltage scaling itself. The high energy savings we obtained over the program execution without DVS (82% for *paraffins*) motivates us to further exploit the technique.

References

[1] Intel XScale microarchitecture. http://developer.intel.com/design/intelxscale.

[2] Trimaran Project. http://www.trimaran.org/status.html.

[3] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Architectural and compiler strategies for dynamic power management in the COPPER project. *International Workshop on Innovative Architecture*, Jan. 2001.

[4] U. K. C.-H. Hsu and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Driven Microarchiteture*, June 1998.

[5] U. K. C.-H. Hsu and M. Hsiao. Compiler-directed dynamic voltage and frequency scheduling for energy reduction in microprocessors. *ISLPED*, pages 275–278, Aug. 2001.

[6] M. Fleischmann. Crusoe power management: Reducing the operating power with LongRun. 2000.

[7] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting on a low-power CPU. In *Proceedings of the First Annual Int'l Conf. on Mobile Computing and Networking*, pages 13–25, Nov. 1995.

[8] I. Hong, D. K. ad G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable-voltage core-based systems. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 18(12), December 1999.

[9] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. *ISLPED*, pages 197–202, Aug. 1998.

[10] C. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. *Sixth Real-Time Technology and Applications Symposium (RTAS)*, May 2000.

[11] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. *37th Design Automation Conference*, pages 806–809, 2000.

[12] Y.-H. Lee and C. M. Krishna. Voltage clock scaling for low energy consumption in realtime embedded systems. *Proceedings of the Sixth Int'l Conf. on RealTime Computing Systems and Applications*, 1998.

[13] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. *Int'l Conf. on Acoustics, Speech and Signal Processing*, June 2000.

[14] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. *Workshop on Compiler and OS for Low Power*, Oct. 2000.

[15] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED*, pages 76–81, Aug. 1998.

[16] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *ISLPED*, pages 96–101, July 2000.

[17] J. Pouwelse, K. Langendoen, and H. Sips. Voltage scaling on a low power microprocessor. *Int'l Symposium on Mobile Multimedia Systems and Applications (MMSA)*, Nov. 2000.

[18] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. *Int'l Symposium on Mobile Multimedia Systems and Applications (MMSA)*, Aug. 2001.

[19] D. Shin and J. Kim. A profile-based energy-efficient intra-task voltage scheduling algorithm for hard real-time applications. *ISLPED*, 2001.

[20] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, Mar. 2001.

[21] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. *Int'l Conf. on ComputerAided Design (ICCAD)*, pages 365–368, Nov. 2000.

[22] A. Sinha and A. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. *14th Int'l Conf. on VLSI Design*, Jan. 2001.

[23] V. Swaminathan and K. Chakrabarty. Investigating the effect of voltage switching on low-energy task scheduling in hard real time systems. *Asia South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2001.

[24] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, Novermber 1994.

[25] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. *IEEE Symposium on Foundations of Computer Science*, pages 374–382, Oct. 1995.