Collaborative Operating System and Compiler Power Management for Real-Time Applications [*]

Nevine AbouGhazaleh, Daniel Mossé, Bruce Childers, Rami Melhem, and Matthew Craven
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{*nevine, mosse, childers, melhem, mcraven*}*@cs.pitt.edu*

Abstract

Managing energy consumption has become vitally important to battery operated portable and embedded systems. A dynamic voltage scaling *(DVS) technique reduces the processor's dynamic power consumption quadratically at the expense of linearly decreasing the performance. Reducing energy using DVS in the context of real-time systems should consider this tradeoff. In this paper, we introduce a novel collaborative approach between the compiler and the operating system (OS) that uses fine-grained information about the execution times of a real-time application to reduce energy consumption. We use the compiler to annotate an application's source code with path-dependent information called* power management hints *(PMHs). This information captures the temporal behavior of the application, which varies by executing different paths. During program execution, the OS periodically changes the processor's frequency and voltage based on the temporal information provided by the PMHs. These speed adaptation points are called* power management points *(PMPs). We evaluate our scheme using two embedded applications: a video decoder and an automatic target recognition application. Our scheme shows an energy reduction of up to 79% over no power management and up to 50% over a static power management scheme.*

1 Introduction

The increase in user demands for mobile and embedded systems technology requires an equivalent increase in processor performance, which causes an increase of the power consumption of these devices. Managing energy consumption, especially in devices that are battery operated, increases the number of applications that can run on the device and extends the device's battery lifetime. With efficient power management, an increase in mission duration and a decrease in both device weight and total cost of manufacturing and operation can be achieved.

Many of the emerging real-time applications designed for battery operated devices such as wireless communication, audio and video processing tend to consume a considerable amount of energy. A current research direction that has proven its efficiency in real-time systems achieves a tradeoff between energy and performance using *dynamic voltage scaling* (DVS). In a DVS equipped processor, such as Transmeta's Crusoe and Intel's XScale processors, the CPU voltage and frequency can be changed on-the-fly to accommodate processor workload. DVS achieves low energy consumption by exploiting the quadratic relation between energy and performance. That is, decreasing the performance (CPU speed) linearly reduces the energy consumption quadratically. The goal is to satisfy the time constraints of a real-time application while consuming the least amount of energy.

Dynamically changing the speed in real-time applications is categorized as *inter-task* or *intra-task* voltage scaling [6]. Inter-task schedules speed changes at each task boundary to meet a deadline associated with each task, while intra-task schedules speed changes within a single task. In this paper, we introduce a novel collaborative compiler and operating system (OS) intra-task approach that uses fine-grained information about an application's execution time to reduce energy consumption. We use the compiler to annotate an application's source code with temporal information called *power management hints* (PMH). These hints capture the dynamic behavior of the application, which may vary by executing different paths with different execution times. During program execution, the operating system (OS) periodically invokes an *interrupt service rou-*

[*]This work has been supported by the Defense Advanced Research Projects Agency (contract F33615-00C-1736) and National Science Foundation (contract ANI-0125704) through the Power Aware Real Time Systems (PARTS) group.

tine (ISR) that adapts the processor's frequency and voltage based on the temporal information provided by the PMHs. These speed adaptation points are called *power management points* (PMPs).

We extend the OS and the compiler to collaborate in implementing a DVS technique that uses the strength of each of them. The compiler conveys path-specific run-time information about a program's progress to the operating system. Through the periodicity of the ISR executing PMPs, the OS controls the number of speed changes during the application execution, aiming to control the overhead of DVS. We evaluate our scheme on popular real-time applications, including a MPEG decoder and automatic target recognition (ATR). We show a reduction in energy consumption of up to 79% over no power management and up to 50% over a static power management scheme.

In the next section, we introduce the DVS and application models used in our work. In Section 3 we describe the collaboration between the OS and the compiler, detailing the different phases of the algorithm for managing power consumption. Section 4 shows the required support from the OS and the compiler for power management. Evaluation of the scheme is discussed in Section 5. Section 6 briefly describes related work and Section 7 concludes the paper.

2 Real-Time Application and DVS Models

Application Model: A real-time application has a deadline, d, by which it should finish execution. However, if there are several applications in the system, the allotted time d comes from CPU reservations in real-time operating systems (OSs) or from engineering real-time embedded systems. Each application is characterized by its *worst-case execution time* (WCET). If $d > WCET$ (i.e., the allotted time exceeds the WCET), the time difference is known as *static slack*. During actual execution, an application runs for *actual execution time* (ACET), which is smaller than or equal to WCET. The difference between WCET and ACET is called *dynamic slack*, which usually comes from data dependencies that cause the program instances to execute different paths. Since in a DVS system the time taken to execute an application varies with the speed at which the processor is operating, the execution of an application should be expressed in cycles rather than time units. Thus, it is safe to represent an application duration by its *worst case cycles* (WCC), where WCET is the time spent executing WCC at maximum frequency. Similarly, ACC is the actual number of cycles spent executing an application.

We consider the general form of real-time applications that may consist of sequential code, branches, loops and procedures. A program is represented by a *path graph* (PG), which is an extended *control flow graph* (CFG) (CFGs are used as an intermediate representation in compilers). PG merges adjacent nodes in a CFG, known as *basic blocks*, to simplify the task of collecting the temporal information as well as the PMH placement. More details about the PG construction are found in [2]. In our scheme, we divide an application's PG into regions. At the start of each region i, the worst case remaining cycles, wcr_i, to the end of the application is known through profiling or software timing analysis.

Energy Model: In CMOS circuits, power is directly proportional to the square of the input voltage: $P \ \alpha \ CV_{dd}^2 f$, where C is the switched capacitance, V_{dd} is the supply voltage, and f is the operating frequency. Hence, reducing the voltage reduces the power consumption quadratically. However, because the processor clock frequency is dependent on the input voltage, reducing V_{dd} causes a program to run slower. The energy consumed by an application is $E = Pt$, where t is the time taken to run an application with an average power P. Thus, running a program with reduced V_{dd} and frequency leads to energy savings.

Processor Model: Several commercially available processors have recently included DVS capabilities to make trade-offs between performance and energy consumption. The Transmeta Crusoe TM5400 [19] has 16 voltage levels that range from 1.1V to 1.65V operating at 200 MHz to 700 MHz, respectively. Each speed step is around 33 MHz. The Intel X-scale [17] has fewer levels and larger steps. The voltages range from 0.75V to 1.8V (150 MHz to 1 GHz with first a 250 MHz step then 200 MHz steps).

Overhead Model: When computing and changing CPU frequency and voltage, several sources of overhead may be encountered. In this paper, we refer to changing the voltage/frequency and speed change interchangeably, but we differentiate the principal sources of overhead between (1) computing a new speed using a dynamic speed setting scheme, and (2) setting the speed through a voltage transition in the processor's DC-DC regulator (resulting in a processor frequency change) and the clock generator. The speed setting overhead has been measured and ranges from 150μ sec to 25μ sec and consumes up to 4μ J [13, 9]. Our measurements show that it is reasonable to assume that the time T_{set} and the energy consumed E_{set} for voltage setting are constants for all the power level transitions. The overhead of computing the speed both in time (T_{comp}) and in energy (E_{comp}) depends on the CPU operating frequency during such computation.

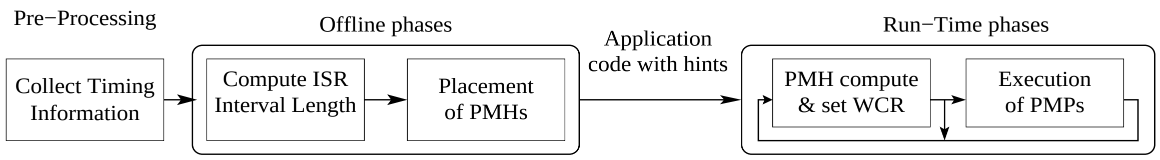


Figure 1. Phases of the collaborative power management scheme.

3 OS-Compiler Collaboration Scheme

Considering the general form of any real-time application code, automatically deciding on the proper location to invoke PMPs using an intra-task DVS scheduling is not trivial. One problem is how frequently one should change the speed. Ideally, the more voltage scaling invocations, the more the application can exploit dynamic slack, and further reduce energy. However, in practice we do not have this freedom, as there is energy and time overhead associated with each speed adjustment as discussed in Section 2. In [1] we studied a special case in which we consider only sequential code. Our goal then was to determine how many PMPs should be inserted in the code to minimize energy consumption while taking into account the overhead of a PMP. We presented a theoretical solution that determines how far apart (in cycles) any two PMPs should be placed. Beyond that preliminary work (i.e., in real applications), the problem is harder due to the presence of branches, loops and procedure calls that eliminate the determinism of the executed path compared to sequential code. In this work, we investigate how to deal with these program structures.

To control the number of speed changes during the execution of an application, we use a collaborative compiler-OS technique. Figure 1 shows that our scheme is divided into offline and run-time phases. Initially, some timing information is collected about a program's execution behavior. During the offline phases, this information is used in computing the ISR interval length in a way that minimizes the number of PMPs executed in an application yet achieves a low energy consumption. Next, the compiler inserts instrumentation code (PMH) that is capable of computing the *worst-case remaining* cycles (wcr) of the application starting from each hint location. The value of wcr at these locations may vary dynamically based on the executed path. For example, a wcr at a hint inside a procedure body is dependent on the path this procedure instance is called from. During run-time, a PMH computes and passes dynamic timing information (i.e., wcr) to the OS in a predetermined memory location named WCR_r. Periodically, a timer interrupt invokes the operating system to execute an *interrupt service routine* (ISR) that does the PMP job; that is, the ISR adjusts the processor speed based on the latest WCR_r value.

Figure 2 shows how PMHs and PMPs work together. An ISR is periodically invoked (the large bars) to adjust processor speed based on WCR_r (WCR_r carries the most recent value of the estimated worst case remaining cycles). In the compiler, the PMH placement algorithm (1) divides an application code into segments such that the worst case execution cycles *of each segment* (wcc) does not exceed the ISR interval length (in cycles) and (2) inserts a PMH (the short bars) after each segment. Each PMH_i computes the worst case remaining cycles (wcr_i) and stores it in WCR_r. A PMH_i is guaranteed to execute at some point before a PMP to update the WCR_r with the value of wcr_i based on a path of execution. Based on the actual execution cycles of each segment (which is less than or equal wcc), more than a PMH can be inserted in a single interval, which improves the estimation of WCR_r (by increasing the probability of executing a hint within a very small time before an ISR occurrence).

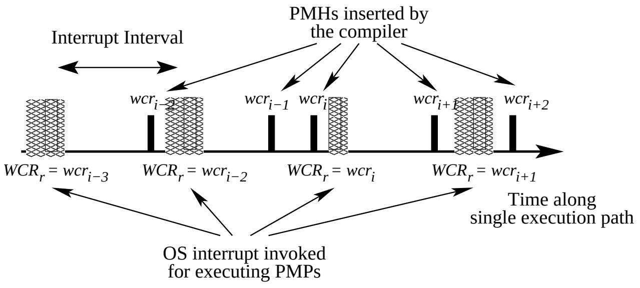


Figure 2. Invocations of PMHs and PMPs for a specific path.

Our scheme captures the strengths of both the OS and the compiler in directing the DVS in contrast to an OS-only or compiler-only DVS direction. The compiler estimates the wcr_i of the application, while the OS controls the number of executed PMPs. On the other hand, in a solely OS directed scheme, the OS is unaware of the application execution progress and its worst case remaining cycles, while a solely compiler-directed scheme can not control how often PMPs will be called (due to the non-determinism in which path is executed) and thus there is no upper bound on the overhead.

The advantage of using a collaborative scheme that includes both PMHs and PMPs over a scheme that uses only PMPs is that timing information can be collected without actually performing a speed change and incurring its high overhead. Since the overhead of executing a PMH (setting a register or writing to memory) is much less than executing a PMP (computing and setting a new frequency and adjusting the supply voltage), we have more freedom to add PMHs in the code as often as necessary to accurately update the WCR_r. For example, consider the case of a loop body that contains mainly two exclusive branches (e.g., `if-then-else` statement), such that the wcc of the first branch (`then` part) is much larger then the second one (`else` part). Consider a scheme that utilizes PMPs alone (the location of the PMPs in the code are statically determined) and places a PMP inside the loop body before the branch. This scheme would hurt the energy consumption if the smaller branch (`else` part) is executed more often than the large branch (`then` part). This is because, with each loop iteration the scheme pays the overhead of the speed change independent of the branch/path visited. However in our scheme, by replacing the PMP in the loop with a PMH, we reduce the overhead induced in each iteration and the actual speed change (PMP) is scheduled after a "reasonable" amount of work done, such that the potential energy gain from DVS justifies the energy overhead exerted by the PMP. The same scenario applies for all program constructs that have variable dynamic execution times.

The strength of our collaborative scheme lies in three properties. First, a separate PMH placement algorithm can be devised to supply the OS with the necessary timing information about an application at a rate proportional to the ISR (PMP) invocation. Second, the actual speed change (which has expensive overhead) is done seldom enough by the OS at pre-computed time intervals to keep the overhead low. This interval is tailored to an application's execution behavior in a way that guarantees minimum energy consumption while meeting deadlines. Finally, by giving the OS control to change speed, our scheme controls the number of executed PMPs based on the length of execution rather than based on a specific path of execution. In the next section we describe the dynamic speed setting schemes used to test our power management algorithm, followed by a detailed description of the different phases in this scheme.

3.1 Dynamic Speed Setting

In this work, we use two schemes from [7] as examples to demonstrate how to include the speed setting schemes in the OS-ISR. We selected the (1) Proportional and (2) Greedy dynamic schemes. We further incorporated the overhead in the speed computation in these schemes. To guarantee meeting the deadline, the total time overhead of computing and changing the speed is deducted from the remaining time before deadline. This time overhead is composed of changing the speed once in the current ISR interval ($T_{comp}(f_{curr}) + T_{set}$) and once for potentially changing the speed after the next interval to the maximum speed ($T_{comp}(f_{next}) + T_{set}$) if the schedule requires so. The total time overhead is expressed as $T_{total}(f_{curr}, f_{next}) = T_{comp}(f_{curr}) + T_{comp}(f_{next}) + 2T_{set}$.

The Proportional scheme In this scheme, reclaimed slack is uniformly distributed to all remaining intervals proportional to their worst-case execution times. Our main concern here is to compute the exact time left for the application to execute before the deadline. The processor's speed at the start of an interval, f_{next}, is computed as follows: the execution times for the remaining tasks are stretched out based on the remaining time to the deadline ($d - ct$) minus possible time spend switching the speed T_{total} where ct is current time at the beginning of the interval. While taking into consideration the overhead of changing the speed, this scheme computes a new speed as:

$$f_{next} = \frac{WCR_r}{d - ct - T_{total}(f_{curr}, f_{next})} \tag{1}$$

The Greedy scheme In this scheme all the available slack is allocated for the next interval. As a result, the scheme tends to schedule the first few intervals at a low speed, and gradually increase it after consuming slack. The frequency for segment i is computed as:

$$f_{next} = \frac{wcc}{d - ct - \frac{WCR_r - wcc}{f_{max}} - T_{total}(f_{curr}, f_{next})} \tag{2}$$

where f_{max} is the maximum frequency the processor can operate on and wcc equals $WCC/$number of intervals. Similar to what we did for the proportional scheme, subtracting the overhead components from the remaining time after accounting for the execution time of the remaining intervals (excluding the current one) running at f_{max}.

3.2 Offline Phases

The compiler plays a major role in the offline phase. It computes the interrupt interval length and then places the PMHs in the application's source code based on timing information collected in pre-processing.

Setting the interrupt interval: Determining the interval for invoking PMPs is based on the average and worst case execution times of a program and the overhead of PMPs. We use the formulation of speed computation for sequential code to determine the interrupt interval length in cycles. This approach is based on our prior work [1] that devised a theoretical formulation of speed computation for sequential code with perfect execution behavior (i.e., the WCC and ACC are known and fixed). With the overhead of the speed computation and voltage change considered, we find the average number of PMPs that should be executed to consume minimum energy. By dividing the average number of cycles for executing an application by this number, we get the interval size. Note that, this division is just an approximation that yields to a sub-optimal solution.

Placement of PMHs: The goal of the PMH placement algorithm is to ensure that there exits at least one PMH executed during each ISR interval. We assume that we have an estimate for the worst-case timing for each branch in the application's code (from pre-processing), either through profiling or static/dynamic analysis [15, 14]. This timing information is used in the PMH placement algorithms and in conveying the remaining time to the operating system. The PMH placement algorithm traverses the PG and, for each path, it uses a cycle counter *ac* that is incremented by the worst case cycles of each traversed region of the PG. A hint is inserted before the counter exceeds the ISR interval length. The counter is reset after any hint insertion. In case of branches, ac is propagated to all branches. After any join, the maximum counter among all joined branches is taken as the new value for ac. In more complex program structures as in loops and procedure calls, PMHs are inserted in the loop and procedure bodies if the body size exceeds the interval length. More details on the PMH placement algorithm are explained in [2, 3].

3.3 Run-Time Phases

During run-time, visited PMHs compute and update WCR_r. At each ISR invocation, the OS evaluates a new speed based on WCR_r. After each ISR, control then switches back to the application during which more PMHs are executed. Execution continues in this way until the program terminates.

PMH computation of wcr: The compiler inserts offline an instrumentation in the application's code to estimate the dynamic wcr_i at each PMH_i. These values are computed based on the execution path followed during runtime especially in case of procedure calls and loops. In loops, wcr_i inside a loop body is dependent on the loop iteration counter. Estimation of wcr_i inside loops are based on work presented in [14]. On the other hand, the worst case remaining cycles at each procedure instance is based on the execution path it is called from. To estimate the dynamic wcr_i at each procedure call (for each procedure containing hints), an entry is inserted at run-time in a dynamic table that stores the worst case remaining cycles for this procedure instance. Estimation of wcr_i at each PMH_i inside a procedure body is based on the stored value in table that corresponds to this specific procedure instance.

Table 1. ISR pseudocode

```
1   $ct$ = read current time
2   $f_{curr}$ = read current frequency
3   $f_{new}$ = compute speed (algo., $ct, d, wc, WCR_r$ )
4   if ($f_{new} \leq f_{min}$ ) $f_{new} = f_{min}$
5   else if ($f_{new} \geq f_{max}$ ) $f_{new} = f_{max}$
6   else foreach power level $Pl_i$    $(i = 1..k)$
7           if ($f_{new} > Pl_i.freq$ )
8               $f_{new} = Pl_{i-1}.freq$
9               break
10  if ($f_{new} \neq f_{curr}$)
11          set_speed ($f_{new}$ )
12          $interval\_time = interval\_cycles/f_{new}$
13  rti
where:
$Pl_i.freq$ is operating frequency for power level $Pl_i$,
$k$ is number of power levels, such that: $Pl_k.freq = f_{min}$,
$Pl_0.freq = f_{max}$, and $Pl_i.freq > Pl_{i+1}.freq$
```

ISR execution of PMP: In our scheme, the OS virtually divides the program into equal intervals by invoking the ISR (for executing a PMP) periodically. With the knowledge of the estimated WCR_r at each interval, dynamic speed setting schemes can compute the desired operating frequency for the next interval (using Equations 1 or 2). Then it issues a speed change request if needed. The pseudocode of the ISR is listed in Table 1. The ISR reads the most recent time, frequency and WCR_r, and computes a new speed. It then selects an appropriate frequency and issues a speed change if the selected frequency is different than the current one.

4 Support for OS-Compiler Interaction

In this section we describe the support needed from the compiler and the operating system.

OS support: The operating system is equipped with an ISR for setting the speed, and two system calls to communicate with the application to transfer the timing information.

- ***SetInterval* system call:** Setting the ISR interval (in cycles) requires a *system call* that is called only once at the start of the program. Although the number of cycles in the interval is constant throughout the application execution, the time interval varies due to frequency changes. The ISR computes the time interval for triggering the next ISR.

- ***SetWcrLocation* system call:** Setting an address of a memory location that stores the value of WCR_r is done through a *system call* that is called only once at the start of the program. At run-time hints store the worst case remaining information in this address. Whenever a PMP is invoked, the ISR loads this information and uses it to compute the speed for the next interval.

- **Interrupt service routine:** The ISR defines a selected dynamic speed setting scheme for computing a new operating frequency, then it selects an appropriate power level to operate on. When working with discrete power levels, for a real-time application to meet its deadline, we choose to operate on the smallest power level larger than or equal to the desired frequency obtained from the speed setting scheme.

- **Preemption support:** In case more than one application running in a system allowing process preemption, the OS should keep track of how long each application was running with respect to the current interval time. At each context switch, the elapsed interval time is stored as part of the status of the departing process (application) and replaced by the corresponding value of the newly dispatched process. Similarly, the interval time and the operating frequency become part of the application context.

Compiler support: To support our proposed scheme, the compiler inserts PMHs in the application code. For computing the remaining time dynamically, the compiler instruments the PMHs in a way to collect the application's dynamic behavior information.

To keep track of the remaining time at each hint located inside a procedure, the remaining time at the start of each procedure instance[1] (p_wcr_j) should be stored and retrieved at run time. The compiler allocates and manages memory space for a table that stores p_wcr_j updated by PMHs. Each entry in this table is deleted at the termination of a procedure instance that corresponds to this entry. From our experiments, this table size does not exceed five entries. That is, there are no more than five active procedures that have hints at the same time. Hints inside these active procedures retrieve p_wcr_j from the table and compute the remaining time based on this value.

5 Evaluation

To evaluate our scheme, we extended the *sim-outorder* simulator from SimpleScalar [16] by adding a module that computes the energy of the running application based on the number of cycles C spent at each voltage level V ($E = CV^2$). C includes cycles for executing the application, the overhead cycles of PMHs and the overhead cycles of computing the speed in PMPs. We also consider the energy overhead consumed in setting the speed. We experimented with the models of Transmeta Crusoe and the Intel XScale processors (see Section 2) and show the effect of our algorithm for these processors. We briefly describe a few differences in the results between the two processors.

We also implemented in SimpleScalar an ISR that computes at each *interval* a new frequency, and then sets the corresponding voltage. To compute the speed, we use the Greedy and Proportional dynamic schemes. We compare these two schemes with no power management (NPM) (execute all intervals at f_{max}) and a static power management (slow CPU down based on only static slack). Currently, instrumentation of hints in the application code is inserted manually. Implementation of the PMH placement algorithm in real compiler is under development.

In this paper, we show results for two real-time applications: an MPEG2 decoder [18], and automated target recognition (ATR) [8]. We show the effect of our scheme on energy consumption and performance. We also discuss the effect of the overhead associated with each approach. The experiments do not consider multiple running applications in the system as it is not supported by the simulator used.

5.1 Impact on energy and performance

A general observation in all the presented results is that, as the time allotted to an application to execute is increased, less energy is consumed – relative to NPM – due to the introduction of more (static) slack that can be used to reduce the processor's speed/voltage. Also, when the deadline is very large, all the schemes tend to run on the minimum voltage level due to the abundance of slack; that is, they usually converge to single low energy consumption. In the Crusoe case, the normalized energy consumption converges to a higher level than in the case of the XScale. This is because the ratio between the minimum and maximum voltages in the Crusoe is larger than in XScale, and hence, there was less opportunity for percentage savings in Crusoe.

Automatic target recognition (ATR): The ATR application[2] does pattern matching of targets in image frames. We experimented with 190 frames. The number of target detections in each frame varies from zero to eight detections. Frame processing time is proportional to the number of detections within a frame. In Figure 3, greedy and proportional consume less energy than the static scheme because of dynamic slack reclamation. Note that all frames met their deadlines. Since Crusoe has more power levels than XScale, the static scheme for Crusoe had more speed transitions as the deadline changes. Each transition corresponds to operating at a different speed level.

[1]The p_wcr_j values are stored for only procedures that include hints inside their bodies.

[2]The original code and data for this application were provided by industrial research partners.

At a tight deadline of 10 msec, Crusoe works on a full load in case of the static scheme (not true with proportional and greedy because of dynamic slack reclamation). On the other hand, the same workload constitutes only about 80% of the XScale processor load due to a higher f_{max}.

In the XScale case, the static scheme outperforms the greedy scheme for 18-20 msec deadlines. By knowing that the smallest two speed levels are 150 and 400 MHz, and most of the speeds computed in the intervals in the greedy scheme require a speed slightly higher than 150MHz but can only operate on 400 MHz (which is the same operating frequency for the static in this range). Hence both schemes run with almost the same power but because of the overhead incorporated with the dynamic schemes, the greedy consumes more energy than the static scheme.

MPEG video decoder: We collected timing information about the MPEG2 decoder using a training data set of six files and tested it on a different set of 20 data files. An important aspect of MPEG decoding is the existence of different types of frames (I, P and B) that vary in their execution times. Moreover, there is also a variation in execution times among frames of the same type. We inserted the PMHs in the decoder's code based on profile information about frames (i.e., independent of their types) for all training movies. We ran experiments for four different frame rates: 15, 30, 45 and 60 frames/sec that correspond to deadlines of 66, 33, 22, and 16 msec. Deadlines are set per frame. For all the movies, the deadline was met for each of the decoded frames. Figure 4 shows the results of the energy consumption for three of the test movies for XScale and Crusoe. Similar to the ATR results, the dynamic schemes consume less energy than the static scheme. Also our scheme showed better results in most cases when compared to a semantic static approach, which sets a different speed for each frame based on its type [2].

Similar to the ATR on XScale at low power levels, for deadline 33 msec, all the schemes approach the same energy consumption (most of the intervals operate at 400 MHz rather than at a lower speed). As the deadline increases (i.e., at 66 msec), the static scheme does not converge to the lowest power level because it can not operate lower than 400MHz due to the lack of sufficient static slack. The behavior is similar for all movies in this set.

5.2 Overhead of the collaborative scheme

Our scheme introduces overhead due to PMPs and PMHs. Here we further investigate the impact of each on overall performance (numbers of cycles and instructions).

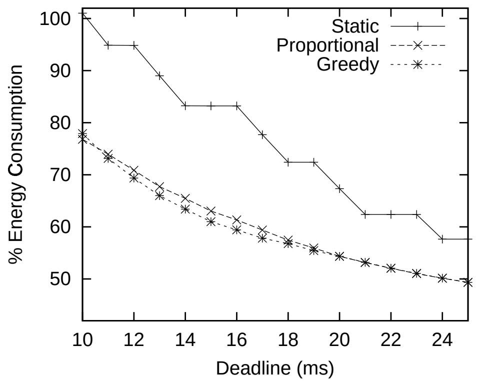
Number of instructions: Increasing the number of inserted instructions could have a negative effect on (1) the cache miss rate and (2) the total number of executed instructions. Both of them could degrade performance. In our scheme, the effect of increasing the code size on the instruction cache miss rate is minimal since the inserted hint code is simple and concise. A PMH takes 12 instructions to execute, while the ISR takes 130 instructions to compute and select a speed (excluding the actual voltage change). On the other hand, the number of executed instructions is kept minimal by (1) invoking the ISR at relatively large intervals (ranges from 250 to 580 Kcycles in the presented applications) and (2) avoiding the excessive insertion of PMHs. For example, in the ATR application, for each interrupt interval, only an extra 533 instructions are executed on average (including all executed PMHs and a PMP in this interval). The total increase in the number of executed instructions due to the overhead is about 0.05% for ATR and 0.25% for MPEG.

Number of cycles: The extra inserted code (PMHs and PMPs) increases the number of cycles taken to execute an application. The average execution of each PMH takes 38 cycles and PMP takes 165 cycles to execute. The total increase in the number of cycles ranges between 0.19% to 0.4% for ATR and 0.4% to 1.7% for MPEG. The total overhead cycles decrease by decreasing the processor frequency. This is due to relatively decreasing the memory latency compared to the CPU frequency.

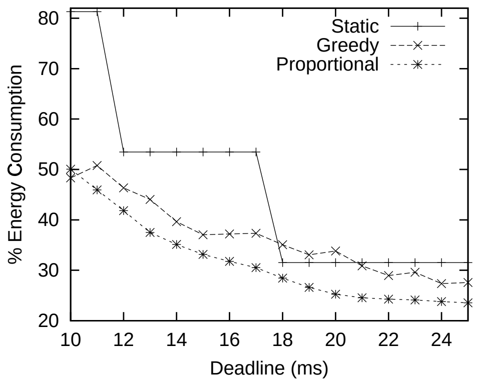
6 Related Work

There have been a number of research projects that use DVS schemes to reduce the processor's energy consumption. Some researchers proposed implementations for intra-task voltage scheduling that is managed either on the compiler/application or the OS level. On the OS level, [5] determines a voltage scheduling that changes the speed within the same task/application based on task's statistical behavior. [10] modified the EDF and RMS in RT-Linux to incorporate voltage scheduling.

References [12] and [4] apply a compiler controlled DVS in real-time applications. In [4] the compiler inserts checkpoints at the start of each branch, loop, function call, and normal segment. Information about the checkpoints along with profile information are used to estimate the remaining cycle count and hence compute a new frequency. Runtime overhead of updating data structures and setting the new voltages are relatively high especially on the constructed nodes granularity (almost every basic block). In [12], each basic block in the constructed CFG is augmented with its worst-case execution along with the remaining WCC till the end of the program. “Branches in CFG that drop the remaining worst case cycles faster than the execution rates are selected as locations to insert the speed change procedure”. Authors did not mention how to deal with procedure

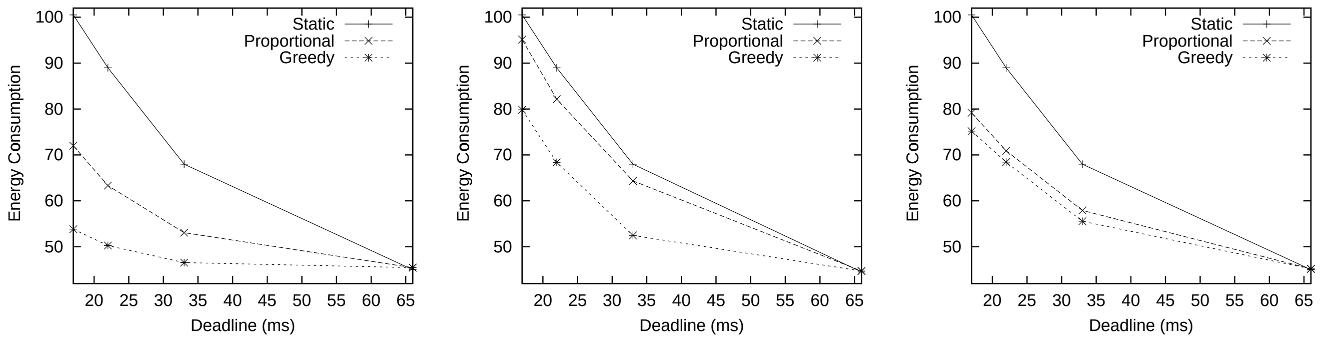


(a)



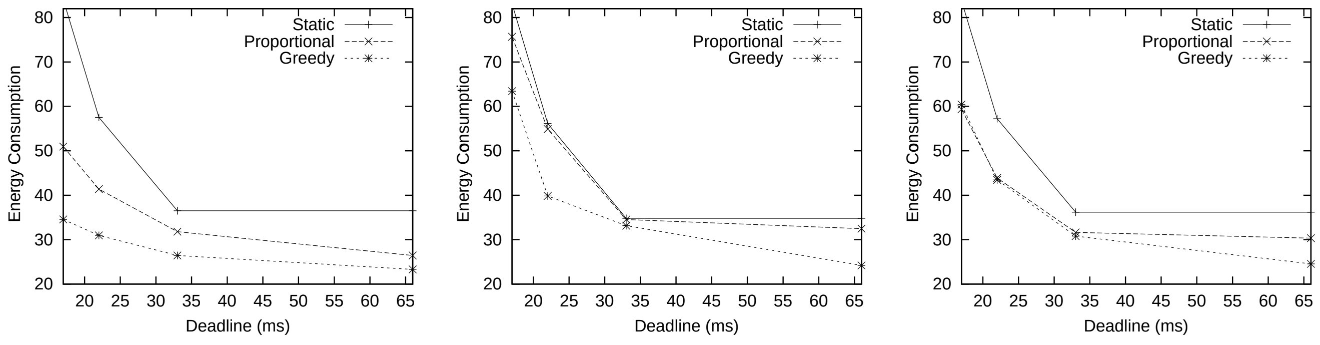
(b)

Figure 3. Average energy consumption normalized to no power management for ATR on the (a) Transmeta Crusoe and (b) Intel Xscale models.



BigE_Anim.mpg atom.mpg robot.mpg

(a)



BigE_Anim.mpg atom.mpg robot.mpg

(b)

Figure 4. Average energy consumption normalized to the no power management scheme for the MPEG2 decoder employing (a) Transmeta Crusoe and (b) Intel XScale models.

calls especially when they are called from different paths; the remaining WCC of these procedures would be dependent on the path they are called from.

In [6], a performance analysis for a set of DVS schemes was presented aiming to find the strengths and drawbacks of the schemes being tested on a unified DVS simulation environment. A couple of intra-DVS offline scaling decisions [12, 5] were studied. It was shown that the performance of these two schemes were quite different depending on the available slack times.

There are several static and dynamic compiler techniques for estimating the best and worst case execution times of programs. A review of different tools for estimating the WCET is presented in [11]. Some of these tools use static analysis to produce a discrete WCET bound. On the other hand, parametric analysis for computing WCET as in [14] evaluates expressions in terms of parameter variables carrying information about some program segments; e.g., WCET of loops is presented by symbolic formulas evaluated at runtime when the number of loop iterations is determined.

7 Conclusion

This paper presented a novel intra-task DVS algorithm achieved through the collaboration of the operating system and the compiler. This collaboration succeeded in reducing the energy consumption by transferring the dynamic execution behavior of the application – through compiler inserted instrumentation – to the operating system who is responsible for scheduling speed changes. Considering the relatively expensive overhead of a voltage/speed change, our scheme reduces the energy consumption by controlling the number of speed changes based on application behavior. Results showed that while meeting all the application's deadlines, our scheme reduces energy up to 79% and 50% over no power management and static power management, respectively.

Acknowledgment

We thank the referees for their criticisms and suggestions that helped us in rewriting the paper in a better form.

References

[1] AbouGhazaleh N, Mossé D, Childers B & Melhem R. Toward The Placement of Power Management Points in Real Time Applications. *Workshop on Compilers and Operating Systems for Low Power (COLP).* 2001.

[2] AbouGhazaleh N, Childers B, Mossé D, Melhem R & Craven M. Collaborative Compiler-OS Power Management for time-sensitive applications. Technical Report TR-02-103, 2002. http://www.cs.pitt.edu/PARTS

[3] AbouGhazaleh N, Childers B, Mossé D, Melhem R & Craven M. Energy Management for Real-Time Embedded Applications with Compiler Support *Languages, Compilers, and Tools for Embedded Systems, (LCTES).* 2003.

[4] Azevedo A, Issenin I, Cornea R, Gupta R, Dutt N, Veidenbaum A & Nicolau A. Profile-based dynamic voltage scheduling using program checkpoints. *Design automation and test in Europe*. 2002.

[5] Flavius Gruian. On energy reduction in Hard Real-Time Systems Containing Tasks with stochastic Execution times. *IEEE Workshop on Power Management for Real-Time and Embedded Systems*. 2001.

[6] Kim W, Shin D, Yun H, Kim J, Min S. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. *Real-Time and Embedded Technology and Applications Symposium*. 2002.

[7] Mossé D, Aydin H, Childers B & Melhem R. Compiler-assisted dynamic power-aware scheduling for real-time applications. *Workshop on Compilers and Operating Systems for Low Power, COLP00* . 2000.

[8] Mahalanobis B, Vijaya K & Sims R. Distance-Classifier correlation filters for multiclass target recognition. *Applied Optics, Vol.35, No.7* 1996.

[9] Pering T, Burd T & Brodersen R. Voltage Scheduling in the lpARM Microprocessor System. *International Symposium on Low Power Electronics and Design* 2000.

[10] Pillai P & Shin K. Real-time Dynamic voltage scaling for low-power embedded operating systems. *18th symposium on operating systems principles* . 2001.

[11] Puschner P & Burns A. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. 2000.

[12] Shin D, Kim J & Lee S. Intra-task Voltage scheduling for low-energy Hard Real-Time Application. *IEEE Design and Test of Computers*. 2001.

[13] Min R, Furrer T & Chandrakasan A. Dynamic Voltage Scaling Techniques for Distributed Micro-sensor Networks. *IEEE VLSI Workshop*. 2000.

[14] Vivancos E, Healy C, Meuller F & Whalley D. Parametric Timing Analysis. *Workshop on Language, Compilers, and Tools for Embedded Systems*. 2001.

[15] Vrchoticky A. Compilation Support for Fine-Grained Execution time Analysis *Workshop on Language, Compiler, and Tool Support for Real-Time Systems* 1994.

[16] http://www.simplescalar.com

[17] http://developer.intel.com/design/intelxscale

[18] MPEG2 decoder, http://www.mpeg.org

[19] Transmeta Corporation, Crusoe Processor Specification, http://www.transmeta.com