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

NEVINE ABOUGHAZALEH, DANIEL MOSSÉ, BRUCE R. CHILDERS, and RAMI MELHEM
University of Pittsburgh

Managing energy consumption has become vitally important to battery-operated portable and embedded systems. *Dynamic voltage scaling* (DVS) reduces the processor's dynamic power consumption quadratically at the expense of linearly decreasing the performance. When reducing energy with DVS for real-time systems, one must consider the performance penalty to ensure that deadlines can be met. In this paper, we introduce a novel collaborative approach between the compiler and the operating system (OS) 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 fine-grained 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 three embedded applications: a video decoder, automatic target recognition, and a sub-band tuner. Our scheme shows an energy reduction of up to 57% over no power-management and up to 32% over a static power-management scheme. We compare our scheme to other schemes that solely utilize PMPs for power-management and show experimentally that our scheme achieves more energy savings. We also analyze the advantages and disadvantages of our approach relative to another compiler-directed scheme.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms: Algorithms, Management, Experimentation

Additional Key Words and Phrases: Real-time, dynamic voltage scaling, power-management, collaborative OS and compiler, voltage scaling points placement

Authors' address: Nevine AbouGhazaleh, Daniel Mossé, Bruce R. Childers, and Rami Melhem, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA.
Support for the Power Aware Real Time Systems (PARTS) group has been provided by DARPA (contract # F33615-00C-1736), NSF (grants ANI-0087609, ACI-0121658, ANI-0125704, CNS-0305198, and CNS-0203945), and IBM Faculty Partnership Award.
Parts of this work appeared in COLP'01, RTAS'03, and LCTES'03.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1539-9087/06/0200-0082 $5.00

1. INTRODUCTION

The expanding demand for mobile and embedded systems technology requires an equivalent improvement in processor performance, which causes an elevates in 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 extension 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 as well as audio and video processing, tend to consume a considerable amount of energy. A current research direction that has proved its efficacy in real-time systems achieves a trade-off between energy and performance using *dynamic voltage scaling* (DVS). In a DVS-equipped processor, such as Transmeta's Crusoe [Transmeta 2002] and Intel's XScale [XScale 2002] processors, CPU voltage and frequency can be changed on-the-fly to accommodate a processor's workload. DVS achieves low-energy consumption by exploiting the quadratic relationship between power and performance. That is, linearly decreasing performance (CPU voltage and speed) reduces the CPU power quadratically. The goal of DVS schemes 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-* or *intratask* voltage scaling [Kim et al. 2002]. Intertask DVS schedules speed changes at each task boundary, while intratask DVS schedules speed changes within a single task. In this paper, we introduce a novel collaborative compiler and operating system (OS) intratask approach that uses fine-grained information about an application's execution time to reduce energy consumption. We use the compiler to extract temporal information and annotate an application's source code with *power-management hints* (PMH). PMHs capture the dynamic behavior of an application, including variances in execution path and time. During program execution, the operating system (OS) periodically invokes an *interrupt service routine* (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 instruments the program to convey path-specific run-time information about the program's progress to the operating system. Through the periodicity of the ISR that executes PMPs, the OS controls the number of speed changes during an application's execution, which, in turn, reduces or controls the overhead associated with the DVS technique.

We evaluate our technique on three time-sensitive applications: an MPEG-2 decoder [MSSG 2001], automatic target recognition (ATR) [Mahalanobis et al. 1996], and sub-band filter [Rickard et al. 2003]. Using models of two commercially available processors with dynamic voltage scaling (Transmeta's Crusoe and Intel's XScale), we show that our technique can achieve significant energy

reduction of up to 57% over no power-management and up to 32% over static power-management. We further demonstrate that our scheme achieves less energy consumption than (1) a scheme that uses a simple compiler placement of PMPs at each procedure call and (2) a scheme similar to ours that uses only PMPs rather than the cheaper PMHs combined with PMPs. We also compare the advantages and disadvantages of our collaborative scheme against a purely compiler-directed scheme [Shin et al. 2001].

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 necessary instrumentation for the implementation of the collaborative scheme in both the compiler and operating system. Evaluation of the scheme is discussed in Section 5. Section 6 briefly describes related work followed by Section 7 that concludes the paper.

2. REAL-TIME APPLICATION AND DVS MODELS

2.1 Application Model

A real-time application has a deadline, d, by which it should finish execution. If there is a single application running in the (embedded) system, then d represents the time allotted for execution of the application. When there are several applications in the system, the allotted time d comes from CPU reservations in real-time operating systems 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 run-time, an application runs for an *actual execution time* that is smaller than or equal to WCET. The difference between the worst-case and actual execution times is called *dynamic slack*, which usually comes from data dependencies that cause program instances to execute different paths.

Since in a DVS system the time taken to execute a program instance varies with the speed at which the processor is operating, the execution of an application is expressed in cycles rather than time units. Thus, we can represent an application's duration by its *worst-case cycles* (WCC), where WCET is the time spent executing WCC at maximum frequency.

2.2 Energy Model

In this paper, we focus on reducing the processor's energy consumption. In CMOS circuits, power dissipation is composed of static and dynamic portions. Dynamic power is directly proportional to the square of the input voltage: $P \propto CV_{dd}^2 f$, 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 also 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. Static power dissipation is because of leakage in transistors. In this work, static leakage is assumed to be constant throughout an application's execution.

2.3 Processor Model

Several commercially available processors have recently included DVS capabilities to make trade-offs between performance and energy consumption. We consider realistic processors with discrete voltages and frequencies. Two examples are the Transmeta Crusoe TM5400 [Transmeta 2002] and the Intel X-scale [XScale 2002]. See Section 5 for details of their configuration (voltage and frequency, etc). We refer to changing the voltage/frequency and speed changes interchangeably.

2.4 Overhead Model

When computing and changing CPU frequency and voltage, two main sources of overhead may be encountered, depending on the CPU architecture: (1) computing a new speed 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. This speed-changing overhead has been measured and ranges from 25 to 150 μsec and consumes energy in the range of micro joules (for example 4μJ in *lparm*) [Min et al. 2000; Pering et al. 2000]. We assume that the time and the energy consumed for voltage setting are constants for all the power level transitions, but the time and energy overhead of computing the speed depends on the CPU operating frequency during such computation.

3. OS COMPILER COLLABORATION

Considering the general form of any real-time application code, automatically deciding on the proper location to invoke PMPs using intratask DVS scheduling is not trivial. One problem is how frequently one should change the speed. Ideally, the more voltage scaling invocations, the more fine-grain control the application has for exploiting dynamic slack and reducing energy. However, in practice, the energy and time overhead associated with each speed adjustment can overshadow the DVS energy savings. In past work [AbouGhazaleh et al. 2001], 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 computing and setting a new speed. We presented a theoretical solution that determines how far apart (in cycles) any two PMPs should be placed; in other words, we computed the PMP invocation frequency. Beyond that preliminary work (i.e., in real applications), the problem is harder because of the presence of branches, loops, and procedure calls that eliminate the determinism of the executed path compared to sequential code. In this work, we show 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

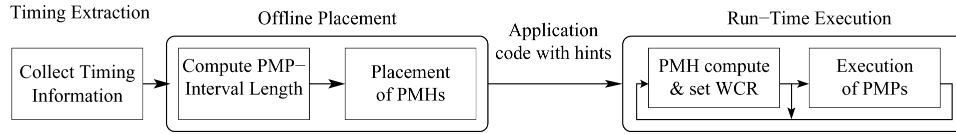


Fig. 1. Phases of the collaborative power-management scheme.

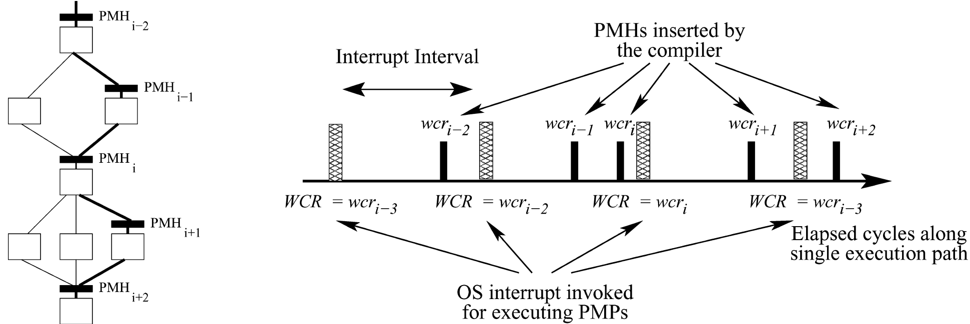


Fig. 2. PMH and PMP invocations (*right*) when executing the bold path in CFG (*left*).

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 to compute how frequently PMPs are invoked. The number of cycles between executions of two PMPs is called *PMP interval*. This interval controls the number of PMPs executed in an application and, therefore, the overhead. Next, the compiler inserts instrumentation code for the PMHs to compute the *worst-case remaining* cycles, wcr_i, starting from the ith PMH location to the end of the application. The value of wcr_i at these locations may vary dynamically based on the executed path for each run. For example, the remaining cycles at a PMH inside a procedure body is dependent on the path from which this procedure is invoked. During run-time, a PMH computes and passes dynamic timing information (i.e., wcr_i) to the OS in a predetermined memory location named *WCR*, which holds the most recent value of the estimated worst-case remaining cycles. Periodically, a timer interrupt invokes the operating system to execute an ISR that does the PMP job; that is, the ISR adjusts the processor speed based on the latest *WCR* value and the remaining time to the deadline.

Figure 2 shows an example CFG and how PMHs and PMPs work together. A PMP is periodically invoked (the large bars) to adjust the processor's speed based on *WCR*. In the compiler, a PMH placement algorithm first divides an application code into *segments*, such that the worst-case execution cycles of each segment, *wcc*, does not exceed the PMP interval length (in cycles) and, second, inserts one PMH (the short bars) after each segment. Each PMH_i computes the worst-case remaining cycles (wcr_i) and stores it in *WCR*. A PMH_i is guaranteed to execute at some point before a PMP to update the *WCR* with the value of wcr_i based on the path of execution. Because the actual execution cycles of each

segment is less than or equal *wcc*, we note that more than one *PMH* can be executed in a single PMP interval, which improves the estimation of *WCR*.

Our scheme uses the strengths of both the OS and the compiler to direct DVS in contrast to an OS-only or compiler-only DVS scheme. Our scheme exploits the compiler's ability to extract run-time information about the application's execution progress indicated by wcr_i. The scheme also takes advantage of the OS's ability to schedule DVS events independent of which path is executed in the application. In a solely OS-directed scheme, the OS is unaware of an application's execution progress and the number of remaining cycles, while a solely compiler-directed scheme can not control how often PMPs will be called (because of the nondeterminism of the path followed during program execution) and, thus, there is no upper bound on the overhead.

Although our description in this paper focuses on intratask DVS, our technique is also useful in sharing slack among multiple tasks. The advantage of our scheme is that it can accurately account for the slack generated by a task instance. There are two cases to consider for distributing the slack. First, if there is *no preemption* in the system, at the end of each task execution, unused slack can be distributed among subsequent tasks according to any of the slack-sharing policies [Mossé et al. 2000]. Second, if the system allows for *preemptive scheduling*, when a preemption occurs, the OS computes how much slack has been generated so far by the application. This is done through a PMP-like interrupt invoked at a context switch. This interrupt computes slack based on the preemption time and the latest WCR value. Then, the task scheduler can decide whether to share the newly generated slack with other tasks or save it to be used when the same task instance resumes execution. In this work, we follow the latter approach. Next, we show the advantage of our collaborative scheme when compared with an offline scheme that inserts PMPs in the code. We also describe a few different intratask PMP placement schemes and counterexamples where these schemes may inefficiently use dynamic and static slack.

3.1 PMP versus PMH Placement Strategy

For a single application under consideration, a PMH inserted in the application's code can be more efficient (with respect to slack utilization) than placement of PMPs directly in an application's code based on information available offline. There are two reasons for the possible inefficiency with inserting PMPs. First, an offline decision may not account for all the run-time slack generated by an application, and second, there is no control on the frequency of executing PMPs because of the uncertainty of which paths will be taken at run-time.

Scheduling the speed offline based on profile information can not use all the generated slack. The actual execution time of an application is likely to differ from its offline estimates. This difference contributes to the dynamic slack generated in the system. Since a compiler-only intratask DVS scheme makes speed change decisions based solely on offline estimates, such a scheme can not exploit all the dynamic slack generated at run-time. In contrast, PMHs can convey run-time knowledge about the application execution. Thus, during run-time, a more informed decision can be made, based on the actual slack available

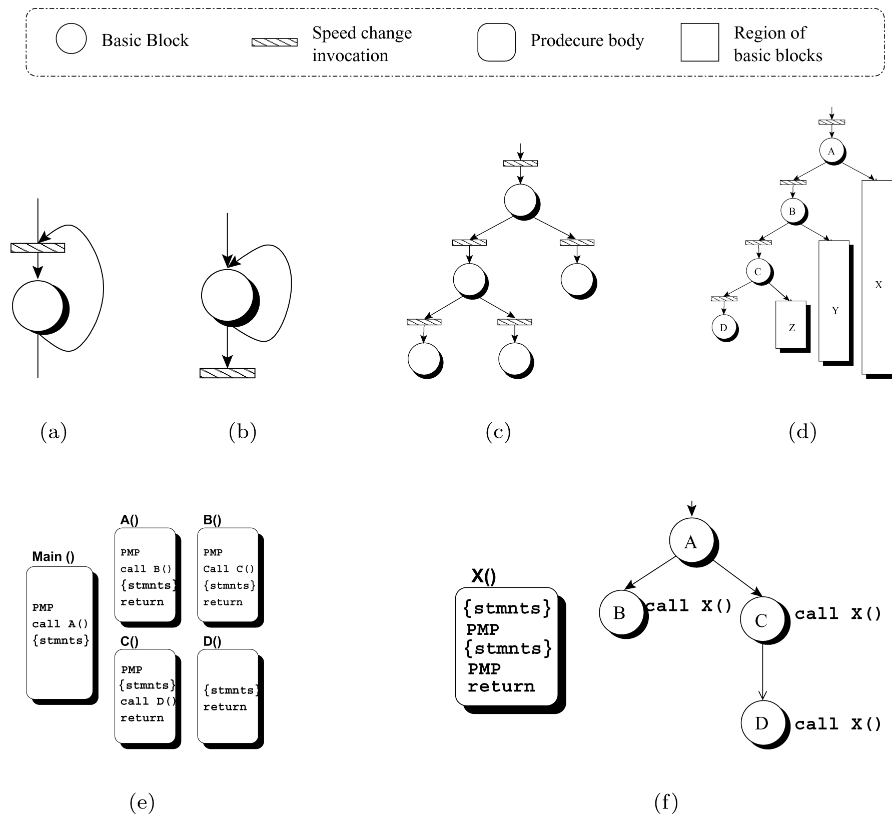


Fig. 3. Some PMP placement strategies and their adverse cases of code structure (see Table I for description).

at the time of the speed transition. Further analysis of the slack utilization efficiency compared to other offline schemes is discussed in Section 5.2.

The uncertainty in knowing the execution path at compile time causes some offline placement strategies to act inefficiently for some combinations of code structures. This inefficiency may contribute to an increase in energy consumption rather than energy savings; this can be because the overhead associated with executing excessive PMPs or because of inefficient slack utilization from too few PMP invocations. Figure 3 and Table I show different PMP placement strategies and some combinations of code structures that show the inefficiency of those strategies to deal with specific cases. Although some slack is exploited, the given examples show that there is no guarantee that the overhead of changing the speed does not offset the total energy savings. We evaluate this issue later in the paper.

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 PMP invocation. Second, the actual speed-change is done seldom enough by the OS at precomputed time intervals (every PMP-interval) to keep the

Table I. Adverse Combinations of Code Structure for Some PMP Placement Strategies

Case	Placement Strategy	Adverse Case	Side Effect
(a)	PMP placed at each loop iteration	Loop body (single iteration) consists of one or few small sized basic blocks, i.e., loop body size is in order of few to hundreds of cycles)	Too frequent speed-change invocations relative to the slack that may be generated in system.
(b)	A PMP placed before the start or after the termination of a loop execution	An application consisting of single main loop	Dynamic slack generated from the loop can not be exploited.
(c)	PMP placed at each branch	Each branch is a basic block before it branches again	Too frequent speed-change invocations.
(d)	Place PMP in branches with non worst-case (cycles) path	Path ABCD is a sequence of basic blocks interleaved with PMPs. Regions X, Y and Z are worst-case paths after blocks A, B, and C, respectively.	Too frequent speed-change invocations.
(e)	PMP placed at each procedure call	Frequent call to procedures of small sizes	Too frequent speed-change invocations.
(f)	PMPs inserted in procedures' bodies	Procedure X contains PMPs and is called from different call-sites in the application	Need a way to estimate the worst-case remaining cycles inside procedure X to compute a new speed

overhead low. The PMP 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 independent of any execution path.

The advantage of using a collaborative scheme that includes both PMHs and PMPs over a scheme that uses only PMPs is that the application's execution progress can be known without actually doing a speed-change and incurring its high overhead. 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 locations of PMPs in the code are statically determined), and places a PMP inside the loop body before the branches. This scheme would hurt energy consumption if the smaller branch is executed more often than the large branch. This is because, with each loop iteration, the scheme pays the overhead of the speed change independent of the branch/path visited. However, using our scheme, by replacing the PMP in the loop with a PMH, we reduce the overhead incurred in each iteration and the actual speed change (PMP) is scheduled after a "reasonable" amount of work, such that the potential energy savings from DVS can justify the energy overhead of the PMP. Similar scenarios occur in all program constructs that have variable execution times.

In the next section, we describe the dynamic speed setting schemes that we use with our power-management algorithm. This description is followed by a detailed discussion of our compiler–OS scheme.

3.2 Dynamic Speed Setting

The OS views the application as a set of sequential segments; an *execution_segment* is a set of instruction between two consecutive PMHs. Each execution_segment has a worst-case execution cycles, *wcc*. Speed changes are scheduled at the start of these segments. In this work, we use two schemes, Proportional and Greedy—first introduced in Mossé et al. [2000] to determine the processor speed. Although we adapt these schemes to account for overheads, our approach can support other schemes as well. Dynamic and static slack generated from past execution_segments are used for reducing the speed for future execution_segments.

To guarantee the deadline, the total time overhead of computing and changing the speed is deducted from the remaining time before the deadline. The overhead consists of the time for voltage setting, T_{set} and the time for computing the speed, T_{comp}. T_{comp} depends on the CPU operating frequency, f_{curr}, during the computation of the speed. This time overhead considered at each PMP is composed of changing the speed once for the current execution_segment ($T_{comp}(f_{curr}) + T_{set}$) and once for potentially changing the speed after the next execution_segment[1] ($T_{comp}(f_{next}) + T_{set}$). The total time overhead is expressed as $T_{total}(f_{curr}, f_{next}) = T_{comp}(f_{curr}) + T_{comp}(f_{next}) + 2T_{set}$. We assume that T_{set} and the corresponding energy, E_{set}, are constants for all the power level transitions.

3.2.1 *The Proportional Scheme.* In this scheme, reclaimed slack is uniformly distributed to all remaining execution_segments 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 execution_segment, f_{next}, is computed as follows: the work for the remaining execution_segments is stretched out based on the remaining time to the deadline minus the time needed to switch the speed ($d - ct - T_{total}$, where ct is current time at the beginning of the execution_segment). Taking into consideration the overhead of changing the speed, this scheme computes a new speed as:

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

3.2.2 *The Greedy Scheme.* In this scheme all available slack is allocated to the next execution_segment. Similar to Proportional, we subtract the overhead from the remaining time after accounting for the execution time of the remaining execution_segments (excluding the current one) running at $f_{\max}$. Accounting for overheads, the new speed is computed as follows:

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

where $f_{\max}$ is the maximum frequency the processor can operate on.

[1]We conservatively consider that speed might need to be raised to the maximum speed to guarantee that deadlines are not violated.

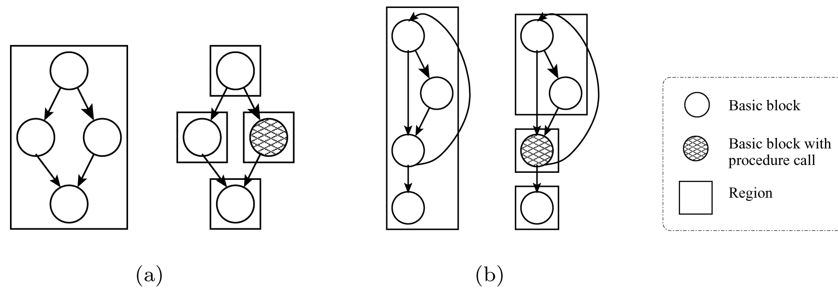


Fig. 4. Examples on region segmentation for (a) branches and (b) loops, with and without procedure calls.

Section 3.4.1 details the methodology for selecting the number of execution_segments and computing *wcc* for an application.

3.3 Timing Extraction

Before deciding on the locations of the PMHs in the code, prior timing knowledge about an application is needed. We assume that the WCET (the worst-case execution time) of applications is known or can be obtained (for example, through profiling or software timing analysis[2] [Vrchoticky 1994; Vivancos et al. 2001]). In this work, we use profiling to obtain timing information. In case a program's actual execution time exceeds the worst-case time determined from the training data, a signal can be invoked to indicate the violation of a deadline.

For collecting timing information, we divide a program's control flow graph (CFG) into *regions*. Each region contains one or more adjacent basic blocks (BB). Since the typical size of a basic block is small (tens of cycles) compared to the typical size of a PMP interval (hundreds of thousands of cycles), blocks can be aggregated into a region to be profiled. Region formation has the key advantage of reducing the number of traversed objects and the complexity of traversing the CFG in later stages of our power-management scheme. Region formation also avoids aggregating too many basic blocks into a region, such that the region execution at run-time exceeds PMP interval. Since some procedures called within a BB may execute for relatively long periods, we avoid overgrowing a region by isolating each BB that contains a procedure call into its own region. We aggregate the blocks into a region while maintaining the structure of the program constructs (for example, loops and control flow statements). Regions are determined by a region construction algorithm described in AbouGhazaleh et al. [2002].

Figure 4 shows examples of region formation and how it reduces the number of regions and preserves program structure. We show the cases where branches and loops contain procedure calls. For Figure 4a (a branch construct like an `if-then-else` statement), if none of the blocks forming the construct contain any procedure calls then all the blocks are merged into one region. Also, if

[2]The usual trade-off applies: profiling cannot guarantee the deadlines of tasks, since it does not derive the worst-case execution, while software analysis allows for that at the expense of lower slack utilization.

followed by a block without calls, then that block is merged with the larger region. However, if any of the BBs that belong to the branch construct contain a call, then this block forms a separate region not to be merged with other blocks. With respect to loops (Figure 4b), if the entire loop body does not include any calls, then it forms a single region that can be merged with other BB outside the loop. Similar to the case of branches with procedure calls, if any of the BBs has a call then the body is divided into more than a single region, and these regions cannot be merged with any region outside the loop.

After forming regions, profiling is used to extract timing information about the constructed regions. For loops, we collect the maximum cycle count of loop segments along with the maximum trip count of that loop, where *loop_segment* includes the total execution cycles of the loop iterations and the *trip count* is the number of iterations in the loop. We use loop_segment and the maximum trip count to estimate the *loop body* size, which is the number of cycles spent in a single iteration of the loop. For each procedure, we collect the time spent executing that procedure. High-level information (WCC and average cycle count) about the entire application is used to compute the PMP interval size.

Section 3.4 describes how to compute the PMP interval and the PMH placement as part of the offline phases of the collaborative scheme. The run-time phase of the scheme is described in Section 3.5.

3.4 Offline Placement

The compiler plays a major role in the offline phase, computing PMP interval and then placing the PMHs in the application code, based on information collected during the timing-extraction phase.

3.4.1 *Setting the PMP Interval.* Determining the PMP interval for invoking PMPs is based on the average and worst-case execution times of a program and the overhead of PMP executions. We present an approximation for finding the optimal number of PMPs to be executed to achieve minimum energy. We incorporate the time overhead of changing the speed, which includes computing and setting the speed, in the speed computation, extending our previous work, which only considered the energy overhead of the speed change in the computation [AbouGhazaleh et al. 2001]. We assume that (1) each execution_segment has a perfect execution behavior (that is, the actual execution times of all the execution_segments are equal to the average execution time of each execution_segment), and (2) the overhead, T_{total}, of computing and setting the speed at each PMP is constant. Below are the theoretical equations for the Greedy and Proportional speed setting schemes.

$$\textbf{Proportional scheme}: \ \phi_i = \frac{1}{n-i+1}\left(\frac{n}{load} - \frac{T_{total}}{wcc}\right)\prod_{k=1}^{i-1}\left[1 - \frac{\alpha}{n-k+1}\right] \tag{3}$$

$$\textbf{Greedy scheme}: \ \phi_i = \frac{1-(1-\alpha)^i}{\alpha} + \left(\frac{n}{load} - n - \frac{T_{total}}{wcc}\right)(1-\alpha)^{i-1} \tag{4}$$

where ϕ_i is the ratio $f_{\max}/f_i$ (f_i is the operating frequency for execution_segment i), n is the number of execution_segments in the longest execution path, *load* equals *WCET*/d, and α is the ratio of average to worst-case cycles in each execution_segment. To obtain n that corresponds to the minimum energy consumption in the average case, we substitute the formula of ϕ_i in the energy formula from Section 2 and solve iteratively for n. Hence, we get the PMP interval size by dividing the average number of cycles for executing an application by n. Below we show the derivations of these formulas starting from Eqs. (1) and (2).

3.4.1.1 *Derivation of the Proportional Scheme Formula.* We start from the Proportional scheme speed adjustment formula (Eq. 1):

$$f_i = \frac{WCR/f_{\max}}{d - ct - T_{total}} f_{\max}$$

Let *avgc*, ac_i be the average case, and actual case execution time of each execution_segment i, respectively, running at $f_{\max}$. Recall that our assumption for the theoretical model asserts that $ac_i = avgc$. Then *ct* equals the sum of the average execution times of the past segments divided by the segments' operating frequency normalized to the maximum frequency. Since d equals *WCET*/*load*, then d equals $n\ wcc/load$. Now, let $\alpha = avgc/wcc$. Then,

$$\begin{aligned}
f_i &= \frac{\sum_i^n wcc}{n\ wcc/load - \sum_{l=1}^{i-1}(avgc\ f_{\max}/f_l) - T_{total}} f_{\max} \\
&= \frac{(n-i+1)wcc}{n\ wcc/load - avgc\sum_{l=1}^{i-1}(f_{\max}/f_l) - T_{total}} f_{\max} \\
&= \frac{n-i+1}{n/load - \alpha\sum_{l=1}^{i-1}\phi_l - (T_{total}/wcc)} f_{\max} \\
f_{\max}/f_i &= \frac{1}{n-i+1}\left[\frac{n}{load} - \alpha\sum_{l=1}^{i-1}\phi_l - \frac{T_{total}}{wcc}\right] \\
\phi_i &= \frac{-\alpha}{n-i+1}\left[\frac{1}{-\alpha}\left(\frac{n}{load} - \frac{T_{total}}{wcc}\right) + \sum_{l=1}^{i-1}\phi_l\right]
\end{aligned}$$

Let $A_i = \frac{-\alpha}{n-i+1}$ and $B = \frac{1}{-\alpha}(\frac{n}{load} - \frac{T_{total}}{wcc})$. Starting from the above equation, we use the following lemma to derive Eq. (3). Proof of Lemma 1 is presented in the Appendix.

Lemma 1.

$$\phi_i = A_i\left(B + \sum_{l=1}^{i-1}\phi_l\right) \Longrightarrow \phi_i = A_i B \Pi_{l=1}^{i-1}(1 + A_l) \quad (5)$$

3.4.1.2 *Derivation of the Greedy Scheme Formula.* We start from the greedy scheme speed adjustment formula (Eq. 2):

$$f_i = \frac{wcc/f_{\max}}{d - ct - (WCR - wcc)/f_{\max} - T_{total}} f_{\max}$$

Using the same assumptions as in the Proportional scheme, we get:

$$\begin{aligned} f_i &= \frac{wcc}{n\, wcc/load \;-\; \sum_{l=1}^{i-1} avgc(f_{\max}/f_l) \;-\; (n-i)wcc - T_{total}}\; f_{\max} \\ &= \frac{wcc}{(n/load \;-n+i)\, wcc \;-\; avgc \sum_{l=1}^{i-1} (f_{\max}/f_l) \;-T_{total}}\; f_{\max} \\ &= \frac{1}{(n/load \;-n+i) \;-\; \alpha \sum_{l=1}^{i-1} (f_{\max}/f_l) \;-T_{total}/wcc}\; f_{\max} \\ \phi_i &= i + \left(\frac{n}{load} \;-n - \frac{T_{total}}{wcc}\right) - \alpha \sum_{l=1}^{i-1} \phi_l \end{aligned}$$

Let $C = -\alpha$ and $D = \frac{n}{load} - n - \frac{T_{total}}{wcc}$. Starting from the above equation, we use Lemma 2 to derive Eq. (4). Derivation of Lemma 2 is detailed in the Appendix.

LEMMA 2.

$$\phi_i = i + D + C \sum_{l=1}^{i-1} \phi_l \implies \phi_i = \frac{(1+C)^i - 1}{C} + D(1+C)^{i-1} \tag{6}$$

Since computing the speed is done in the OS, the OS needs to be periodically updated about the application's execution progress (through PMHs). Thus, the PMH placement in the application's code directly affects the slack utilization efficiency. Next we present our PMH placement technique.

3.4.2 *Placement of PMHs.* After computing the PMP interval, the goal of the PMH-placement algorithm is to ensure that, at least, a PMH is executed before the invocation of the next PMP. The placement algorithm traverses the CFG to insert PMHs no further apart than the size of the PMP interval (in cycles). Ideally, a PMH should execute right before the PMP is invoked, so that the PMP has the most accurate information. Since we do not control the application's dynamic behavior, we allow more than a single PMH to be executed in each execution_segment to improve the probability of an accurate speed computation.

In the PMH-placement algorithm [AbouGhazaleh et al. 2003a], while traversing the CFG of a procedure, a cycle counter *ac* is incremented by the value of the elapsed worst-case cycles of each traversed region. A PMH is inserted in the code just before this counter exceeds the PMP interval and the counter is reset. PMH locations are selected according to the different types of code structures in an application, namely sequential code, branches, loops or procedure calls. Next, we describe the criteria of placement in each of these cases.

3.4.2.1 *Sequential Code.* We define a *sequential segment* as a series of adjacent regions in the CFG that are not separated by branches, loops, joins, or back edges (although it may include a procedure call). Sequential placement inserts a PMH just before *ac* exceeds the PMP interval. It is nontrivial to insert a PMH in a region containing a procedure call, since the procedure's cycles are accounted for in the enclosing region's execution cycles. If the called procedure

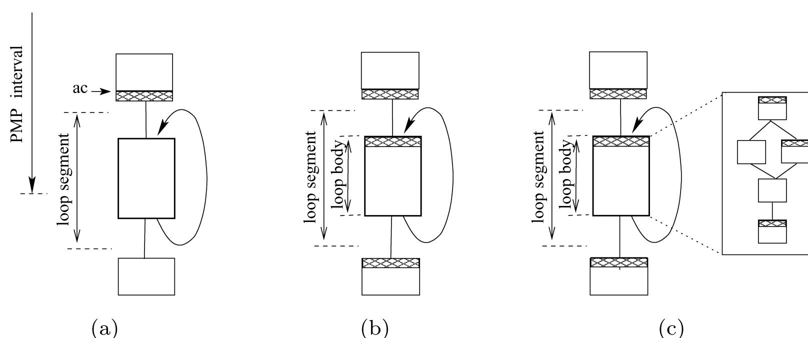


Fig. 5. PMH placement in loops when (a) ac + loop_segment > PMP interval, (b) loop_body < PMP interval < loop_segment, and (c) loop_body > PMP interval.

is too large (i.e., larger than the PMP interval), then inserting PMHs only at the region boundary is not sufficient to update the *WCR* before the next PMP invocation. For this reason, we need to further investigate possible locations inside a region related to the locations of procedure calls. For regions in a sequential segment, PMHs are inserted according to the following rules:

- When the cumulative counter *ac* exceeds the PMP interval cycles and there are no procedure calls in the region then a PMH is placed before the current region.
- If a region contains a procedure call and the body of a called procedure exceeds the PMP interval, a PMH is placed before the procedure call and another PMH after the procedure call. The called procedure is marked for later placement. The PMH before the call indicates the worst-case remaining cycles at the start of this procedure's execution. The PMHs before and after the called procedure—that contain PMHs—simplify the PMH placement scheme by avoiding the need to track interprocedural information about *ac* (i.e., *ac* does not have to be remembered from one procedure to the next).

3.4.2.2 *Branches.* For any branch structure, each path leaving a branch is treated as a sequential segment or recursively as a branch structure. At any branch, the value of *ac* is propagated to all the branch's paths. In a join, *ac* is set as the maximum value of all the propagated counters just before the join.

3.4.2.3 *Loops.* The decision of placing PMHs in a loop is based on the loop_segment and loop body sizes (in cycles). The different cases for inserting PMHs in loops are as follows:

- If the sum of *ac* and loop_segment exceeds PMP interval but the loop_segment is smaller than PMP interval, then a PMH is placed before the loop (see Figure 5a).
- If a loop_segment exceeds PMP interval, but the loop body is smaller than PMP interval, then a PMH is placed at the beginning of the loop body, in addition to the PMH placed before the loop. Another PMH is inserted after the loop exit (see Figure 5b).

- If the size of the loop body exceeds PMP interval, a PMH is placed at the start of the loop body and the loop is treated separately as either sequential code or code with branches. Another PMH is inserted after the loop exit (see Figure 5c).

The reason for placing a PMH after the loop in the last two cases is to adjust any overestimation of *WCR* done by the loop's PMHs. Nevertheless, overestimation of *WCR* is possible in the case where the actual number of loop iterations is unknown during loop execution.

3.4.2.4 *Procedure Calls.* As described above, in the processing of sequential segments, procedure calls are detected and accordingly some procedures are selected to undergo PMH placement in their bodies. The procedure selection is subject to satisfying the need for updating the *WCR* (through PMHs) at least once during each PMP_interval. For each procedure selected for placement, a PMH is placed before a procedure call to compute wcr_i of each instance of this procedure, and to store wcr_i in the procedure's activation frame. Instrumentation code is inserted in the procedure prolog to retrieve the value of wcr_i computed dynamically by the PMH located before the call. Each PMH located inside this procedure uses this value in its calculations of the remaining cycles. This instrumentation is necessary because a procedure may be called from different program paths and each with different wcr_i value even for the same called procedure (but different instance). This is especially beneficial in case of recursive calls, where each call instance has its own wcr_i estimate.

3.5 Run-Time Execution

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

3.5.1 *PMH Computation of wcr_i.* The placement algorithm can insert two different types of PMHs, *static* or *index-controlled* based on the program structure. The two types compute the worst-case remaining cycles based on whether the PMH is located inside a loop or not.

3.5.1.1 *Static PMH.* This type of PMH is placed in any location in the code outside a loop body. Generally, a PMH is located inside a procedure. A PMH computes wcr_i based on the remaining cycles at the start of the procedure instance, p_wcr. Since the path is only known at run-time, during execution, a procedure retrieves its p_wcr stored in its stack space. A static PMH computes wcr_i by subtracting the displacement of the PMH location in the procedure from p_wcr. For example, for a PMH inserted 100 cycles from the beginning of the procedure, the remaining number of cycles in the program is computed as: $wcr_i = p_wcr - 100$.

3.5.1.2 *Index-Controlled PMH.* PMHs of this type are inserted inside the body of a loop where wcr_i varies according to the current loop iteration

```
1   ct = read current time
2   f_curr = read current frequency
3   f_new = compute_speed (algo, ct, d, wcc, WCR )
4   if (f_new ≠ f_curr)
5       if (f_new ≤ f_min ) f_new = f_min
6       else if (f_new ≥ f_max ) f_new = f_max
7       else
8           set_speed ( discretize_speed (f_new) )
9           next_PMP = ct + (PMP-interval / f_new)
10          set_timer (next_PMP)
11  rti // return from interrupt
```

Fig. 6. ISR pseudocode for PMPs.

counter. The PMH computes the worst-case remaining cycle based on equations from Vivancos et al. [2001]. Using the worst-case cycles of a loop segment, *wc_loop_segment*, the term *wc_loop_segment*/*maximum_trip_count* (maximum number of iterations) estimates the size of a loop body, *loop_body*. Then, wcr_i is computed as *wcr_before_loop* − (*iteration_count* ∗ *loop_body*) − *ldisp*, where *wcr_before_loop* is the remaining cycle determined at run-time from the PMH that precedes the loop and *ldisp* is number of cycles elapsed since the start of the loop body in a single iteration. The same technique can also be applied in case of nested loops.

Estimating the loop's execution time using *wc_loop_segment* rather than using *wc_loop_body* * *maximum_trip_count* (this term denoted by *wc_lbody*) creates a tradeoff between energy savings and timeliness. Although accounting for the *wc_lbody* overestimates the loop execution time and thus increases the probability that a task will not miss its deadline.[3] This overestimation delays the slack availability until the loop finishes execution. On the other hand, using *wc_loop_segment* allows the slack to be used earlier in the execution, resulting in more energy savings. Even if the actual cycles of a loop iteration exceeds *loop_body*, the application is less likely to miss a deadline, in some cases: there is no large variation in the actual cycles of each iteration or the iteration with the duration of *wc_lbody* happens early in the execution that the remaining code would generate enough slack to avoid a deadline miss. Our experiments confirm that these cases are very common and thus no deadlines were missed. We conclude that, if the application is very sensitive to deadline misses, then index-controlled PMH computes the wcr_i as a function of the *wc_lbody*; otherwise, PMH computes it based on *wc_loop_segment*.

3.5.2 *ISR Execution of PMP.* In our scheme, the OS virtually divides the program into equal execution_segments by invoking the ISR to execute a PMP periodically. With the knowledge of the estimated *WCR* at each execution_segment, a dynamic speed-setting scheme computes the desired operating frequency for the next execution_segment (using the equations in Section 3.2) and then issues a speed-change request, if needed. The pseudocode of the ISR is listed in Figure 6. The ISR reads the current time, frequency, and *WCR*, and then computes a new speed. If the computed frequency is different than the

[3]Software analysis (in Section 3.3) does guarantee deadlines.

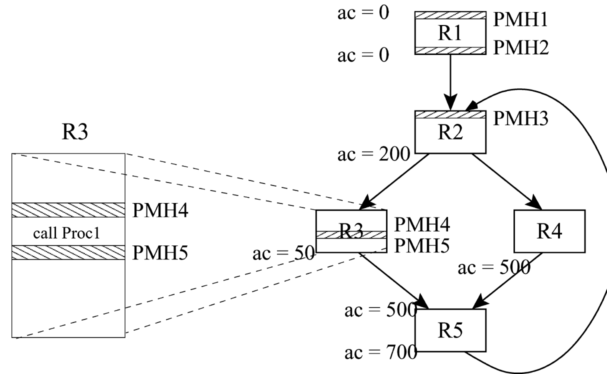


Fig. 7. Example of the PMH placement in a simple CFG.

Table II. Profiled *wcc* for Each Region in CFG Shown in Figure 7

Region	WCC (cycle)
1	500
2	200
3	1200
4	300
5	200

current one, the ISR searches for an appropriate frequency, invokes a speed-change call, adjusts PMP interval time, according to the new set frequency, and sets the timer for the next ISR invocation.

3.6 Example Showing the PMH Placement and Execution

We present an example that shows how the placement algorithm works for the simple CFG shown in Figure 7. The timing information for the CFG is listed in Tables II and III. Assume that the PMP interval is 1000 cycles. Below, we first give the details on how the algorithm selects the locations to insert PMHs and then the details of how each inserted PMH computes *WCR*.

- **PMH_1:** A PMH is placed at the beginning of the CFG, indicating the *WCR* (= 16,500 cycles) at the start of this procedure.
- **PMH_2 and PMH_3:** Since R2 starts a loop with a segment size of 16,000 cycles that is larger than PMP interval, PMH_2 is placed at the end of R1 (before the loop). Because the body of the loop exceeds PMP interval, PMH_3 is placed at the start of R2 (inside the loop) and, the loop body is traversed for further PMH placement (similar to case (c) in Figure 5).
- **PMH_4 and PMH_5:** Because the sum of *ac* and R3 cycles is larger than PMP interval, the algorithm looks for the first called procedure (i.e., `Proc1`) in region R3, which is located at 50 cycles from the beginning of R3. Since the procedure body is larger than PMP interval, PMH_4 and PMH_5 are placed before and after the procedure call, respectively. Procedure `Proc1` is marked

Table III. Other Profiled Information for CFG Shown in Figure 7

Loop body	1600 cycles
Max TripCounts	10 iterations
Procedure size	1100 cycles
disp of Proc1	50 cycles

Table IV. Details of the Inserted PMHs

PMH	Type	Computed as:
1	Static	p_wcr − 0 (= 16,500 for this procedure instance)
2	Static	p_wcr − 500 (= 16,000)
3	Index-controlled	16,000 − (iteration count × 1600)
4	Index-controlled	16,000 − (iteration count × 1600) − 250
5	Index-controlled	16,000 − (iteration count × 1600) − 1350

for later PMH placement. The PMHs placed inside procedure `Proc1` are not shown in this example.

Assuming that the presented CFG is the application's `main` procedure, *p_wcr* equals the application's worst-case remaining cycles (= 16,500). Table IV lists the details of the computation done by each inserted PMH. Note that PMH_2 is useful in evaluating *wcr_before_loop*.

4. OS SUPPORT FOR THE COLLABORATIVE SCHEME

To support our collaborative scheme on the OS side, the OS uses information provided by the compiler and accordingly schedule the speed change. To do so, we introduce an ISR for setting the speed [AbouGhazaleh et al. 2003b] and a system call to set the application's power-management parameters in the OS. In addition, more information is stored in the process's context to support the multiprocess preemption. Some details about each of OS extension needed are given below.

- **Power-management parameters initialization system call:** This system call inserted by the compiler sets the application's power-management parameters in the OS. The parameters include the length of PMP interval (in cycles), and the memory location of the WCR. This system call is called once at the start of each application. Although PMP interval is constant (in cycles) throughout the application execution, each ISR instance computes its equivalent time (to set the interrupt timer) based on the frequency set by this PMP instance. If there are multiple applications running, the context-switching mechanism adjusts these timers (see below).
- **Interrupt service routine:** According to the dynamic speed-setting scheme for computing a new operating frequency, the ISR selects an appropriate speed/power level to operate on. When working with discrete power levels, for a real-time application to meet its deadline, there are two possible ways to set the operating speed. One is to operate on the smallest power level larger than or equal to the desired frequency obtained from the speed-setting

Table V. Simplescalar Configuration

fetch width	4 instruction/cycle
decode width	4 instruction/cycle
issue width	4 out of order
commit width	4 instruction/cycle
RUU size	64 instruction
LSQ size	16 instruction
FUs	4 int, 1 int mult/divide, 4 fp, 1 fp mult/divide
branch pred.	bimododal, 2048 table size
L1 D-cache	512 sets, 32 byte block, 4 byte/block, 1 cycle, 4-way
L1 I-cache	512 sets, 32 byte block, 4 byte/block, 1 cycle, 4-way
L2 cache	1024 sets, 64 byte block, 4 byte/block, 1 cycle, 4-way
memory	40 cycle hit, 8 bytes bus width
TLBs	instruction:16 sets, 4096 byte page - data: 32 sets, 4096byte page

scheme. The second is to use the dual-speed-per-task scheme [Ishihara and Yasuura 1998]. The second scheme may be more efficient, depending on the overhead of two speed changes. However, in this work we choose to select the closest speed larger than the desired speed for simplicity.

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

  In case of multiple applications, every time the application is preempted, the timer value is stopped and the time value is reset when the application is resumed. For this, the PCB must contain a variable, *saved_PMP*, that saves these values: at preemption time, $saved_PMP = next_PMP - ct$ and at resume time, $next_PMP = ct + saved_PMP$, where *ct* is the current time.

  As mentioned above in Section 3, when the system allows for multitasking, our scheme will extract timing information (i.e., slack) from the task execution through the PMHs and the operating system will decide how to use such slack in intertask DVS [Mossé et al. 2000; Pillai and Shin 2001].

5. EVALUATION

Our evaluation of the collaborative scheme is divided into two parts. First, we compare our scheme against other compiler and OS intratask DVS schemes using simulations in Section 5.1. Second, we qualitatively analyze the advantages and disadvantages of our scheme against a compiler-directed intratask DVS scheme in Section 5.2.

5.1 Simulation Results

We evaluate the efficacy of our scheme in reducing the energy consumption of DVS processors using the SimpleScalar microarchitecture toolkit [Simplescalar 2001] with configurations shown in Table V. A dual-issue processor with similar

Table VI. Power Levels in the Transmeta Processors Model

Frequency (MHz)	700	666	633	600	566	533	500	466
Voltage (V)	1.65	1.65	1.60	1.60	1.55	1.55	1.50	1.50
Frequency (MHz)	433	400	366	333	300	266	233	200
Voltage (V)	1.45	1.40	1.35	1.30	1.25	1.20	1.15	1.10

Table VII. Power Levels in the Intel XScale Processor Model

Frequency (MHz)	1000	800	600	400	150
Voltage (V)	1.8	1.6	1.3	1.0	0.75

configurations was also tested and had the same energy-savings trends. Because these results are similar, we present only the results for the four-way issue machine.

We extended the *sim-outorder* simulator with a module to compute the dynamic and static energy of running an application as $E_{tot} = E_{dyn} + E_{static}$. Dynamic energy, E_{dyn}, is a function of the number of cycles, C, spent at each voltage level, V ($E_{dyn} = Pt = CV^2$, since $f \alpha \frac{1}{t}$ and $E = kCV^2$, where k is a constant; in this paper, we consider only $k = 1$). C includes the cycles for executing the application as well as the cycles for computing the speed in PMPs and wcr_i in PMHs. Static energy, E_{static}, is the integration of the static power dissipated until the deadline. Static power is assumed to be constant and equals 10% of the maximum power [Butts and Sohi 2000]. We also considered the overhead of setting the speed by adding a constant energy overhead for each speed change to the total energy consumption. In the experiments below, we used the Transmeta Crusoe and the Intel XScale CPU cores. The Transmeta Crusoe has 16 speed levels and the Intel XScale has 5 levels. Tables VI and VII show the different speeds and the corresponding voltages for both the Transmeta and Intel models. The energy consumed in other subsystems is beyond the scope of this evaluation.

We emulate the effect and overhead of the ISR in SimpleScalar by flushing the pipeline at the entry and exit of each ISR. The PMP (i.e., the ISR) computes a new frequency every PMP interval and then sets the corresponding voltage: the no-power-management (NPM) scheme that always executes at the maximum speed and static power-management (SPM) that slows down the CPU based on static slack [Aydin et al. 2001]. In addition, we evaluate our Greedy and Proportional power-management from Section 3.2.

We show the no-power-management (NPM) scheme that always executes at the maximum speed and static power-management (SPM) scheme that slows down the CPU based on static slack [Aydin et al. 2001]. To demonstrate the potential benefit of using PMHs, we compare the collaborative technique (PMP + PMH) with two schemes that use only PMPs: the first (*PMP-pcall*) places a PMP before each procedure call; the second (*PMP-plcmnt*) inserts PMPs according to the PMH placement algorithm.

In our benchmarks, the most frequently called procedures have simple code structures and are less likely to generate slack because of their small sizes. To

be fair to the PMP-call scheme (i.e., to reduce its overhead), we do not place PMPs before small procedures that do not have any call to other procedures. We show experimental results for three time-sensitive applications: automatic target recognition (ATR), an MPEG2 decoder, and a sub-band tuner. In the simulation, all the application's deadlines were met for the tested data sets.

5.1.1 *Impact on Energy and Performance.* Our results show that, as the time allotted (or the deadline) for an application to execute is increased, less dynamic energy is consumed because of the introduction of more slack that can be used to reduce the speed/voltage. However, as we extend the deadline, more static energy is consumed. Hence, the total energy is reduced because of savings in dynamic energy, but it can be increased owing to the increase in static energy. This is especially obvious when extending the deadline does not result in enough slack to lower the speed. When an application runs at the same frequency as with a tighter deadline, it consumes the same dynamic energy, but higher static energy. This is clear with the ATR application in the Intel case when the deadline exceeds 12 ms; the application consumes constant dynamic energy, but the static energy consumption increases as the deadline increases.

5.1.1.1 *Automatic Target Recognition.* The ATR application[4] does pattern matching of targets in input image frames. We experimented with 190 frames. The number of target detections in each frame varies from zero to eight detections; the measured frame processing time is proportional to the number of detections within a frame. In Figure 8, since Transmeta has more power levels than Intel, the static scheme for Intel is flat for several consecutive deadline values; e.g., for deadlines larger than 12 ms, the operating frequency is the same. At tight deadlines (up to 8 ms), the static scheme for the Transmeta processor executes the application at the highest speed, $f_{\max}$; this yields a higher-than-NPM energy consumption because of the overhead of the PMP executed at each data frame processing. This is not true for our Proportional and Greedy schemes because of dynamic slack reclamation. On the other hand, the same workload constitutes only about 80% of the Intel processor load because of a higher $f_{\max}$ than Transmeta.

When using the PMP-plcmnt and PMP + PMH schemes in the Transmeta model, Greedy and Proportional consume less energy than the static scheme because of dynamic slack reclamation. However, when using the Intel model, static power-management may consume less energy for some deadlines (12–21 ms for Proportional and 8 and 12–16 ms for Greedy). At those deadline values, the dynamic schemes operate most of the time on the same frequency as static power-management. These schemes do not have enough slack to operate at a lower performance level; however, the overhead of code instrumentation and voltage scaling increase the energy consumption of the dynamic schemes with respect to the static scheme.

The energy savings of the Greedy scheme relative to the Proportional scheme vary based on the deadline. When deadlines are short, Greedy consumes less

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

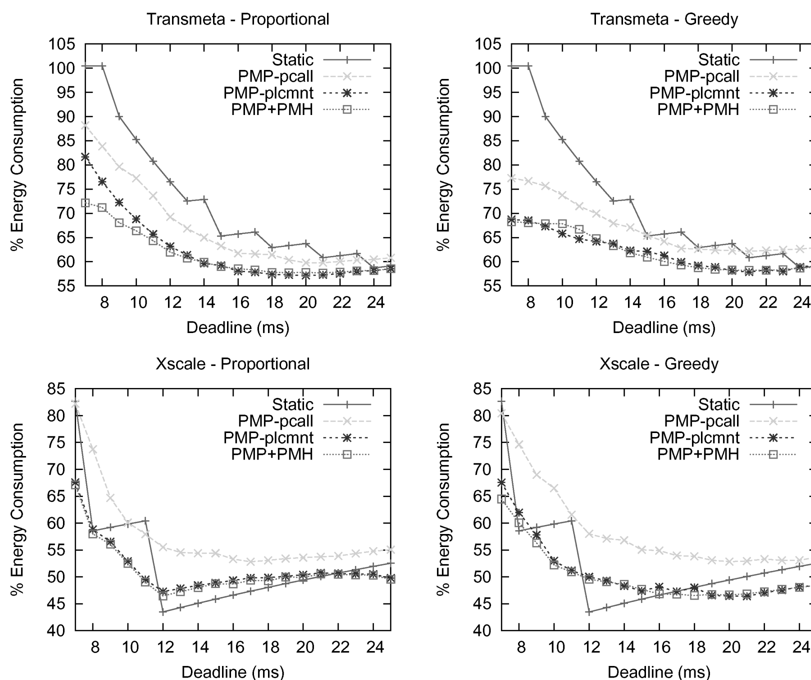


Fig. 8. Average energy consumption normalized to no-power-managemnt for ATR employing Transmeta Crusoe (top row) and Intel Xscale (bottom row) models.

energy than Proportional. Because Greedy is more aggressive in using slack than Proportional, it is more likely to select a lower speed earlier in the execution than Proportional. Once Greedy transitions to a low speed, the dynamic slack reclaimed from future intervals helps Greedy to stay at relatively low speed later in the execution. On the other hand, Proportional selects a medium range of speed levels throughout the execution. When increasing the deadline (more than 10 ms), Greedy exploits most of the static slack early in the execution to reach low speeds. However, the amount of dynamic slack generated later in the execution is not large enough to maintain these low speeds. Thus, the power manager needs to increase the speed causing Greedy to consume more energy than Proportional. In the case of the Intel processor, Greedy's energy consumption is lower than Proportional at more relaxed deadlines ($\geq$15 ms). This is due to the minimum speed level that prevents Greedy from exploiting all the slack at the beginning of execution.

In the Intel case, the static scheme outperforms Greedy scheme for 12–16 ms deadlines. This is because most of the speeds computed by Greedy scheme are slightly higher than 150 MHz but can only operate on 400 MHz, which is the same operating frequency for static power-management in this deadline range. Hence, both schemes run with almost the same power but because of the overhead incorporated with the dynamic PMP-only schemes, Greedy consumes more energy than the static scheme.

The PMP-plcmnt scheme has close energy consumption to our PMP + PMH scheme for ATR. There are two reasons for the result: high overhead because

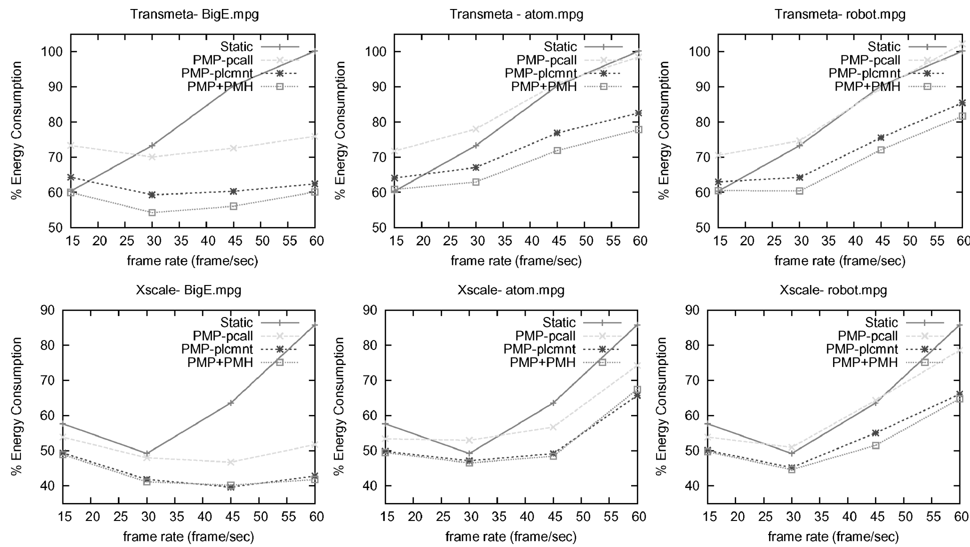


Fig. 9. Average energy consumption of Greedy PMP + PMH normalized to the no power-management scheme for the MPEG2 decoder employing Transmeta Crusoe (top row) and Intel XScale (bottom row) models.

of the large number of called procedures and the higher accuracy of the finer-granularity PMP invocation. It is also the case that most of the executed PMPs do not lead to a speed change, and thus there is no energy savings achieved from executing these PMPs.

5.1.1.2 *MPEG Video Decoder.* We collected timing information about the MPEG2 decoder [MSSG 2001] using a training data set of 6 movies and tested it on 20 different movies. Our scheme inserts PMHs in the decoder's code based on profile information about frames of the training set movies. We run experiments for four different frame rates: 15, 30, 45, and 60 frame/sec that correspond to deadlines 66, 33, 22, and 16 ms per frame, respectively.

Figure 9 shows the normalized energy consumption for three of the test movies for Transmeta (upper part of the figure) and Intel (lower part of the figure) models. For presentation simplicity, we only show the energy consumption for the Greedy scheme because Greedy outperforms the Proportional scheme. Similar to the ATR results, our proposed PMP + PMH scheme consumes less energy than the other schemes. The MPEG decoder code includes mainly a set of nested loops, most of which have large variation in their execution times. As a result, in our scheme, the number of executed PMHs is much larger than the invoked PMPs. Hence, the *WCR* estimation is improved over having only one PMH for each PMP invocation. In comparison with the PMP-plcmnt technique (replacing the PMHs with PMPs), PMP execution's energy overhead is higher, which overshadows the energy saving from the few extra speed adjustments. This is especially true for the Transmeta model. Because of the large number of speed levels that are close in range, this creates the chance for large number of speed changes in PMP-plcmnt. Each speed change results

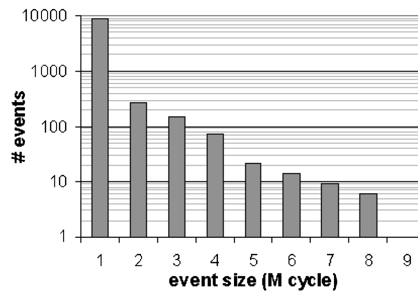


Fig. 10. Size distribution (in million cycles) for the sub-band filter events.

in small energy savings compared to the PMP energy overhead. The PMP-pcall scheme has the highest energy consumption because of the large number of procedure calls within loops in this application.

5.1.1.3 *Sub-Band Tuner.* A sub-band tuner[5] is a signal-processing application that accepts signals as input events. It uses time and frequency domain analysis to extract and process signals of interest, called peaks and ridges, from those events. The number of cycles per event processed varies based on the number of peaks and ridges detected in each event, but the deadline for each event is fixed. We experimented with a trace of 9853 events. From the collected timing information about each event processing time, we noticed a large variation between the average and worst-case execution times. An event's average execution time is 200 times smaller than its worst-case execution time. The size distribution is shown in Figure 10 (note the logarithmic scale for the Y-axis). To make good use of the dynamic slack for the average event, the scheme tailors the frequency of invoking a speed-change (PMP interval) based on the event's average execution time and PMHs are placed accordingly. As a result, events larger than average size would experience more PMH executions and speed-change invocations. This effect is prominent in case of very long events.

Figure 11 shows the energy consumption for the tested schemes. Among the dynamic schemes, PMP + PMH performs best. When comparing PMP + PMH with static, PMP + PMH performs well in high system loads (small deadlines). As the system load decreases, the difference in energy consumed by static and PMP + PMH decreases. For the Intel model, the static scheme performs better than PMP + PMH for very small loads, because of the large difference between the lowest two speed levels (400 and 150 MHz). When the static operates at 400 MHz, the dynamic schemes may have dynamic slacks to operate on less than this frequency, but not enough slack to operate at 150 MHz; therefore, the 400 MHz frequency is selected to avoid missing deadlines. The difference in energy consumption is because of the extra instrumentation added by the dynamic schemes; this is particularly true when there is a large variation in the execution times among different events, as with the sub-band tuner. In the Transmeta model, this does not happen since there are several frequencies

[5]The original code and data trace files for this application were also provided by our industrial research partners.

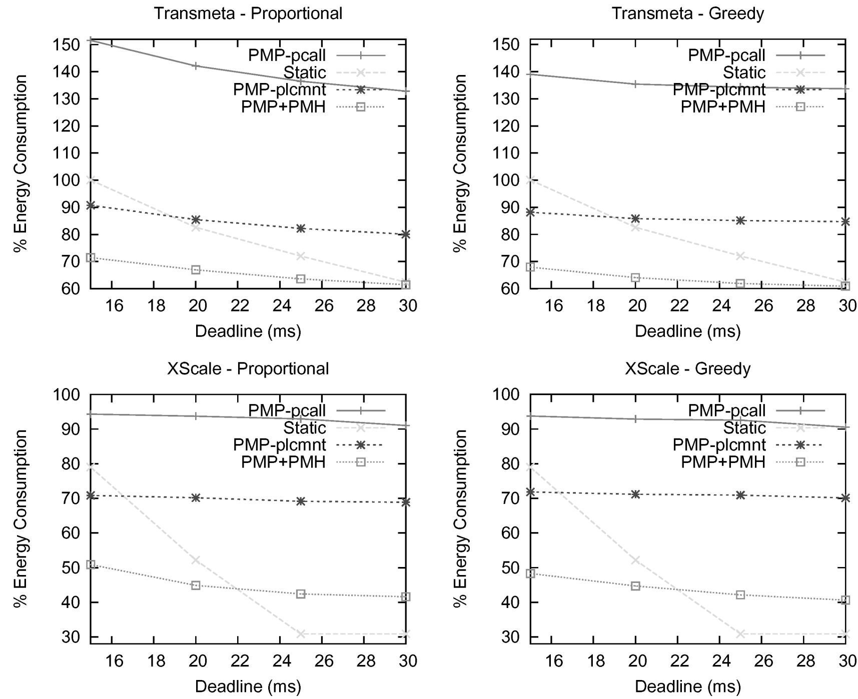


Fig. 11. Average energy consumption normalized to the no power-management scheme for the Sub-band filter application employing Transmeta Crusoe (top row) and Intel XScale (bottom row) models.

Table VIII. The Number of Executed PMPs, PMH, and Speed Changes for the Sub-Band Tuner

	No. PMHs Executed	No. PMPs Executed	No. Speed Changes
PMP-plcmnt	—	$\approx 19 \times 10^6$	$\approx 13.5 \times 10^3$
PMP + PMH	$\approx 19 \times 10^6$	$\approx 77.5 \times 10^3$	$\approx 11.8 \times 10^3$

between 200 and 400 MHz, allowing for the dynamic schemes to actually execute at lower frequencies.

Although PMP-plcmnt performs almost as well as PMP + PMH for the ATR and MPEG benchmarks, PMP-plcmnt performs much worse for this application. This is because of the large variation in execution times described earlier that required inserting relatively large number of PMHs. The overhead effect is significant in PMP-plcmnt, because the amount of dynamic slack between two consecutive PMPs is too small to compensate for this overhead. Table VIII compares the two techniques in term of the number of executed PMHs, PMPs, and actual speed change that takes place. We deduce that the work done by most of the executed PMPs (in PMP-plcmnt) was not useful. On the contrary, the overhead of placed PMHs in our scheme is minimal (few cycles) compared to PMP-plcmnt. Thus, the total energy consumption using PMHs is much less than inserting PMPs directly in the code.

5.1.2 *Run-Time Overhead.* Our scheme introduces overhead due to PMPs and PMHs. In this section, we further investigate the impact of each on overall performance (number of cycles and number of instructions).

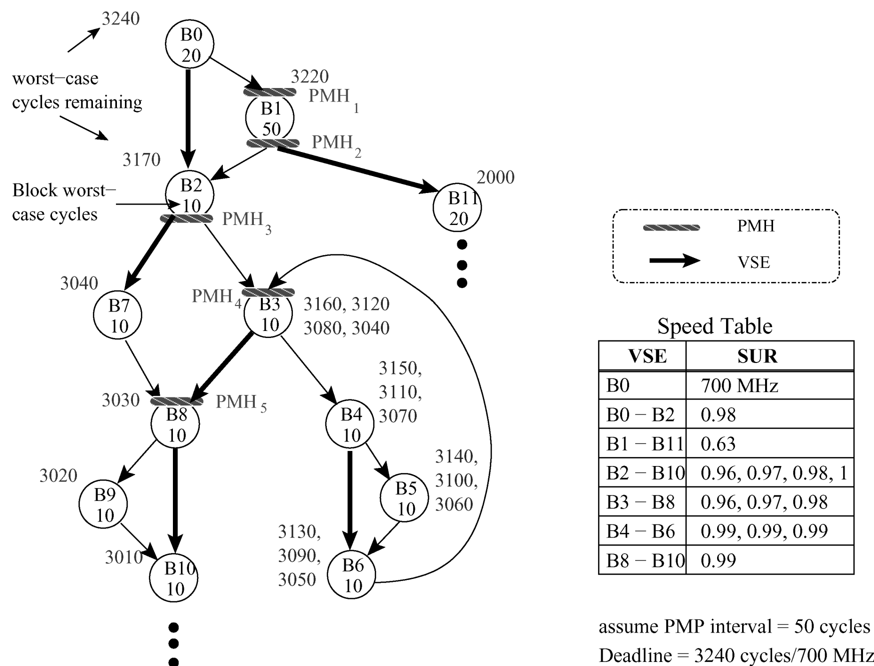
5.1.2.1 *Number of Instructions.* Increasing the number of instructions in the application by inserting PMPs and PMHs could have a negative effect on (1) the cache miss rate and (2) the total number of executed instructions. Both these factors could degrade performance. We measured their effects through an implementation in SimpleScalar. In our scheme, the measured effect of increasing the code size on the instruction cache miss rate is minimal, since the inserted PMH code is very small. There is no significant increase in cache misses (instruction and data) for our benchmarks. Hence, we do not expect a negative effect on the memory energy consumption because of our scheme. A static PMH takes 36 instructions to execute. Index-controlled PMHs are simplified to static PMHs inserted inside the loop with the addition of a division operation executed only once for each loop to compute the size of a loop body. The value of wcr_i is decremented in each iteration by the size of the loop body. Our measurements indicated that a PMP takes from 74 to 463 instructions to compute and select a speed (excluding the actual voltage change).

The number of executed instructions is kept minimal by (1) invoking the PMP at relatively large PMP intervals (it ranges from 50 to 580 K cycles in the presented applications) and (2) avoiding the excessive insertion of PMHs. For example, in the ATR application, for each PMP interval, only extra 505 instructions are executed, on average (including all executed PMHs and a PMP in this PMP interval[6]). The total increase in the number of executed instructions because of the overhead is 0.05% for ATR, 0.25% for MPEG, and 3.7% for sub-band tuner. With respect to other intratask DVS schemes, like PMP-plcmnt, the average increase in the number of executed instructions is <1% in ATR, 1.4% in MPEG, and 47% in sub-band tuner. The large increase in the number of executed instructions in sub-band tuner when using PMP-plcmnt is due to the excessive execution of PMPs (as described in Section 5.1.1).

5.1.2.2 *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 about 13 cycles and PMP takes from 22 to 229 cycles to execute,[7] excluding the actual voltage change). The total increase in the number of cycles ranges between 0.19–0.4% for ATR, 0.4–1.7% for MPEG, and 4.6–4.8% for sub-band tuner (the range corresponds to the low to high deadlines selected). In comparison to PMP-plcmnt, the average increase in the number of executed instructions is <1% in ATR, 1.5% in MPEG, and 64% in sub-band tuner. When using PMP-plcmnt, the execution of a large number of PMPs in sub-band tuner causes a relatively larger increase in the number of cycles.

We also note that the total overhead cycles decrease when the processor frequency is decreased. This is because the memory latency becomes relatively

[6]An inserted PMH can be executed more than once in a single PMP interval, for example, if placed inside a loop.
[7]Note this is a four-way architecture, executing about two instructions per cycle.



Speed Table

VSE	SUR
B0	700 MHz
B0 − B2	0.98
B1 − B11	0.63
B2 − B10	0.96, 0.97, 0.98, 1
B3 − B8	0.96, 0.97, 0.98
B4 − B6	0.99, 0.99, 0.99
B8 − B10	0.99

Fig. 12. Sample CFG and its constructed speed table.

small when compared to the CPU frequency; that is, the CPU stalls for fewer cycles (but still stalls for the same amount of time) waiting for data from memory.

5.2 Analysis of the Collaborative Scheme versus a Compiler Intratask DVS Scheme

In this section, we highlight the differences between the collaborative scheme and compiler-directed intratask DVS schemes. We compare our collaborative scheme with the scheme presented by Shin et al. [2001] as an example of an intratask DVS scheme. In Shin's scheme, the compiler constructs a CFG augmented with the *WCR* at each basic block and then selects the CFG edges where it inserts voltage scaling code. The selected voltage scaling edges (VSE) are the ones that drop the *WCR* more than the worst-case path at that branch. In Figure 12, we show an example of an augmented CFG and the selected VSE marked as bold arrows. This CFG is an extension of the example in Shin et al. [2001]. The compiler computes the speed slow-down factor (SUR) for each VSE and stores them in a *speed table*. The speed table for the example CFG is shown in Figure 12. At run-time, the new speed is computed at each VSE as $SUR * f_{curr}$, where f_{curr} is the current operating frequency. One of the advantages of this scheme is its low run-time overhead. Run-time overhead is reduced by both selecting the VSE and computing the SUR for each VSE (except after loop exit) at compile time.

The scheme in Shin et al. [2001] will work best when using continuous-frequency scaling, but it does not work as well when using a more realistic

processor with discrete speed levels. A solution would be to round the computed speed to the lowest speed level higher than the desired speed (as we follow in our scheme). Because of the speed rounding and the fact that the SUR is computed based on local information (about a specific branch), slack that is not used at any of the VSEs and is never reclaimed later in the execution. For example, if we consider the Transmeta model with an initial frequency of 700 MHz, then according to the speed table (computed offline), the scheme will not change the speed in any of the shown VSEs except in the B_1-B_{11} edge. For example, in the B_0-B_2 edge, since no speed change takes place ($f = 0.98 * 700 = 686$ and 686 MHz $>$ 666 MHz) then a 50-cycle slack is lost (3220-3170). Even at the B_1-B_{11} edge, the speed was reduced to 466 MHz, although the desired speed was 441 MHz. The slack resulting from the rounding to the higher discrete frequency is considered as lost slack. Furthermore, subsequent VSEs will scale down the speed relative to the 466 MHz rather than 441 MHz; hence, this extra speedup looses more slack.

Another reason for losing slack is because of scheduling the speed based on profile information rather than run-time information. Consider when the actual execution of B_1 takes 20 cycles rather than 50 cycles. The difference in both times can not be reclaimed as slack with an offline speed scheduling. Although profile information can give a good estimate for each basic block, the overhead of executing a VSE will prevent its insertion at every basic block boundary. Thus, the profiled time between consecutive VSE will not accurately reflect the actual execution time. Hence, the further apart the VSE are from each other, the more slack is lost.

In the collaborative scheme, we extract global information about the progress of an application (through PMHs) and decide the speed setting at run-time (through PMPs) based on the run-time behavior of the application. Since we accumulate the slack, at the time of PMP invocation, we can exploit all the slack generated since the last PMP. Figure 12 shows the PMH placement based on a PMP interval of 50 cycles. If execution follows the B_0-B_2-B_3-B_8-B_{10} path (and the actual execution of the basic blocks matches the profiled measurements), a PMP is invoked after B_8 finishes execution; PMH_5 sets WCR to 3030 cycles. The computed speed is then set to $\frac{3030}{3240-50} * 700 = 664$ MHz, which is rounded to 666 MHz. The scheme was able to reduce the speed to the next lower level because it had a more accurate estimate of the actual available slack.

Scheduling the speed based on knowledge of the overall application's *WCR*, the *current time* and the *deadline* (Eqs. 1 and 2) account for the dynamic slack due to the application's execution time variation and the speed rounding. For example, the speed rounding from 664 to 666 MHz creates extra slack that will be used at the next PMP as this speedup will be reflected in the *current time*. Also, when the actual execution is less than the profile measurements, this difference will show as a slack when using the *current time*. Thus, the collaborative scheme succeeds in detecting slack that was not used by the compiler-only scheme.

On the other hand, the collaborative scheme may not report the exact amount of slack at the time of a PMP invocation owing to the lack of perfect

synchronization between PMHs and PMPs. For example, the PMP executed after B_8, in our example, has an actual slack of 220 cycles (3240–3020), but PMH_5 reported only 210 cycles (3240–3030). Although this slack difference is not exploited at this PMP, it will be used, along with any newly generated slack, at the next PMP. Also, in contrast to the VSE execution, the lack of perfect PMH–PMP synchronization causes the slack to be used only when a PMP is invoked and not when the slack is generated.

In summary, a compiler-directed intratask DVS scheme will outperform the collaborative scheme when both (1) there are plenty of CPU speed levels and (2) the overhead of the speed change is insignificant so that executing frequent VSEs does not negatively affect energy consumption. Otherwise, based on the frequency of executing VSEs, the energy consumption would fall between the PMP-pcall and the PMP-plcmnt schemes presented in the paper.

6. RELATED WORK

There have been a number of research projects that use DVS schemes to reduce the processor's energy consumption. Examples on work that implements DVS in non-time-critical applications include Childers et al. [2000] and Hsu and Kremer [2002]. An operating system solution proposed in Childers et al. [2000], periodically invokes an interrupt that adjusts the processor speed in order to maintain a desired performance goal. The OS keeps track of the accumulated application's instruction-level parallelism throughout the application execution time. In Hsu and Kremer [2002], the compiler identifies program regions where the CPU can be slowed down. The speed set in each region is based on the expected time the CPU would wait for a memory access. An implementation for this scheme and a physical measurement of the consumed energy on a real system was presented in Hsu and Kremer [2003]. However, this work did not consider real-time constraints and did not exploit slack time generated in computation-dominated applications. An analytical study of the bounds on energy savings from using intratask DVS is presented in Xie et al. [2003]. Work presented in Saputra et al. [2002] selects the best supply voltage for each loop nest based on simulated energy consumed in a loop nest. The voltage levels are set at compile time for each region using an integer linear programming DVS strategy.

Time restrictions in time-sensitive applications mandate the processor to finish the application's execution before its deadline. On the OS level, Gruian [2001] determines a voltage scheduling that changes the speed within the same task/application based on task's statistical behavior. The work in Pillai and Shin [2001] modified the EDF and RMS in RT-Linux to incorporate voltage scheduling.

Shin et al. [2001] and Azevedo et al. [2002] apply a compiler-controlled DVS in real-time applications. In Azevedo et al. [2002], 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. Run-time

overhead of updating data structures and setting the new voltages is relatively high, especially on the constructed nodes granularity (almost every basic block). In Shin et al. [2001], the compiler selects a set of branches to insert the voltage change calls. It computes and stores the slowdown factor at each voltage change branch. In addition to not exploiting all the generated slack when operating at a discrete speed levels, the scheme has a larger memory footprint. Since procedures are called from different paths with different *WCR*, more memory is needed to store the speed-slowdown factors (computed offline) for each possible procedure instance. Alternatively, the slowdown factor can be dynamically computed at the expense of increasing the run-time overhead.

In Kim et al. [2002], 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 [Shin et al. 2001; Gruian 2001] 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 Puschner and Burns [2000]. Some of these tools use static analysis to produce a discrete WCET bound. On the other hand, parametric analysis for computing WCET as in Vivancos et al. [2001] evaluates expressions in terms of parameter variables carrying information about some program segments, e.g., WCET of loops is presented by symbolic formulas that are evaluated at run-time when the number of loop iterations is determined.

7. CONCLUSION

In this paper, we presented a hybrid compiler operating system intratask DVS scheme for reducing the energy consumption of time-sensitive embedded applications. The operating system periodically invokes *power-management points* (PMPs) to adapt processor performance based on the dynamic behavior of an application. Information about run-time temporal behavior of the application is gathered by very low cost *power-management hints* (PMHs) inserted by the compiler in the application. We described our collaborative scheme and its advantages over a purely compiler-based approach. We also presented an algorithm for inserting PMHs in the application code to collect accurate timing information for the operating system. 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. Finally, we presented results that show the effectiveness of our scheme. Our results showed that our scheme is up to 57% better than no-power-management and up to 32% better than static power-management on several embedded applications. Furthermore, we demonstrated that our compiler-directed scheme can achieve better energy reductions than a scheme that relies on the programmer and semantic information about an application (see AbouGhazaleh et al. [2002] for more details).

APPENDIX

A.1 Derivation for Lemma 1

LEMMA A.1

$$\phi_i = A_i \left(B + \sum_{l=1}^{i-1} \phi_l \right) \Longrightarrow \phi_i = A_i B \Pi_{l=1}^{i-1} (1 + A_l) \tag{7}$$

PROOF (BY INDUCTION)

Base case: at $i = 1$, it is trivial to see that the left-hand side (LHS) of Eq. (7) is the same as the right-hand side (RHS):

$$LHS = A_1 B = RHS$$

Induction step: Let Eq. (7) hold for all $n < i$. We prove that it also holds for $n = i$. By substituting the RHS of Eq. (7) for ϕ_l, it is sufficient to prove that:

$$\phi_i = A_i \left(B + \sum_{l=1}^{i-1} A_l B \Pi_{k=1}^{l-1} (1 + A_k) \right) = A_i B \Pi_{l=1}^{i-1} (1 + A_l)$$

as below:

$$\begin{aligned}
\phi_i &= A_i \left(B + \sum_{l=1}^{i-1} A_l B \Pi_{k=1}^{l-1} (1 + A_k) \right) \\
&= A_i B \left(1 + \sum_{l=1}^{i-1} A_l \Pi_{k=1}^{l-1} (1 + A_k) \right) \\
&= A_i B \left(1 + A_1 + \sum_{l=2}^{i-1} A_l \Pi_{k=1}^{l-1} (1 + A_k) \right) \\
&= A_i B \left((1 + A_1) + \sum_{l=2}^{i-1} A_l (1 + A_1) \Pi_{k=2}^{l-1} (1 + A_k) \right) \\
&= A_i B \left((1 + A_1) + A_2 + (1 + A_1) \ + \ \sum_{l=3}^{i-1} A_l \Pi_{k=2}^{l-1} (1 + A_k) \right) \\
&= A_i B \left((1 + A_1)(1 + A_2) \ + \ \sum_{l=3}^{i-1} A_l \Pi_{k=2}^{l-1} (1 + A_k)) \right) \\
&= \ldots . \\
&= A_i B ((1 + A_1)(1 + A_2). \ldots .(1 + A_{i-1})) \\
&= A_i B \Pi_{l=1}^{i-1} (1 + A_l) \\
&= RHS
\end{aligned}$$

End of Lemma 1.

A.2 Derivation for Lemma 2

LEMMA A.2

$$\phi_i = i + D + C\sum_{l=1}^{i-1}\phi_l \implies \phi_i = \frac{(1+C)^i - 1}{C} + D(1+C)^{i-1} \quad (8)$$

PROOF (BY INDUCTION)

Base case: at $i = 1$, it is trivial to see that LHS of Eq. (8) is the same as the RHS:

$$LHS = \phi_1 = 1 + D = RHS$$

Induction step: Let Eq. (8) hold for all $n < i$. We prove that it also holds for $n = i$. By substituting the RHS of Eq. (8) for ϕ_l, it is sufficient to prove that:

$$\phi_i = i + D + C\sum_{l=1}^{i-1}\frac{(1+C)^l - 1}{C} + D(1+C)^{l-1} = \frac{(1+C)^i - 1}{C} + D(1+C)^{i-1}$$

as follows:

$$\begin{aligned}
\phi_i &= i + D + C\sum_{l=1}^{i-1}\left[\frac{(1+C)^l - 1}{C} + D(1+C)^{l-1}\right] \\
&= i + D + \sum_{l=1}^{i-1}((1+C)^l - 1) + DC\sum_{l=1}^{i-1}(1+C)^{l-1} \\
&= i + D - (i-1) + \sum_{l=1}^{i-1}(1+C)^l + DC\sum_{l=1}^{i-1}(1+C)^{l-1} \\
&= D + 1 + \frac{(1+C)^i - 1}{C} - 1 + DC\frac{(1+C)^{i-1} - 1}{C} \\
&= \frac{(1+C)^i - 1}{C} + D(1+C)^{i-1} \\
&= RHS
\end{aligned}$$

End of Lemma 2.

REFERENCES

ABOUGHAZALEH, N., MOSSE, D., CHILDERS, B., AND MELHEM, R. 2001. Toward the placement of power management points in real-time applications. In *COLP'01: Workshop on Compilers and Operating Systems for Low Power*. IEEE press, Piscataway, NJ.

ABOUGHAZALEH, N., CHILDERS, B., MOSSE, D., MELHEM, R., AND CRAVEN, M. 2002. Collaborative compiler-OS power management for time-sensitive applications. Tech. Rep. TR-02-103, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA.

ABOUGHAZALEH, N., CHILDERS, B., MOSSE, D., MELHEM, R., AND CRAVEN, M. 2003a. Energy management for real-time embedded applications with compiler Support. In *LCTES'03: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems*. ACM Press, New York. 284–293.

ABOUGHAZALEH, N., MOSSE, D., CHILDERS, B., MELHEM, R., AND CRAVEN, M. 2003b. Collaborative operating system and compiler power management for real-time applications. In *RTAS'03: IEEE Real-Time Embedded Technology and Applications Symposium*. IEEE Press, Piscataway, NJ.

AYDIN, H., MELHEM, R., MOSSE, D., AND ALVAREZ, P. 2001. Determining optimal Processor Speeds for Periodic Real-time Tasks with Different Power Characteristics. In *ECRT'01: IEEE Euromicro Conference on Real-Time Systems*. IEEE Press, Piscataway, NJ.

AZEVEDO, A., ISSENIN, I., CORNEA, R., GUPTA, R., DUTT, N., VEIDENBAUM, A., AND NICOLAU, A. 2002. Profile-based dynamic voltage scheduling using program checkpoints. In *DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE Computer Society, Washington, DC. 168.

BUTTS, J. A. AND SOHI, G. S. 2000. A static power model for architects. In *MICRO 33: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, New York. 191–201.

CHILDERS, B., TANG, H., AND MELHEM, R. 2000. Adapting processor supply voltage to instruction-level parallelism. In *Koolchips 2000 Workshop, During 33th Annual International Symposium on Microarchitecture (MICRO-33)*.

GRUIAN, F. 2001. On energy reduction in hard real-time systems containing tasks with stochastic execution times. In *IEEE Workshop on Power Management for Real-Time and Embedded Systems*.

HSU, C. AND KREMER, U. 2002. Single vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches. In *Power Aware Computer Systems (PACS)*. ACM Press, New York.

HSU, C. AND KREMER, U. 2003. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI'03: Programming Language Design and Implementation*. ACM Press, New York.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 International Symposium on Low Power Electronics and Design*. ACM Press, New York. 197–202.

KIM, W., SHIN, D., YUN, H., KIM, J., AND MIN, S. 2002. Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In *Real-Time and Embedded Technology and Applications Symposium.* IEEE press, Piscataway, NJ.

MAHALANOBIS, A., VIJAYA KUMAR, B. V. K., AND SIMS, S. R. F. 1996. Distance-classifier correlation filters for multiclass target recognition. *Optical Society of America 35,* 7 (June), 3127.

MIN, R., FURRER, T., AND CHANDRAKASAN, A. 2000. Dynamic voltage scaling techniques for distributed micro-sensor networks. In *IEEE VLSI Workshop*. IEEE Press, Piscataway, NJ.

MOSSÉ, D., AYDIN, H., CHILDERS, B., AND MELHEM, R. 2000. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *COLP'00: Workshop on Compilers and Operating Systems for Low Power*. IEEE Press, Piscataway, NJ.

MSSG. 2001. MPEG software simulation group, MPEG2 decoder source code. http://www.mpeg.org/ MPEG/MSSG.

PERING, T., BURD, T., AND BRODERSEN, R. 2000. Voltage scheduling in the lpARM microprocessor system. In *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design*. ACM Press, New York. 96–101.

PILLAI, P. AND SHIN, K. G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP'01: 18th ACM Symposium on Operating Systems Principles*. ACM Press, New York.

PUSCHNER, P. AND BURNS, A. 2000. *Guest Editorial: A Review of Worst-Case Execution-Time Analysis*. Vol. 18. Kluwer Academic Publ. Norwell, MA.

RICKARD, D., BERGER, R., CHAN, E., CLEGG, B., PATTON, S., ANDERSON, R., BROWN, R., SYLVESTER, D., GUTHAUS, M., DEOGUN, H., LIU, K. J. R., PANDANA, C., AND CHANDRACHOODAN, N. 2003. BAE Systems mission specific processor technology. In *GOMAC'03: 28th Annual Government Microcircuit Applications and Critical Technology Conference*.

SAPUTRA, H., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M. J., HU, J. S., HSU, C.-H., AND KREMER, U. 2002. Energy-conscious compilation based on voltage scaling. In *LCTES/SCOPES'02: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*. ACM Press, New York. 2–11.

SHIN, D., KIM, J., AND LEE., S. 2001. Intra-task voltage scheduling for low-energy hard real-time application. In *IEEE Design and Test of Computers*. IEEE Press, Piscataway, NJ.

SIMPLESCALAR. 2001. Architecture simulator. http://www.simplescalar.com.

TRANSMETA. 2002. Crusoe processor specification. http://www.transmeta.com.

VIVANCOS, E., HEALY, C., MUELLER, F., AND WHALLEY, D. 2001. Parametric timing analysis. In *LCTES'01: Workshop on Language, Compilers, and Tools for Embedded Systems*. ACM Press, New York.

VRCHOTICKY, A. 1994. Compilation support for fine-grained execution time analysis. Tech. rep., Technical University of Vienna.

XIE, F., MARTONOSI, M., AND MALIK, S. 2003. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM Press, New York. 49–62.

XSCALE. 2002. Intel XScale processors. http://developer.intel.com/design/intelxscale.

Received January 2004; revised April 2005; accepted May 2005