Energy Management for Real-Time Embedded Applications with Compiler Support *

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

ABSTRACT

Reducing device energy has become one of the most important challenges to embedded systems designers. Processors with dynamic voltage scaling permit trading performance for reduced energy consumption as a program executes. In this paper, we first present a novel hybrid scheme that uses dynamic voltage scaling to adjust the performance of embedded applications to reduce energy consumption while also meeting time constraints. Our fine-grained approach uses the compiler to insert power management hints *in the application code. These hints convey path-specific run-time information about the program's progress to* power management points *invoked by the operating system that adjust processor performance. Second, we present an algorithm for inserting power management hints along different program paths. Finally, we experimentally evaluate our approach and show that significant energy reduction can be achieved. On two embedded applications, MPEG movie decoding and automatic target recognition, our scheme reduces energy by up to 79% over no power management and by up to 50% over static power management. We also experimentally demonstrate that our scheme achieves more energy savings compared to two purely compiler-directed schemes.*

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*

*This work has been supported by the Defense Advanced Research Projects Agency through Power Aware Real-Time Systems (PARTS) group under contract F33615-00C-1736.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro£t or commercial advantage and that copies bear this notice and the full citation on the £rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior speci£c permission and/or a fee.
LCTES'03, June 11–13, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-647-1/03/0006 ...$5.00.

General Terms

Algorithms, Management, Experimentation

Keywords

Real-Time, Dynamic Voltage Scaling, Power management

1. INTRODUCTION

Energy consumption has become vitally important to battery-operated portable and embedded systems. By lowering energy consumption, battery life and mission duration is extended and more capabilities can be included in a device for the same battery capacity. Furthermore, reduced energy results in lighter devices and less expensive packaging. These factors reduce the total cost of building portable and embedded systems.

Embedded systems are experiencing explosive growth in sales volume with estimates of up to 400,000 cell phones sold per day by 2006 [18] and up to 60 million personal digital assistants (PDAs) sold in 2008 [19]. Indeed, embedded computer systems can be found in a number of domains, ranging from military devices to inexpensive consumer electronics to complex automobile control systems. One distinguishing and common characteristic of such systems is their sensitivity to cost, particularly energy consumption.

A number of these computer systems include applications that have time constraints that must be satisfied during an application's execution. An example of such a time-sensitive application is the MPEG decoder which displays movies with a given frame rate. Other time constrained applications include digital speech coding, Doppler radar-based cruise control and collision warning for automobiles, face screening, living maps for driving direction, voice recognition, highway sensor networks for telematics, and automatic target recognition.

There have been several techniques proposed for managing energy in portable and embedded computer systems. One solution that has been adopted by a number of processor manufacturers, including Intel, AMD, and Transmeta, is *dynamic voltage scaling* (DVS). In a processor that has DVS, the clock frequency and supply voltage can be changed on-the-fly to make a trade off between performance and energy consumption.

While most DVS techniques exploit variability in load across an entire set of applications, in this paper we propose an approach that uses fine-grain information about

execution variability within an application to reduce energy consumption. The technique exploits timing variability in program paths, resulting from constructs such as loops with variable trip counts. Ours is a hybrid approach that relies on both the compiler and the operating system to adapt performance and reduce energy consumption of the processor. The compiler conveys path-specific run-time information about a program's progress to the operating system. These so-called *power management hints* (PMH) are inserted by the compiler in the code, based on program structure and estimated worst-case performance. A PMH is very low cost instrumentation that collects path-specific information for the operating system about how the program is behaving relative to worst-case performance. The operating system periodically invokes a *power management point* (PMP) to change the processor's performance based on the timing information from the power management hints. This collaborative approach has the advantage that the lightweight hints can collect accurate timing information for the operating system without actually changing performance. Further, the periodicity of performance/energy adaptation can be controlled independently of power management hints to better balance the high overhead of adaptation. Also, as we show with two different algorithms, a collaborative scheme separates hint insertion from performance adaptation, permitting the use of different algorithms in the operating system to make trade offs between performance and energy consumption. In this way, energy consumption can be reduced by running at the lowest performance settings based on actual program performance while still meeting time constraints.

We evaluate our technique on two time-sensitive applications; an MPEG-2 decoder and an automatic target recognition (ATR) application. Using models of commercially available processors with dynamic voltage scaling, the Transmeta's Crusoe and Intel's XScale, we show that our technique can achieve significant energy reductions of up to 79% over no power management and up to 50% over static power management. We further demonstrate that our collaborative scheme achieves less energy consumption than a scheme that uses a simple compiler placement of PMPs at each procedure call and even better than a scheme that uses PMPs only placed in code using our hint-placement algorithm.

In the next section, we review why DVS is useful for reducing the energy consumption of applications with time constraints. We then explain our collaborative compiler-operating system scheme in Section 3 and the compiler's role in inserting power management hints in Section 4. A set of experimental results shows the effectiveness of our approach in Section 5. Related work is discussed in Section 6 and we conclude the paper in Section 7.

2. DVS AND APPLICATION MODELS

Several commercial processors have recently included DVS capabilities to make trade-offs between performance and energy consumption. 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 related to 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 slower with reduced V_{dd} and frequency leads to energy savings.

Two examples of DVS processors come from Transmeta and Intel. The Transmeta Crusoe [22] 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 [20] has fewer levels and larger steps. The voltages range from 0.75V to 1.8V (150 MHz to 1 GHz with a 250 then 200 MHz step). In all DVS processors, a transition from one power level to another has a time and energy overhead. The overhead results from changing the supply voltage and allowing it to become stable. This overhead ranges from 25μ sec to 150μ sec and consumes up to 4μJ [14, 9]. For the rest of this paper, we refer to changing the voltage and frequency as a "speed change".

We assume that a time-sensitive application has a certain allotted time frame by which it should finish execution, even when applying DVS. This allotted time usually comes from CPU reservations in real-time operating systems (OSs) or from engineering time-sensitive embedded systems. Each application is characterized by its *worst-case execution time*. To avoid violating the time constraint, an application is scheduled assuming it will run for its worst-case duration. 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 is expressed in cycles rather than time units; i.e., the application duration is represented by *worst case cycles* (WCC).

3. COMPILER & OS COLLABORATION

To use DVS, system designers must define how to initiate a speed/voltage change and how to select a speed/voltage level. Automatically deciding on the proper locations to insert PMPs by the compiler in an application code is not trivial. One problem is how frequently the speed should be changed. Ideally, the more voltage scaling invocations, the more the application can exploit dynamic slack, and further reduce energy. However, we do not have this freedom, as there is energy and time overhead associated with each speed adjustment. In [1] a special case was considered in which a program's code is composed of sequential blocks. We showed that too many PMP invocations would hurt as much as too few of them. Our goal was to determine how many PMPs should be inserted in the code to minimize energy consumption while taking into account the speed/voltage scheduling overhead. We presented a theoretical solution that determines how far apart (in cycles) any two PMPs should be placed. With sequential code and an estimate of instruction latency, we could then insert PMPs into the code. Beyond that preliminary work (i.e., applications with sequential code), the problem in real applications 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 gen-

eral, the overhead of DVS should be kept to a minimum by avoiding excessive PMP invocations.

To control the overhead of speed scheduling during the execution of an application, we use a compiler-directed technique. During compile time, the compiler inserts instrumentation code (PMH) that computes information about the *worst-case remaining* cycles (WCR) of the application. During execution, a PMH passes timing information to the operating system. To keep the overhead of hints as low as possible, it updates the WCR in a predetermined memory location or a special register, rather than invoking the OS or performing any speed changes. Periodically, the operating system is invoked by a watchdog timer interrupt to execute a PMP. The time interval for the watchdog timer is determined at compile-time based on minimizing the number of PMPs executed for an application, yet achieve as low energy consumption as possible, as in [1]. The timer interrupt is armed with an *interrupt service routine* (ISR) that does the PMP; that is, the ISR adjusts processor speed based on the latest WCR information (provided by the hints). 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 minimize energy consumption while meeting time constraints. 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.

The advantage to using a collaborative scheme that includes both PMHs and PMPs is that timing information can be inexpensively collected without actually doing a speed change and incurring its high overhead. Since the overhead of executing a hint (setting a register or writing to memory) is much less than executing a PMP (computing a new frequency and adjusting the supply voltage), we have more freedom to add hints in the code as often as necessary to enhance the accuracy of updating the WCR.

3.1 The Collaborative OS Role

In order to implement the collaborative scheme, the OS requires a definition for a new ISR for adjusting the processor speed and two system calls to communicate with the running application. First, the ISR reads the current WCR and computes the speed according to the selected dynamic speed setting scheme from [7, 1]. Then, the ISR issues a speed change request if needed. Second, the system calls transfer the timing information from the application to the OS. Two system calls are needed that are called only once at the start of an application: (1) a system call that gives the address of the buffer that holds timing information collected by hints, so that whenever a PMP is invoked, it can find the WCR in this location, and (2) another system call is called to set the ISR interval (in cycles).

Figure 1 shows how PMHs and PMPs work together. In this figure, a PMP interrupt service routine is periodically invoked (the large bars) to adjust processor speed based on WCR. The power management hints are executed at some point before a PMP to update the WCR based on a path of execution. The task of the compiler is to insert the hints in the application code in such a way that guarantees that a PMH is always executed before a PMP is invoked. However, note that more than one hint can be executed before the PMP without harm. The compiler is essentially mapping the time interval to the static program code and inserting at least one PMH in that interval. This work deals with non-preemptive applications executing in the system, which is the case for many embedded systems.

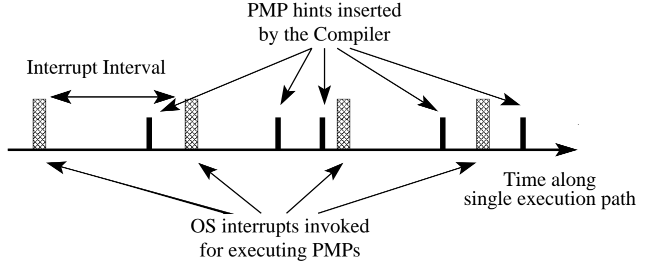

Figure 1: Invocations of PMHs and PMPs for a specific path.

In case of process preemption, the OS should keep track of the operating frequency and the interrupt interval status as part of the context of the departing process (application). This information gets replaced by the corresponding values of the newly dispatched process.

3.2 The Compiler Role

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 j [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. The next section presents our algorithm for inserting power management hints.

4. PMH PLACEMENT SCHEME

Our algorithm for energy management has four steps: (1) extraction of timing information, (2) determination of the PMP interrupt interval, (3) PMH insertion by the compiler, and (4) run-time execution of PMHs and PMPs. Initially, timing extraction is done to gather timing information about the application (Section 4.1). Next, using the

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

timing information, an interval of how often the OS should invoke a PMP is computed (Section 4.2). Then, the compiler inserts PMHs in the code at specific locations (Section 4.3). Lastly, hints are executed at run-time to update the WCR based on actual execution paths taken through the code (Section 4.4).

4.1 Extraction of Timing Information

Before deciding on the locations of the PMHs in the code, prior timing knowledge about an application is needed. Software timing analysis is one way in which this information can be determined. In [16], the compiler estimates the performance of a piece of code in fine-grained execution time analysis. Other techniques use static and dynamic path analysis [15]. While such techniques could be used to guide the insertion of PMHs, it is beyond the scope of our work. Instead, we rely on program profiles to collect timing information at the source code level. The profiles should be collected on representative data input sets that show a variety of an application's behavior. The data set should include a worst-case scenario for the application being considered. In case an actual program execution time exceeds the worst case time from the training data set, a signal could be invoked to indicate the violation of a deadline.

For collecting the timing information in our approach, we divide the control flow graph (CFG) into regions. Each region contains one or more adjacent basic blocks. Since the typical size of a block is small (tens of cycles) compared to the interrupt interval (hundreds of thousands of cycles), some blocks can be aggregated into a region to be profiled. Regions are determined by a region construction algorithm described in [2]. The basic idea of region construction is to isolate basic blocks containing procedure calls (as their execution times contribute to the time of the basic block) in a separate region. Other basic blocks can form a single region unless one or more basic blocks containing procedure call(s) is part of the same program statement.

After forming regions, profiling is used to extract timing information about the program. To estimate the timing information for loops at run time, we collect the maximum cycle count of loop segments along with the maximum trip count of that loop. Note that we refer to *loop body* as the piece of code iteratively executed inside the loop, while *loop segment* includes the total execution cycles of the loop and the *trip count* is the number of iterations in the loop. For each procedure, we collect the time spent executing that procedure. Since the PMH placement algorithm traverses the modified CFG region-by-region, we also collect the worst-case cycles of each region. High level information (worst and average case cycle count) about the entire application is used to compute the interrupt interval size as described in Section 4.2.

4.2 Setting the Interrupt Interval Length

To determine the interval for invoking PMPs, we view a program's entire execution as sequential computation to determine the interrupt interval size in cycles. This approach is based on our prior work [1] that devised a theoretical formulation of speed computation for sequential code. With the knowledge of the worst and average case execution cycles of the application and with the overhead of the speed computation and voltage change considered, we find an estimate of the optimal number of PMPs that should be inserted to consume minimum energy. By dividing the total worst-case cycles of the application by the number of PMPs, we get the best interval size (in cycles) to execute the PMPs.

During run-time, whenever a speed change is encountered in the service routine, the interrupt timer is updated according to the new frequency. For the rest of this paper, the terms "interrupt interval" and "interval" are used interchangeably.

4.3 Insertion of PMHs

After computing the interrupt interval, the goal of the hint-placement algorithm is to ensure that a hint is executed before the invocation of the next interrupt instance. The placement algorithm traverses the CFG to insert PMHs no further apart than the size of the interrupt interval. Note that hints are inserted in terms of the interrupt interval specified in cycles. Ideally, a hint should execute right before the ISR is invoked. Since we do not control the application's dynamic behavior, more than a single hint can be executed in each interval to improve the accuracy of speed computation.

The basic idea of the hint-placement algorithm is 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 regions. A PMH is inserted in the code before this counter exceeds the interval length. The hint instrumentation is inserted in the appropriate locations and the counter is reset. PMH locations are selected based on the code structures: sequential code, branches, loops or procedure calls. Next we describe the criteria of placement in each of these cases.

4.3.1 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. Sequential placement inserts a hint just before ac exceeds the interval size. It is non-trivial to insert a hint when a region contains a procedure call, since the procedure's cycles is accounted for in the enclosing region's execution time. If the called procedure is too large, then inserting hints only at the region boundary is not sufficient to update the WCR time before the next service routine 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, hints are inserted according the following cases:

- When the cumulative counter ac exceeds the interrupt interval cycles and there are no procedure calls in the region then a hint is placed before the current region.

- If a region contains a procedure call and the body of a called procedure exceeds the interval size, a hint is placed before the procedure call and another hint after the procedure call. The procedure is marked for later placement by adding it to a table that we call the *Procedure Placement Table* (PPT). The hint before the call indicates the WCR at the start of this

procedure execution. Computation of WCR is path-dependent; that is, its value is dependent on the path this procedure instance is called from. The hints before and after the called procedure that contains hints simplify the PMH placement scheme by eliminating the need for any inter-procedure placement of hints (i.e., ac does not have to be remembered from one procedure to the next).

4.3.2 Branches

For any branch structure, each of the individual branch paths 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 paths. In a join, ac is set as the maximum value of propagated counters of all the joined branches.

4.3.3 Loops

The decision of placing hints in a loop is based on the profiled loop segment and loop body sizes. The different cases for inserting PMHs in loops are as follows:

- If the sum of ac and loop segment exceeds the interval but the loop segment is smaller than the interval then one hint is placed before the loop.
- In case a loop segment exceeds the interval but the loop body is smaller than the interval, then a PMH is placed at the beginning inside of the loop body in addition to the PMH placed before the loop. Another hint is inserted after the loop exit.
- If the size of the loop body exceeds the interval, a hint 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.

The reason for placing a hint after the loop in the last two cases is to adjust any over estimation of the WCR done by the loop hints. Over-estimation of the WCR is possible in the case when the loop bounds are unknown during loop execution.

4.3.4 Procedure calls

In the processing of sequential segments, procedure calls are detected and also the selection of the procedures that require hint placement takes place (by storing them in the PPT). The procedure selection is subject to satisfying the need for updating the WCR (through hints) at least once during each ISR interval. For each procedure processed from the PPT, instrumentation code is inserted in the procedure prologue to retrieve the WCR computed dynamically by the hint located before the call. The instrumentation is necessary because a procedure may be called from multiple sites (in different program paths) and each one can result in a different WCR value for the called procedure.

At run-time, the hints placed before procedure calls compute the WCR of procedure instances. This value is stored in the procedure's stack space. Each hint located inside this procedure uses this value in its calculations. This is beneficial in case of recursive calls, where each call instance has its own WCR estimate.

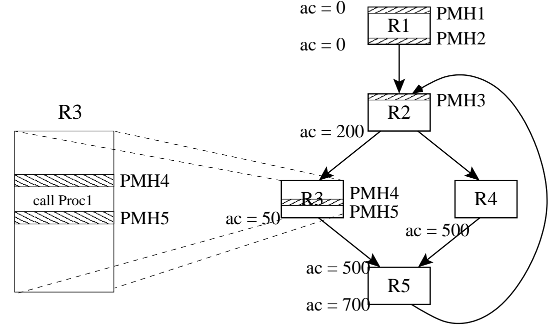

Figure 2: Example of the PMH placement methodology in a simple CFG.

Table 1: The regions' timing information

Region#	1	2	3	4	5
WCC	500	200	1200	300	200
Loop body		1600	max. iter. count		10
Procedure size		1100	disp of Proc1		50

4.4 Run-Time PMH Execution

The placement algorithm can insert two different types of hints based on a program structure. The two types compute the WCR differently based on whether the PMH is located inside a loop or not.

4.4.1 Static PMH

This type of hint is placed in any location in the code outside a loop body. A PMH inside a procedure computes the WCR based on the WCR at the start of the procedure instance, p_wcr. Since the execution path is only known at run-time, a procedure instance retrieves its p_wcr stored in its stack's space as discussed in Section 4.3. A static hint computes the WCR by subtracting the displacement of the hint location from p_wcr. For example, in a PMH inserted 100 cycles from the beginning of the procedure, the WCR is computed as $p_wcr - 100$.

4.4.2 Index-controlled PMH

Hints of this type are inserted inside the body of a loop where the WCR varies according to the current loop iteration counter. The hint computes the WCR based on equations from [15]. WCR is computed as $wcr_before_loop - (iteration_count * loop_body) - ldisp$. The wcr_before_loop value is determined at run-time from the hint that precedes the loop. The average $loop_body$ is computed as $(loop_segment/\ maximum_iteration_count)$. The displacement, $ldisp$, is computed from the start of the loop body. This technique can also be used in case of nested loops.

4.5 Example of PMH Placement

This example shows how the placement algorithm works for a simple CFG as in Figure 2, where each rectangle represents a region. Assume the interrupt interval length equals 1000 cycles. We first give the details on how the algorithm selects the hints' locations and then the details of each inserted hint. The timing information for the CFG is listed in Table 1.

Table 2: The inserted PMHs details

no.	Type	Computed as
1	static	p_wcr - 0 (= 16500 for this procedure instance)
2	static	p_wcr - 500 (= 16000)
3	index-controlled	16000 - (iteration count x 1600)
4	index-controlled	16000 - (iteration count x 1600) - 250
5	index-controlled	16000 - (iteration count x 1600) - 1350

PMH1: A hint is placed at the beginning of the procedure, indicating the WCR (= 16500 cycles).
PMH2 and PMH3: Since R2 starts a loop with a segment size of 16000 that is larger than the interval size, then PMH2 is placed at the end of R1 (before the loop). Because the body of the loop exceeds the interval, PMH3 is placed at the start of R2 (inside the loop).
PMH4 and PMH5: Because the sum of ac and R3 cycles is larger than *interval*, the algorithm looks for the first called procedure (i.e., `Proc1`) in the region R3 which is located at 50 cycles from the top of R3. Since the procedure body is larger than *interval* then PMH4 and PMH5 are placed before and after the procedure call, respectively. An entry corresponding to `Proc1` is inserted in the runtime table (discussed in Section 4.3). The hints placed inside procedure `Proc1` are not shown in this example.

Assuming that the presented CFG is the application's main procedure, p_wcr = application's WCR = 16500. Table 2 lists the details of each inserted hint. Note that PMH2 is useful in evaluating the WCR just before the loop, wcr_before_loop.

5. EXPERIMENTS

We evaluate the efficacy of our scheme with respect to reducing the energy consumption in processors. For this, we use the Simplescalar micro-architecture toolkit [17] with the configuration shown in Table 3. We extended the *sim-outorder* simulator 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 the cycles for executing the application as well as the cycles for computing the speed in PMPs and the WCR in PMHs. We also considered the overhead of setting the speed by adding a constant energy overhead for each speed change to the total energy. In the experiments below, we used the Transmeta Crusoe and the Intel XScale processor speed models. The Transmeta Crusoe has 16 speed levels and the XScale has five levels. The voltage/frequency ranges of the processors are discussed in Section 2. The energy consumed in other subsystems is beyond the scope of this evaluation.

We emulate the effect of the ISR in Simplescalar by flushing the pipeline at the entry and exit of each routine. The ISR computes a new frequency at each interval and then sets the corresponding voltage. To compute the speed, we use a dynamic *greedy* and a static speed management schemes presented in [7]. The greedy scheme uses all the available slack to reduce the speed for the next *interval*, while the static speed management runs the entire application on a single CPU speed based on the static slack. Our base case is with no power management (i.e., execute at maximum speed). We also compare our scheme with two schemes that use only PMPs; the first (*PMP-pcall*) places a PMP before each procedure call, and the second (*PMP-placement*) inserts PMPs according to our hint placement algorithm to show the potential benefit of using hints.

We show the experimental results of two time-sensitive applications; automated target recognition (ATR) [8] and an MPEG2 decoder [21]. 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 due to the introduction of more slack that can be used to reduce the speed/voltage of the processor.

Although the schemes that use only PMPs have better estimation of the WCR at each PMP invocation, the overhead introduced for such estimation and computation of a new speed occurs very often that may overshadow the benefit of a potential voltage scaling.

Automatic target recognition (ATR)

The ATR application[2] does pattern matching of a target in data frames (images). We experimented with 190 actual data frames. The number of target detections in each frame varies from zero to eight detections. A frame processing time is proportional to the number of detections within that frame. Each frame has a certain time interval to be processed. In Figure 3, the greedy scheme utilizing hints consume less energy than the static scheme because of dynamic slack reclamation.

Since the static scheme operates on a single speed throughout a frame processing. As deadline increases, the system load decreases and the processor operates on a discrete lower performance level. This explains the step-like shape of the normalized energy of the static scheme. The number of steps increases in the Crusoe over the XScale model as XScale has fewer performance levels to operate on. The energy of the static power management may exceed the no power management at full load due to the introduced overhead of setting the speed once at the start of each frame.

The overhead in the PMP-pcall scheme is high due to the large number of called procedures in ATR. It is also the case that most of the executed PMPs do not yield a speed change, and thus no energy saving is achieved from these PMPs. Due to this overhead, the scheme may consume more energy than the static scheme (as in Figure 3-b). The PMP-placement scheme has close energy consumption to our scheme.

[2]The code and data were provided by Northrop Grumman/Vanderbilt University(ISIS).

Table 3: The simplescalar configuration

fetch width	4 inst/cycle	decode width	4 inst/cycle
issue width	4 out of order	commit width	4 inst/cycle
RUU size	16 inst	LSQ size	8 inst
FUs	4 int, 1 int mult/divide, 4 fp - 1 fp mult/divide		
branch pred.	bimododal, 2048 table size		
L1 D-cache	512 sets, 32byte block, 4byte/block,1 cycle		
L1 I-cache	512 sets, 32 byte block, 4byte/block,1 cycle		
L2 cache	1024 sets, 64byte block, 4byte/block,1 cycle		
memory	18 cycle hit, 8 bytes bus width		
TLBs	instruction:16 sets, 4096 byte page - data:32 sets, 4096byte page		

MPEG2 decoder

We collect timing information about the MPEG2 decoder using a training data set of six files and test it on a set of 20 different data files. The scheme inserts PMHs in the decoder's code based on the profiled information about frames of the training set movies. We run experiments for four different frame rates: 15, 30, 45 and 60 frame/sec. Those correspond to deadlines 66, 33, 22, and 16 msec per frame. In our experiments the deadlines were met for each frame in the tested movies. Figure 4 shows the results of the energy consumption for three of the test movies for Transmeta and XScale. Similar to ATR results, our proposed 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 hints is much larger than the invoked PMPs. Hence the WCR estimation is improved. In comparison with PMP-placement technique (replacing the hints with PMPs), the overhead increases which overshadows the gain of the few extra speed adjustments. The PMP-pcall scheme has the highest energy consumption due to the large number of procedure calls within loops in this application.

Overhead of the scheme

Increasing the number of inserted instructions could have a negative effect on the cache miss rate and the total number of executed instructions. Both factors 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 very small. There is no appreciable increase in cache misses (instruction and data) for our benchmarks. For example in ATR, the absolute number of misses in the data cache increased by 0.02%, level two cache misses increased by 0.01%. Instruction caches misses increased by 20%, however this increase is only 445 misses (on average) which are most likely to be compulsory misses. Hence we do not expect a negative effect on the memory energy consumption due to our scheme.

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 invoking the ISR (PMP) at relatively large intervals (ranges from 250 to 580 Kcycle in the presented applications) and avoiding the excessive insertion of PMHs. For example in ATR application, for each interrupt interval, only extra 533 instructions are executed on average (including all executed PMHs and a PMP in this interval[3]). The total increase in the number of executed instructions due to the overhead is 0.05% for ATR and 0.25% for MPEG. This code increase contributes to an increase in the number of cycles that ranges between 0.19% to 0.4% for ATR and 0.4% to 1.7% for MPEG. Note that the total overhead cycles decreases 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. Examples on work that implements DVS in non time-critical applications include [4, 6]. An operating system solution proposed in [4], periodically invokes an interrupt that adjusts the processor speed in order to maintain a desired performance goal. The OS keeps track of the of the accumulated application's instruction level parallelism throughout the application execution time. In [6], 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. Work presented in [12] 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, work in [5] determines a voltage scheduling that changes the speed within the same task/application based on task's statistical behavior. The work in [10] modified the EDF and RMS in RT-Linux to incorporate voltage scheduling.

References [13, 3, 12] deal with applying a compiler controlled DVS in time-sensitive applications. Authors of [12] introduced a dynamic voltage scaling technique that takes place at the boundary of loop nests. In [3] the compiler inserts checkpoints at the start of each branch, loop, function call, and normal segments. Each code segment initiated with a checkpoint constitutes a node in a hierarchical CFG. At each checkpoint encountered during runtime, informa-

[3] An inserted hint can be executed more than once in a single interval, for example if placed inside a loop.

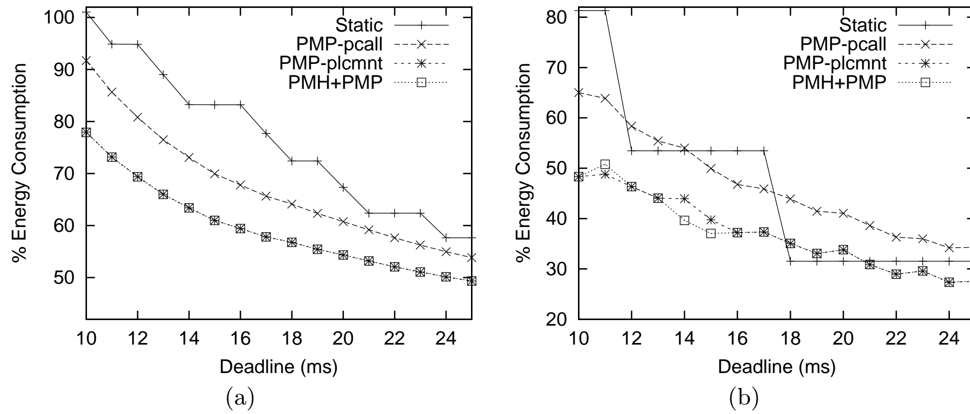

Figure 3: Average energy consumption normalized to the no power management scheme for ATR employing the (a) Transmeta Crusoe and (b) Intel XScale processor models.

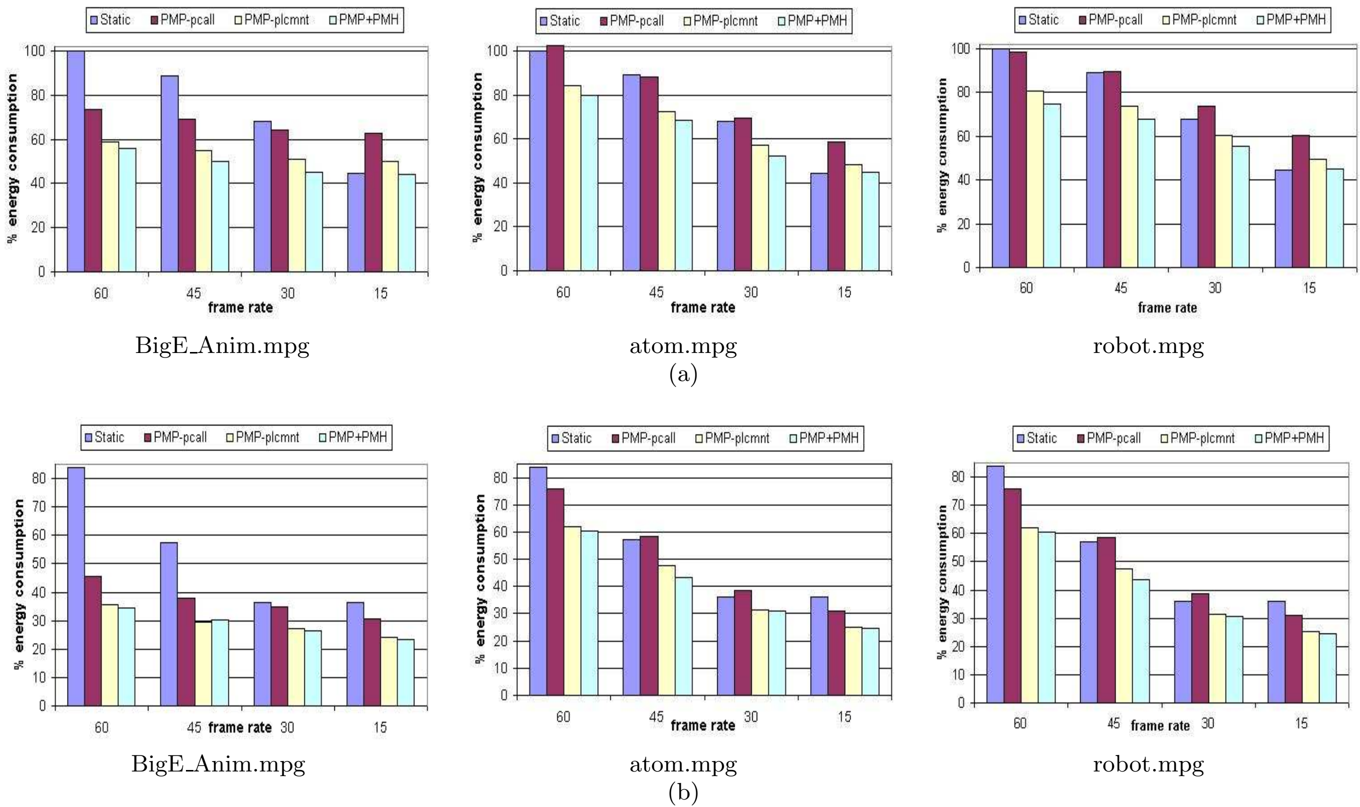

Figure 4: Average energy consumption normalized to the no power management scheme for the MPEG2 decoder employing (a) Transmeta Crusoe and (b) Intel XScale models.

tion about the checkpoints along with profile information is 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). The authors proposed pruning the hierarchical CFG to reduce the runtime overhead by merging small execution and variation nodes, but it was not clearly explained what nodes to merge. In [13], each basic block in the constructed CFG is augmented with its worst-case execution along with the WCR till the end of the program. "Branches in CFG that drop the remaining worst case faster than the execution rates are selected as locations to insert the speed change procedure". The speed update ratio is expressed in terms of the difference in the remaining worst case cycles between the departed node and its selected successor. However they did not mention how to deal with procedure and function calls especially when they are called from different paths; the WCR cycles of these procedures would be dependent on the path they are called from.

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 worst case execution time is presented in [11]. Some of these tools use static analysis to produce a discrete worst case bound. On the other hand, parametric analysis for computing the worst case as in [15] evaluates expressions in terms of parameter variables carrying information about some program segments; e.g., worst case execution time of loops is presented by symbolic formulas that are evaluated at runtime when the number of loop iterations is determined.

7. CONCLUSION

In this paper, we presented a hybrid compiler-operating system scheme for reducing the energy consumption of time-sensitive embedded applications running on processors with dynamic voltage scaling. The operating system periodically adapts processor performance based on the dynamic behavior of an application. Information about how the application is behaving is gathered through 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 power management hints in an application code to collect accurate timing information for the operating system. Finally, we presented results that show the effectiveness of our scheme over other schemes that utilize PMPs alone. Our results showed that our scheme is up to 79% better than no power management and up to 50% better than static power management on several embedded applications. Improvement over schemes that use only PMPs is dependent on the PMP placement strategy.

We are currently implementing our techniques for timing extraction and power management hint insertion in a post link-time inter-procedural optimization framework.

8. 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, COLP01.* 2001.

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

[3] 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.

[4] Childers B, Tang H, Melhem R. Adapting processor supply voltage to instruction-level parallelism. *Koolchips 2000 Workshop, during MICRO-33.* 2000.

[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] Hsu C, Kremer U. Single vs multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches. *PACS.* 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] Saputra H, Kandemir M, Vijaykrishnan N, Irwim M, Hu J, Hsu C-H, Kremer U. Energy-Conscious Compilation Based on Voltage Scaling. *LCTES'02-SCOPES'02.* 2002.

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

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

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

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

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

[18] Web-Feet Research. http://sanjose.bizjournals.com/sanjose/stories/2001/08/06/daily5.html

[19] Computer Industry Almanac http://www.c-i-a.com/pr1002.htm

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

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

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

9. GLOSSARY OF USED ACRONYMS

ATR : Automatic target recognition

CFG : Control flow graph

DVS : Dynamic Voltage scaling

ISR : Interrupt service routine

PMH : Power management hint

PMP : Power management point

PPT : Procedure placement table (for hint placement)

WCC : Worst-case execution cycles.

WCR : Worst case remaining cycles (till the end of an application)

p_wcr : WCR at the start of a procedure instance.